Implementation

Arthur J. Redfern
arthur.redfern@utdallas.edu

Disclaimer

- Previous math lectures
  - A broad presentation of material (linear algebra, calculus, probability and algorithms)
  - Perhaps a little biasing from me in terms of presentation, importance and intuition
  - But in general the material stands on it's own and there's not ambiguity at our level of review

- Previous network design and training lectures
  - A broad presentation of material (design and training)
  - A little more biasing from me in terms of presentation, importance and intuition
  - But pointers to many many references were provided for you to go deeper on any topic

- This series of network implementation lectures
  - We'll still cover a lot of topics, but the presentation of material will be more narrow
  - You'll get more of my opinion in terms of the right way to do things
  - But pointers will still be given to additional references and you're free to draw your own differing conclusions

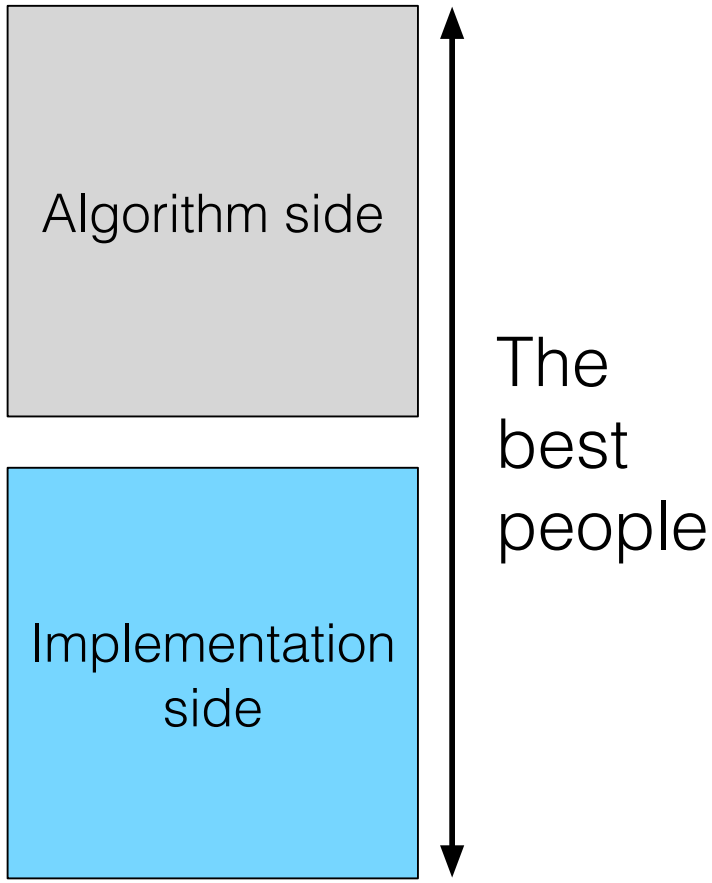
Outline

- Motivation
- Networks
- Software
- Hardware
- Performance
- Backup
  - Example software
  - Example hardware

Motivation

Why Discuss Implementation

- Progress in xNNs is directly linked to progress in improved implementations
- At large companies with big ML related business
  - About 1/2 the people are on the algorithm side
  - About 1/2 the people are on the implementation side
- The best people understand both
  - I want you to understand both



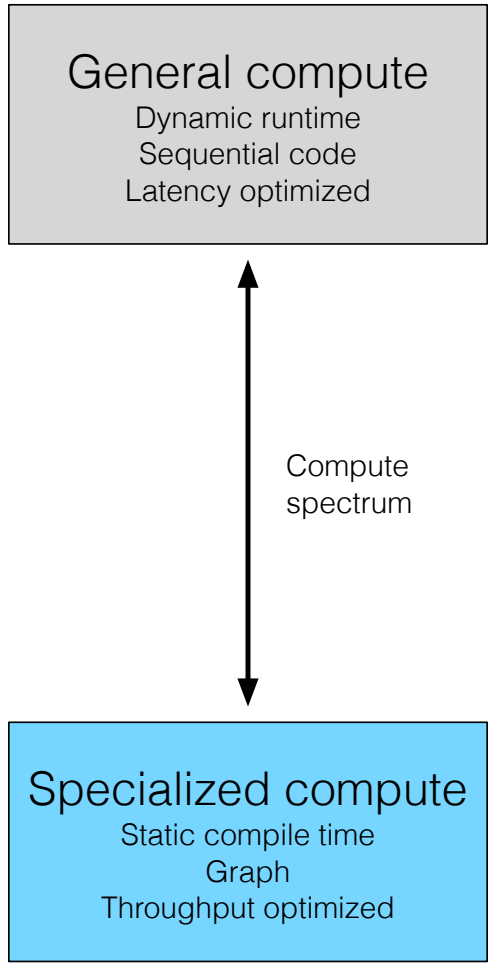
The Future Of Hardware And Software

Yes, that's a slightly grandiose slide title / slight exaggeration; no, it's not that far from the truth

- Is a bifurcation where only 2 points matter
  - Big code, small general compute
  - Small code, big specialized compute

- Big code, small general compute → map to host (x86, ARM, RISC-V, ...)
  - Hardware agnostic software
  - Runtime intelligent hardware
    - Cache, branch prediction, out of order processing, speculative execution, ...
    - This has been beaten to death, gains are small and incremental; you're picking up crumbs
  - Examples: high level operating systems, control code, ...

- Small code, big specialized compute → map to ~ DSA
  - Compile time intelligent software
  - Runtime deterministic hardware
    - This is where the action and ability to differentiate in hardware is
  - Examples: xNNs, almost all other technologies you're going to be interested in, ...

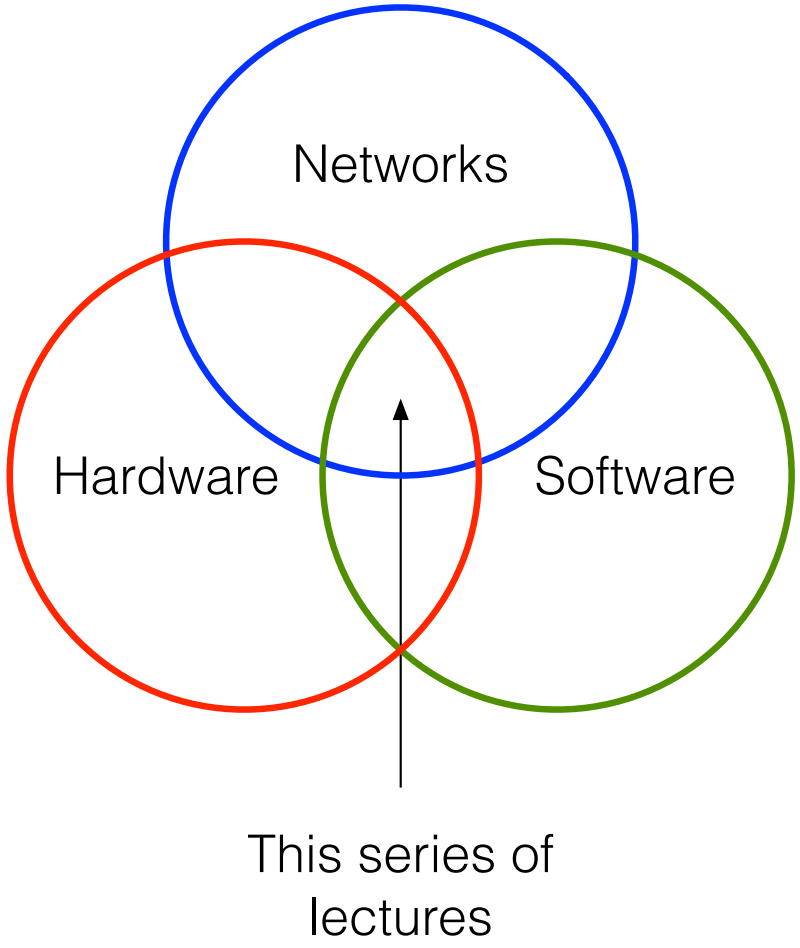


Co Design For Optimality

- Design networks for optimal performance on hardware
  - Layer sizes and operations designed to efficiently map to hardware
  - Sparsity, quantization and compression to reduce memory, reduce data movement and improve compute
- Design software to optimally map networks to hardware
  - User specifies part of a high level hardware agnostic graph as a starting point
  - Software tools complete the high level graph specification and optimally map the high level graph to hardware
- Design hardware to be an optimal target for networks
  - Memory sized such that most feature maps remain on device
  - DMA allowing background data movement for remaining off device transfers in parallel with foreground compute
  - Computational primitive approach to big compute for ASIC efficiency with mathematical generality
  - Small general host approach for handling a long tail of operations and future proofing
  - Deterministic control via sequencing through a low level compile time optimized graph

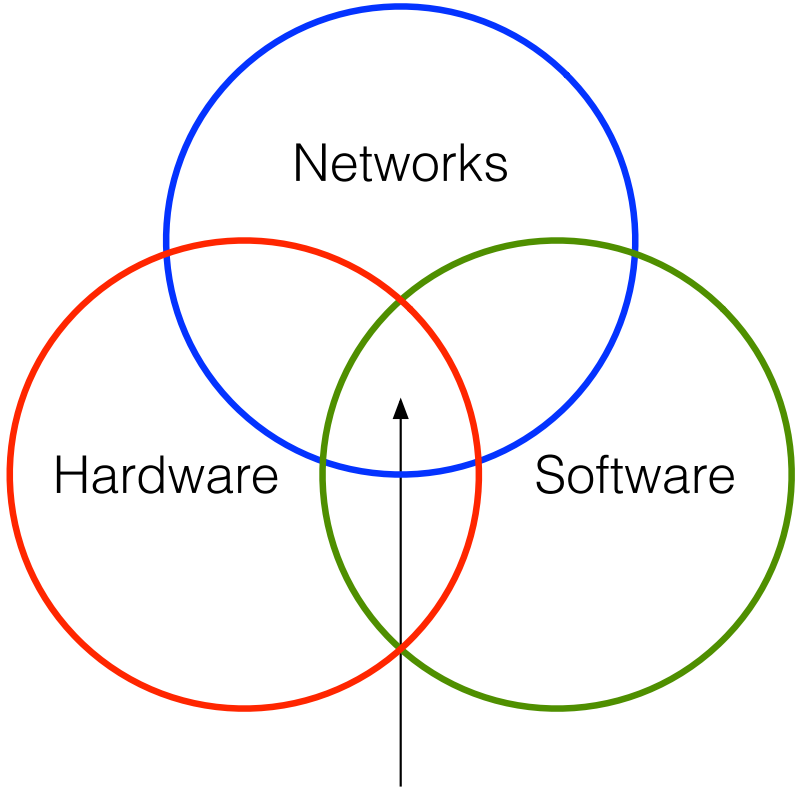
A Lecturer's Apology

- Presentation of material in this lecture is sequential
  - Networks
  - Software
  - Hardware
- But all of these topics are actually optimized together at the same time
  - As such, there are dependencies in the material
- A suggested slide reading strategy to address this interdependence
  - Start with a bit of a high level view of everything
  - Then sequential presentation of topics in more detail from me
  - Then go back and re read having seen everything once before



Preview

- Network
  - Complexity
  - Sparsity
  - Quantization
  - Compression
- Software
  - Graph specification
  - Graph compilation
  - Graph execution
- Hardware
  - Physics
  - System on a chip architecture
  - Domain specific architecture
  - Network architecture



Networks

Networks Outline

- Complexity
- Sparsity
- Quantization
- Compression

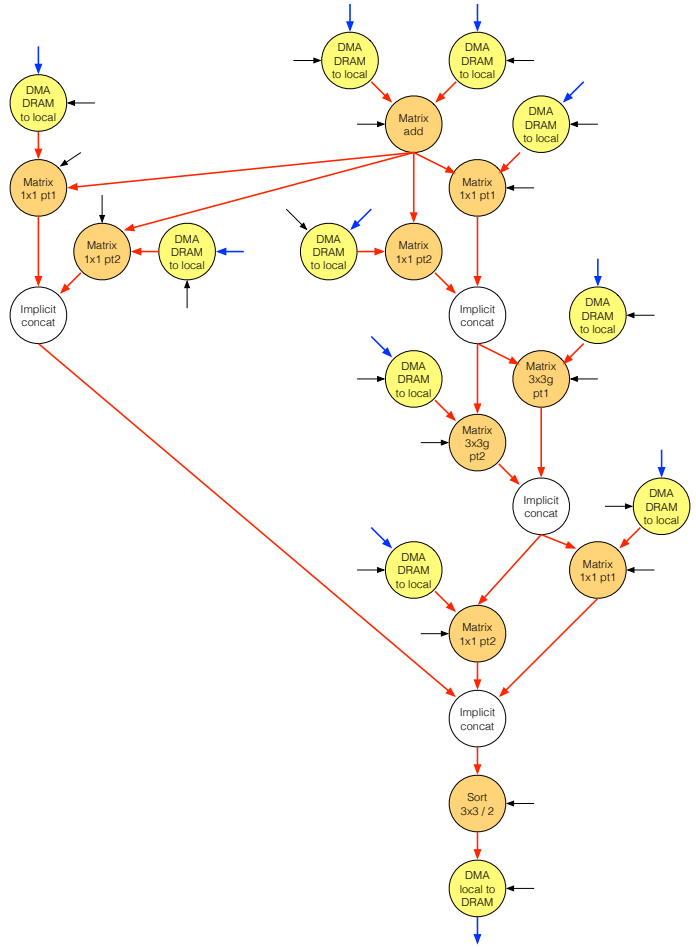
Flow

- Design networks with memory and compute complexity well matched to target hardware
- If the hardware is able to take advantage of sparsity
  - Encourage a target level of sparsity in the filter coefficients during training (e.g., L1 regularization and thresholding)
  - Use ReLU or ReLUx nonlinearities to encourage 0s in feature maps
- If the hardware is able to take advantage of quantization
  - Use ReLUx nonlinearities that limit feature map ranges to make it easier to quantize feature maps; pay special attention to locations in the network where feature maps add or (practically also) concatenate as the input feature maps to these operators need to have the same quantization scales
  - Ideally quantize filter coefficients during training (e.g., start at 32b float, gradually deflate to target precision); multiplicative coefficients are easier, additive coefficients may need additional range but that's usually ok from a complexity perspective; post training quantization is also possible in some cases but typically not as good
- If the hardware is able to take advantage of compression
  - Filter coefficients after training are static and compression can be optimized for specific cases
  - Feature maps are dynamic, compression typically exploits 0s or non uniform densities and correlations for data reductions
  - Compression needs to be balanced against / coordinated with the extra latency that it typically introduces

Networks – Complexity

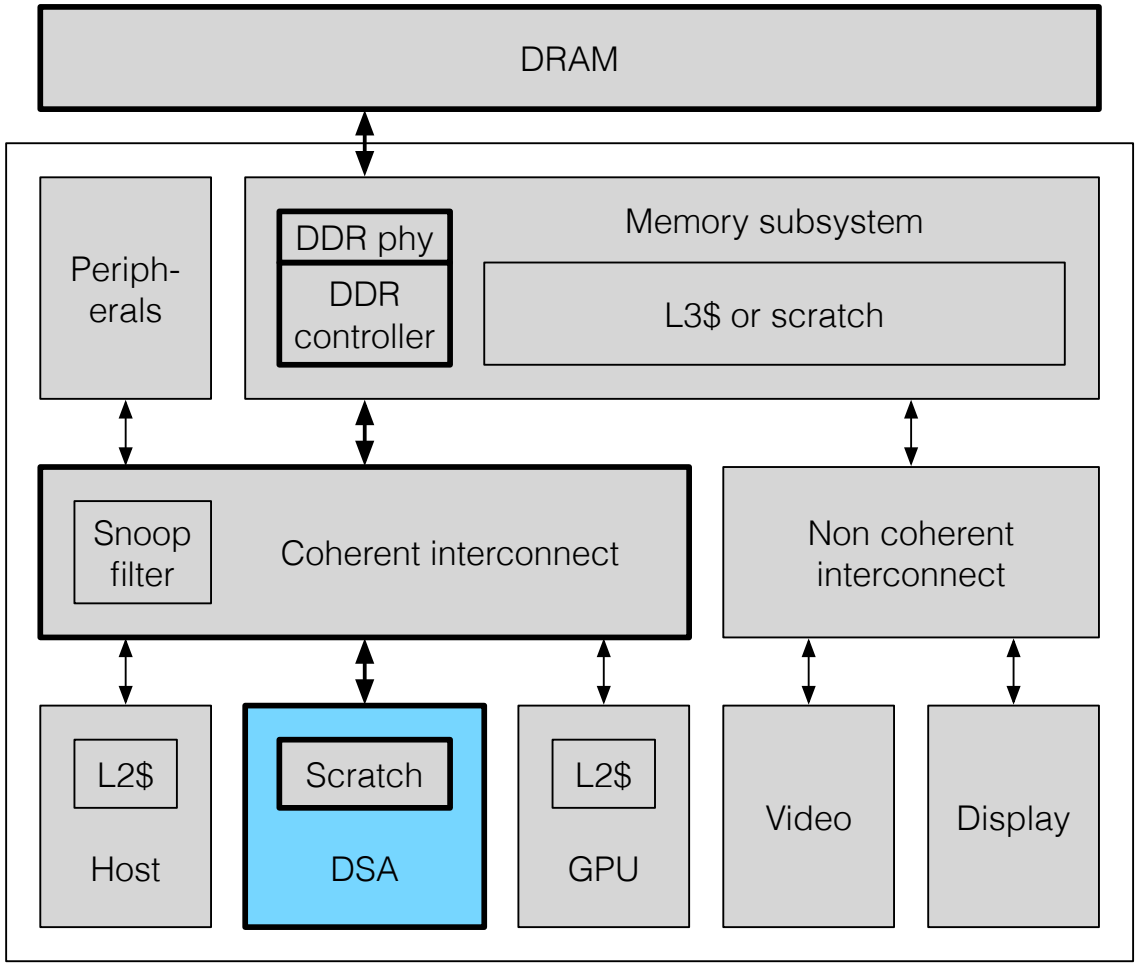
Bookkeeping

- Complexity
  - Typically learn about complexity in terms of order of operations
  - Here we're going to count operations exactly
  - All scale factors, constants, precision, memory, … matter
- Starting point
  - High level graph, edges are memory and nodes are operators
  - Generic model of off device memory, off device to on device bandwidth, on device memory and compute next to on device memory
- Track
  - Memory per edge
  - Operations per node (also type)



Bookkeeping

- Complexity
  - Typically learn about complexity in terms of order of operations
  - Here we're going to count operations exactly
  - All scale factors, constants, precision, memory, … matter
- Starting point
  - High level graph, edges are memory and nodes are operators
  - Generic model of off device memory, off device to on device bandwidth, on device memory and compute next to on device memory
- Track
  - Memory per edge
  - Operations per node (also type)



Theoretical Complexity

- Model complexity is to a 1st order approximation proportional to input pixels
  - True for CNN style 2D convolution
  - True for pooling

- Compute and filter parameter memory of a CNN style 2D convolutional layer is proportional to the square of the number of feature maps
  - 2x input / output feature maps → 4x compute and 4x filter parameter memory
  - Note that the above calculation is without grouping
  - With grouping that maintains the same number of input and output feature maps per group and just increases the number of groups the complexity increase is back to proportional

CNN style 2D convolution

MACs (assuming F – 1 pad) =
$N_i N_o F_r F_c L_r L_c$

Filter memory =
$N_i N_o F_r F_c$

Feature map memory =
$(N_i + N_o) L_r L_c$

Hardware Size Vs Model Size

- A models run on hardware of a given size

- Hardware size vs model size leads to 3 possibilities with respect complexity from an efficiency perspective
  - Too small to fully exercise compute resources
  - Optimally sized to fully exercise compute resources and feature maps fit fully on device
  - Too big for feature maps to fit on device and off device data movement increases nonlinearly

Training Vs Testing

Training

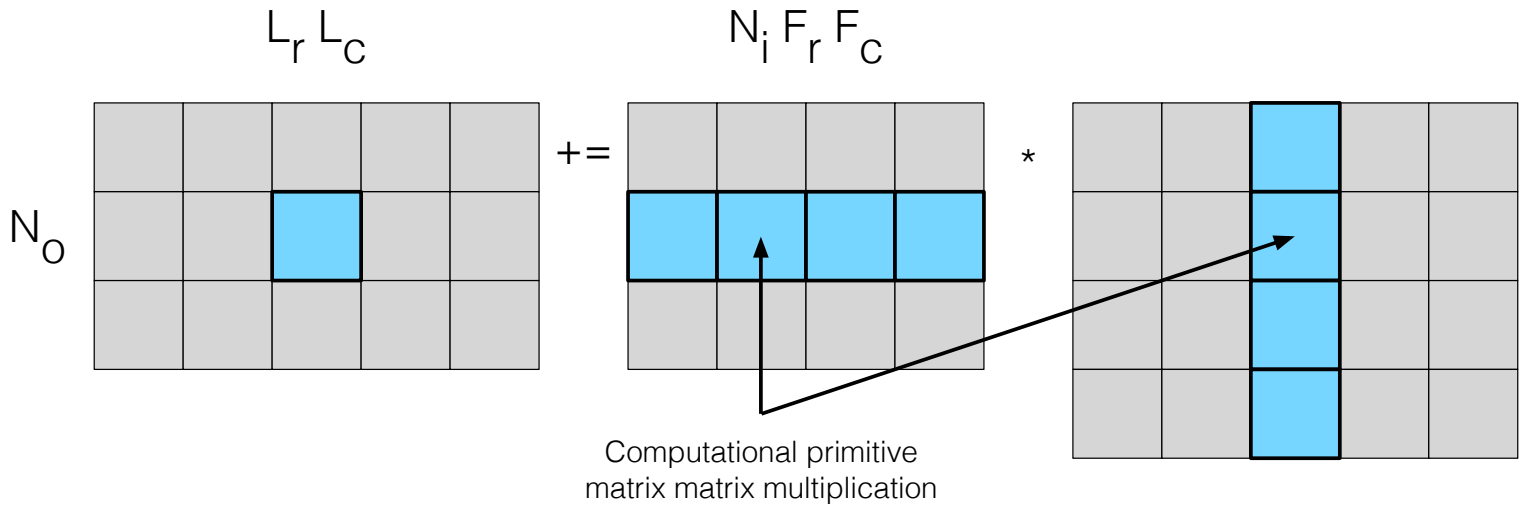
- Can batch inputs (allowing the amortizing of weight movement across multiple inputs)
- Need batch norm operations
- Need error calculation
- Need to maintain memory space for reverse mode automatic differentiation so there's less reuse of memory
- Typically need higher precision floating point

Testing

- Sometimes can batch inputs; sometimes process 1 input at a time for latency reasons
- Absorb batch norm operations
- Need network output
- Don't need to maintain memory space for reverse mode automatic differentiation so there's more reuse of memory possible
- Frequently ok with lower precision fixed point

CNN Style 2D Convolution Computation

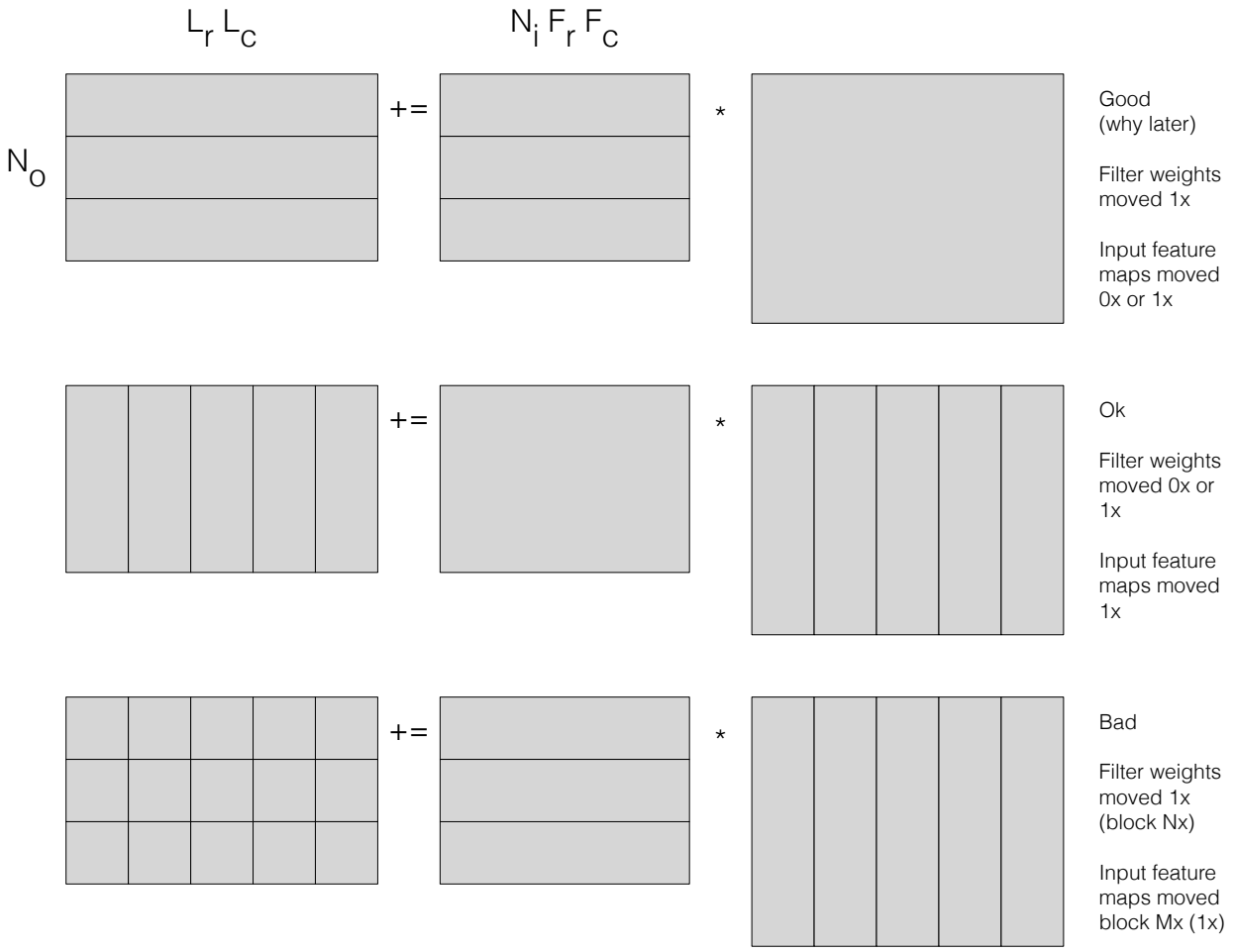
- A preview
  - We're going to specify hardware with an optimized matrix matrix multiplication primitive
  - Large matrix matrix multiplication is going to be implemented via tiled smaller matrix matrix multiplication
  - Data movement limits compute
- Tiling efficiency
  - In this figure N_o, L_r L_c and N_i F_r F_c are all integer multiples of the tile size
  - Excess compute (inefficiency) occurs when they're not
  - Extreme case: fully grouped $N_i = N_o = 1$



CNN Style 2D Convolution Data Movement

- A preview
  - We're going to specify hardware with a fixed amount of on device memory
  - Compute is out of on device memory
  - Off device to on device data movement is slow and consumes a lot of power

- Memory efficiency
  - A good situation is that input feature maps for a given layer fit on device and filter coefficients are block row moved for that layer
  - An ok situation is that filter coefficients can fit fully on device for a given layer and input feature maps are block row moved
  - A bad situation is that neither input feature maps or filter coefficients fit fully on device for a given layer and multiple moves are needed



Rules Of Thumb

- Memory
  - Early in the network feature maps tend to dominate memory
  - Later in the network filter coefficients tend to dominate memory
- Compute
  - Early in the network tends to dominate compute
  - Compute is typically proportional to data volume
  - Data volume shrinks by ~ 2 after every pooling stage (1/4 space, 2x channel)

CNN style 2D convolution

MACs (assuming F – 1 pad) =
$N_i\ N_o\ F_r\ F_c\ L_r\ L_c$

Filter memory =
$N_i\ N_o\ F_r\ F_c$

Feature map memory =
$(N_i + N_o)\ L_r\ L_c$

Simplifying Convolutional Layers

- Typically means reducing compute
- Examples
  - Random sparsity in filter coefficients (will discuss in next section)
    - Can be encouraged during training
    - Ex: using a threshold and a deflationary approach
  - Structured sparsity in filter coefficients
    - Can be forced in training
    - Ex: grouping
  - Input or output feature map channel reductions
    - Induces fewer filter coefficients
- Also means matching the shape of the convolutional layer optimally to the hardware

Simplifying Pooling Layers

- Typically means removing overlap when striding to allow for better memory locality and no buffering of overlap

- Examples
  - Use 2x2/2 vs 3x3/2

Simplifying Fully Connected Layers

- Typically means reducing memory

- Examples
  - Decompositions of the filter matrix that exploit structure
  - SVD / outer product based decompositions of the matrix that reduce the number of required parameters

Networks – Sparsity

Simplifying Networks Improves Performance

- Given a trained network network make structural modifications to reduce complexity
  - Modifications can be un structured perturbations
    - Random sparsity in filter coefficients
  - Modifications can be structured perturbations
    - Removal of whole feature maps
    - Remove of whole filters or more likely groups of filters
  - Modifications can change graph structure
    - Transforming 3 layers to 1

An incomplete laundry list of methods

- Optimal brain damage
  - http://yann.lecun.com/exdb/publis/pdf/lecun-90b.pdf
- Optimal brain surgeon and general network pruning
  - https://authors.library.caltech.edu/54981/1/Optimal%20Brain%20Surgeon%20and%20general%20network%20pruning.pdf
- Efficient and accurate approximations of nonlinear convolutional networks
  - https://arxiv.org/abs/1411.4229
- Accelerating very deep convolutional networks for classification and detection
  - https://arxiv.org/abs/1505.06798
- Learning efficient convolutional networks through network slimming
  - https://arxiv.org/abs/1708.06519
- To prune or not to prune: exploring the efficacy of pruning for model compression
  - https://openreview.net/pdf?id=Sy1iIDkPM

Simplify Or Design A Simpler Network?

- Taking an untrained network, reducing it's complexity via structural modifications and training it is not referred to here as network simplification
  - That’s really just simple network design and training
- Starting from a simple design is likely a better strategy than starting from a more complex design and simplifying
  - However, some places have found it easier to train a larger model first then simplify
  - But that could be because the structure of the simpler network was sub optimal in the first place
- However, just designing a simple network isn't practical if the simplifications of interest aren’t predictable at network design time
  - An example of this would be random sparsity in filters

Networks – Quantization

Precision Affects Practical Complexity

- Hardware feature map and filter parameter memory complexity of a convolutional layer is as a 1st order approximation proportional to the number of bits per element
  - Why compression is important
- Hardware compute complexity of a linear layer is as a 1st order approximation proportional to the square of the number of bits per element
  - Key operations include additions and multiplications
  - Multiplies dominate the practical hardware complexity calculation
  - Hardware adder complexity is to a 1st order approximation proportional to the number of bits
  - Hardware multiplier complexity is to a 1st order approximation proportional to the square of the number of bits
  - Why quantization is important

8 and 16 bit fixed point multiplication

Let x_8^{lo}, x_8^{hi}, y_8^{lo} and y_8^{hi} be 8 bit integers and let x_{16} and y_{16} be 16 bit integers such that

$$x_{16} = 2^8 x_8^{hi} + x_8^{lo}$$
$$y_{16} = 2^8 y_8^{hi} + y_8^{lo}$$

Then 16 bit integer multiplication can be implemented via 4x 8 bit multiplication, 3 shifts and 3 adds

$$x_{16}\, y_{16} = (2^8 x_8^{hi} + x_8^{lo})\,(2^8 y_8^{hi} + y_8^{lo})$$
$$= 2^{16} x_8^{hi} y_8^{hi} + 2^8 x_8^{hi} y_8^{lo} + 2^8 x_8^{lo} y_8^{hi} + x_8^{lo} y_8^{lo}$$

2x the number of bits → ~ 4x the integer multiplier complexity

Goal Is To Reduce Bits Per Memory Element

- The purpose of quantization
  - Reduce the number of bits per memory element to reduce memory and bandwidth requirements and improve (practical implementations of) computation while minimizing accuracy loss

- Rules of thumb
  - Memory is proportional to the number of bits
  - Bandwidth is proportional to the number of bits
  - Adder complexity is proportional to the number of bits
  - Multiplier complexity is proportional to the square of the number of bits
  - Comparator complexity is proportional to the number of bits

Data Formats

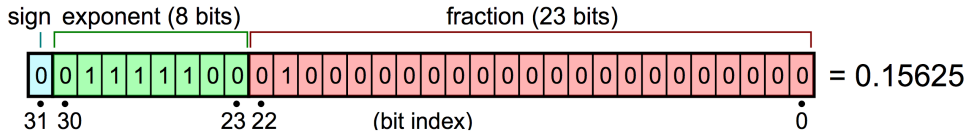
- Common data formats
  - 64 bit float 53.11 (IEEE 754 double)
  - 32 bit float 24.8 (IEEE 754 single)
  - 16 bit float 11.5 (IEEE 754 half; more precision less range than bfloat16)
  - 16 bit float 8.8 (bfloat16; more range less precision than IEEE 754 half)
  - 16 bit fixed (signed and unsigned)
  - 8 bit fixed (signed and unsigned)
  - ---------- (mainstream above line, research-y below line; becoming a candidate for analog)
  - 4 bit fixed
  - 2 bit fixed
  - 1.5 bit fixed {-1, 0, 1}
  - 1 bit fixed Nice for compactness but no 0

- Additional data formats that have positive hardware qualities
  - Power of 2 filter coefficients
    - Simplifies multiplication to shift and add
    - Results in non uniform step sizes which are not always good
    - But still use arbitrary precision for bias coefficients

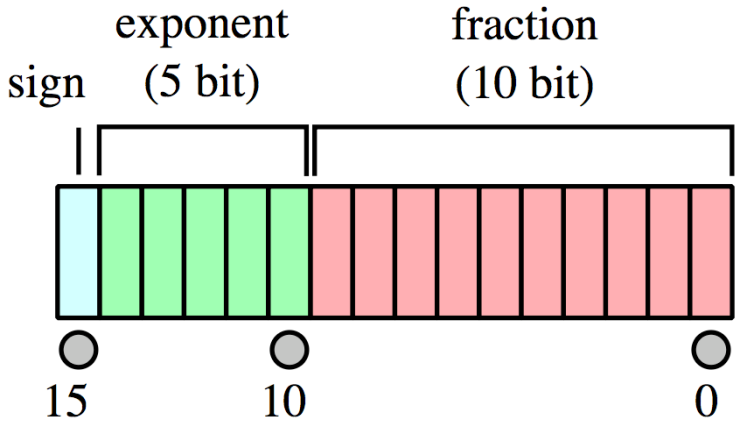
Data Formats For Network Training

Training typically needs more bits than testing; IEEE 754 32 bit float is most common now, bfloat16 will likely gain in popularity going forward; int8 with block scaling is also becoming more common to either integrate with training from the start or after an initial floating point warm up period

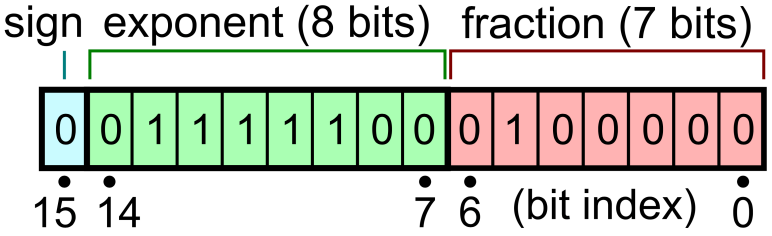
- IEEE 754 float 32: 1 bit sign, 8 bits exponent, 23 bits significand
  - Ignoring special values signaled by exponent values of 0 and 255
  - Value $= \text{sign} \times \text{range} \times \text{precision}$
    $= (-1)^{\text{sign}} \times 2^{\text{exponent} - 127} \times 1.\text{significand}_2$
    $= \{-1, 1\} \times 2^{\{-126, \ldots, 127\}} \times [1, 2)_{\text{23 bits precision}}$



- IEEE 754 float 16: 1 bit sign, 5 bits exponent, 10 bits significand
  - Ignoring special values signaled by exponent values of 0 and 31
  - Value $= (-1)^{\text{sign}} \times 2^{\text{exponent} - 15} \times 1.\text{significand}_2$
    $= \{-1, 1\} \times 2^{\{-14, \ldots, 15\}} \times [1, 2)_{\text{10 bits precision}}$



- bfloat 16: 1 bit sign, 8 bits exponent, 7 bits significand
  - Ignoring special values signaled by exponent values of 0 and 255
  - Value $= (-1)^{\text{sign}} \times 2^{\text{exponent} - 127} \times 1.\text{significand}_2$
    $= \{-1, 1\} \times 2^{\{-126, \ldots, 127\}} \times [1, 2)_{\text{7 bits precision}}$



Figures from https://en.wikipedia.org/wiki/IEEE_754 and https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

Floating Point To Fixed Point Conversion

Signed fixed point is an integer in $\{-2^{bits-1}, ..., 2^{bits-1} - 1\}$ and unsigned fixed point is an integer in $\{0, ..., 2^{bits} - 1\}$

- Signed example for $\{\mathbf{X}_q, s_X\}$ = quantize($\mathbf{X}$, bits)
  - Target range of $\{-2^{bits-1} - 1, ..., 2^{bits-1} - 1\}$
    - Keeping symmetric about 0 by ignoring the value -2^{bits-1}
    - For 8 bits this is {–127, ..., 127}
    - Assume scale s_X is chosen such that there's no clipping
  - maxAbsX = max(|$\mathbf{X}$|)
  - s_X = maxAbsX / $(2^{bits-1} - 1)$
  - $\mathbf{X}_q$ = round($\mathbf{X}$ / s_X)

- Unsigned example for $\{\mathbf{X}_q, s_X\}$ = quantize($\mathbf{X}$, bits)
  - Target range of $\{0, ..., 2^{bits} - 1\}$
    - For 8 bits this is {0, ..., 255}
    - Assume input $\mathbf{X}$ is non negative
    - Assume scale s_X is chosen such that there's no clipping
  - maxX = max($\mathbf{X}$)
  - s_X = maxX / $(2^{bits} - 1)$
  - $\mathbf{X}_q$ = round($\mathbf{X}$ / s_X)

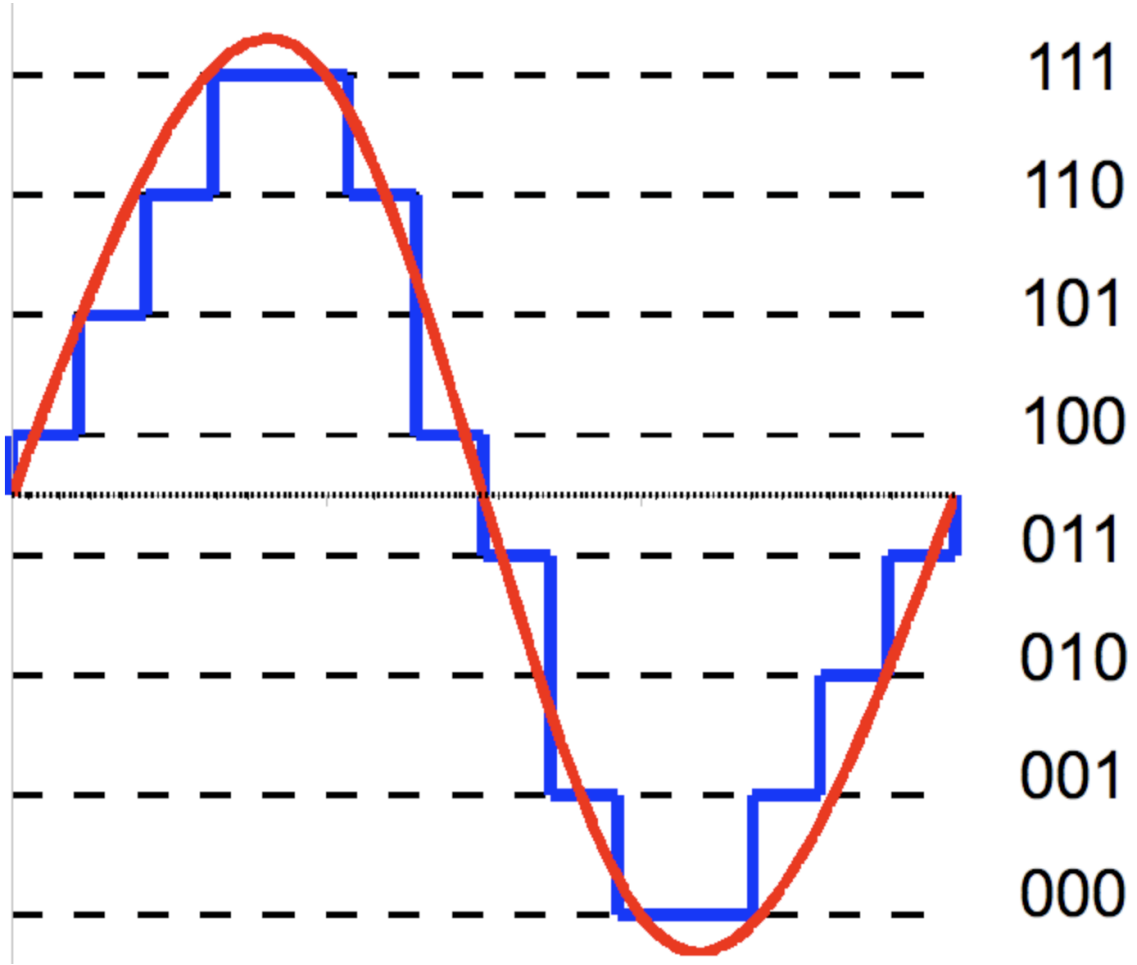


Figure from https://en.wikipedia.org/wiki/Quantization_(signal_processing)

Items That Can Be Quantized

- All memory elements
  - Feature maps
  - Filter coefficients and other parameters
  - Gradients and associated training terms
    - Possible useful for training
    - But also make training more difficult and it's already a hassle
    - So skip for now and focus on quantizing memory used in testing
    - Note that will still cover quantized training
    - Quantized values will be in the forward path though

Why is high precision not always needed in xNNs?

- A linear layer is ~ doing template matching or drawing boundaries
- A loss of precision implies an imperfect template or boundary
- If template matches or classes are sufficiently separated then a bit of noise can be ok
- The question is how much is tolerable
- For different places in the network? Can network modifications be made to make it more tolerable?

- Feature maps and filter coefficients can use different (but compatible) quantization choices
  - Ex: 4 bit signed fixed point filter coefficients and 8 bit unsigned fixed point feature maps

- Need to understand the implications of the reduction in different values that a memory element can take
  - Appropriately balance reduction in range and precision
  - Goal is to maximize efficiency gain and minimize accuracy loss

Fx Pt Quantized CNN Style 2D Convolution

- Start from what can be calculated with fixed point hardware
  - $\mathbf{Y}_q = \text{clip}(\text{round}(s_c\,(\mathbf{H}_q\,\mathbf{X}_q + \mathbf{V}_q)))$
  - $\mathbf{Y}_q$ is the fixed point quantized 2D matrix created via re arranging the 3D tensor of output feature maps
  - $\mathbf{H}_q$ is the fixed point quantized 2D matrix created via re arranging the 4D tensor of filter coefficients
  - $\mathbf{X}_q$ is the fixed point quantized 2D matrix created via a Toeplitz style arrangement of the 3D tensor of input feature maps
  - $\mathbf{V}_q$ is the fixed point quantized 2D bias matrix created via an outer product of a bias vector and 1s row vector

- Compute scale s_c
  - s_c is a scale selected to constrain the output fixed point range to the target number of bits
  - Can select a static s_c based on training data
  - Can select a dynamic s_c based on monitoring accumulator values to maximize dynamic range individually for each input (though this requires some downstream adjustments where additions occur)

- Note
  - $\mathbf{H}_q = \text{clip}(\text{round}(\mathbf{H} / s_H)) \approx \mathbf{H} / s_H$ — static, typically selected to maximize range
  - $\mathbf{X}_q = \text{clip}(\text{round}(\mathbf{X} / s_X)) \approx \mathbf{X} / s_X$ — static or dynamic, from the previous layer
  - $\mathbf{V}_q = \text{clip}(\text{round}(\mathbf{V} / s_V)) \approx \mathbf{V} / (s_H\, s_X)$ — static or dynamic, dependent on the filter and input scale; $s_V = s_H\, s_X$ to align bias (additive) scale with combined filter and input (multiplicative) scale

Fx Pt Quantized CNN Style 2D Convolution

- Substituting in and ignoring clipping and rounding
  - $\mathbf{Y}_q \approx s_c ((\mathbf{H} / s_H)(\mathbf{X} / s_X) + (\mathbf{V} / (s_H s_X)))$
    $\approx (s_c / (s_H s_X)) (\mathbf{H}\,\mathbf{X} + \mathbf{V})$

- Let $s_Y = (s_H s_X) / s_c$ and $\mathbf{Y} = s_Y \mathbf{Y}_q$ then
  - $s_Y \mathbf{Y}_q \approx \mathbf{H}\,\mathbf{X} + \mathbf{V}$
  - $\mathbf{Y} \approx \mathbf{H}\,\mathbf{X} + \mathbf{V}$

- This implies we can approximate floating point CNN style 2D convolution $\mathbf{Y} = \mathbf{H}\,\mathbf{X} + \mathbf{V}$ with fixed point CNN style 2D convolution $\mathbf{Y}_q = \text{clip}(\text{round}(s_c\,(\mathbf{H}_q\,\mathbf{X}_q + \mathbf{V}_q)))$ and the following constraints
  - Bias scale $s_V = s_H s_X$ is dependent on the filter and input scale; as such, the bias typically uses 2x – 4x the number of bits vs multiplicative parameters
  - Output feature map scale $s_Y = (s_H s_X) / s_c$ is a function of the filter, input and compute scales; for convenience of implementation the compute scale s_c may have some constraints (e.g., only powers of 2)
  - Note the coupling of scales from 1 layer to the next
  - Note that typically do accumulation at 4x input scale before compute scale

Fx Pt Quantized Pooling

- Not a big deal
  - For max pooling: the set of possible output values come from the set of input values
  - For avg pooling: the set of possible output values is bound by the range of input values

31	21	33	34	5	2	15
10	29	32	6	27	16	13
7	4	28	20	24	30	26
25	18	14	35	22	1	3
17	23	12	8	19	9	11

Max pool ↓ 3x3 / 2

33	34	30
28	35	30

32	36	68	56	72	48
8	40	64	84	80	12
28	96	92	76	16	4
52	88	44	20	60	24

Avg pool ↓ 2x2 / 2

29	68	53
66	58	26

Fx Pt Quantized N Input 1 Output Operations

With additive (definitely) or concatenative (probably) operations

- A bit of a hassle
  - Need to align scales of inputs
  - A simple way to do this is to us the largest scale as a common scale to bring all other scales to, then perform the operation
- Examples
  - Element wise addition (as in ResNet)
  - Concatenation (as in GoogLeNet / Inception, DenseNet, ...)
  - After grouped convolutions (if different scales are used for different groups)

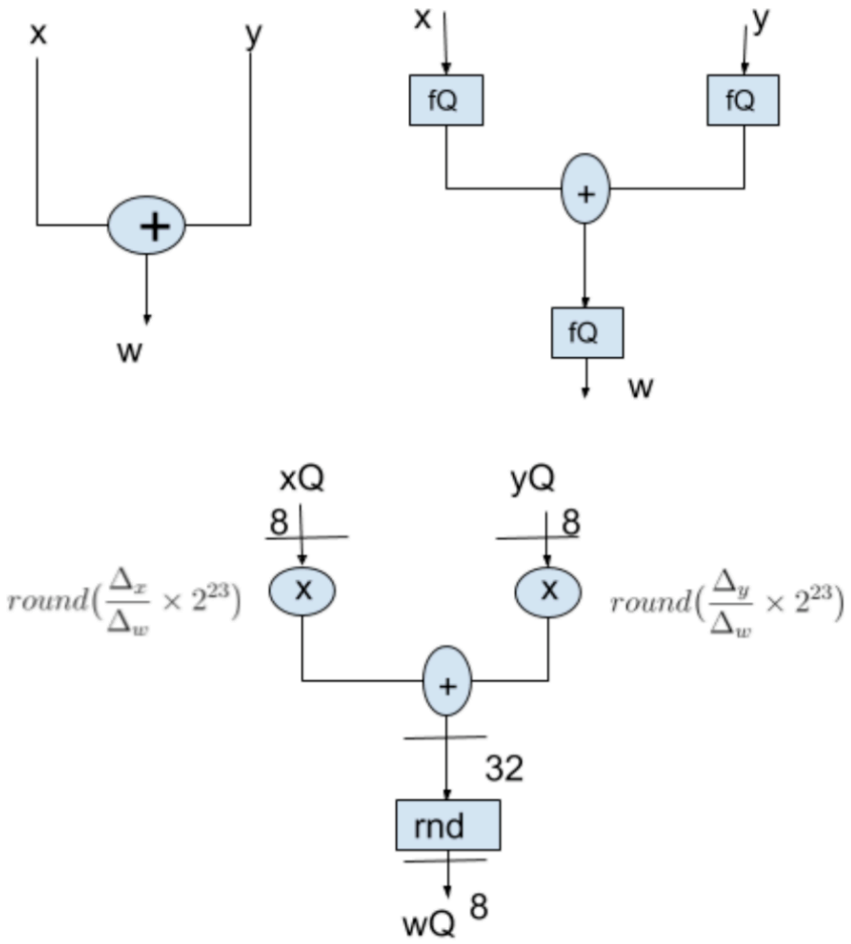
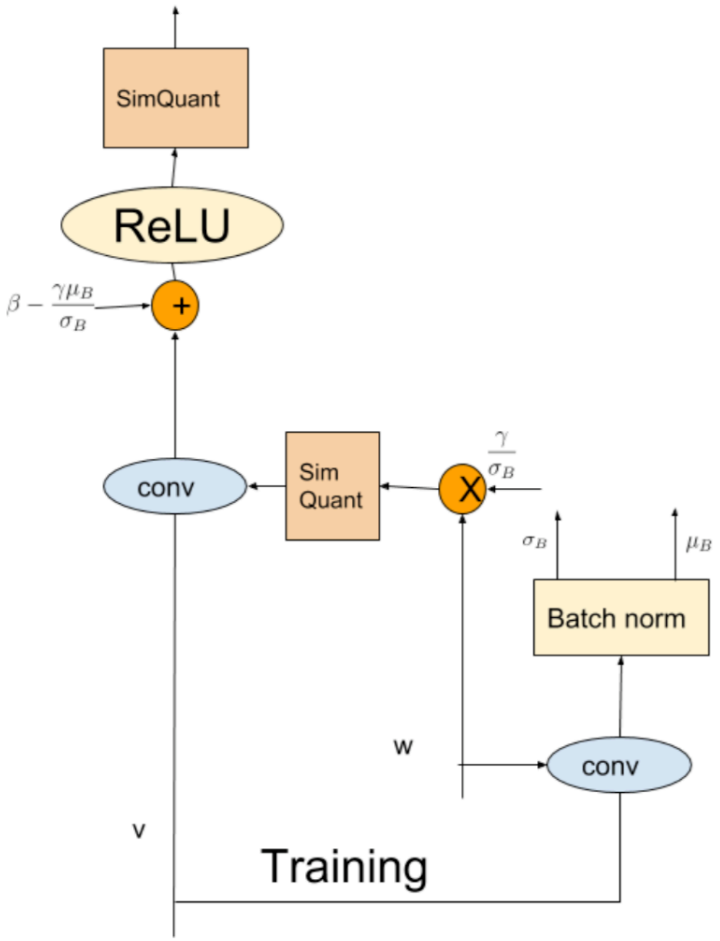


Figure from https://arxiv.org/abs/1806.08342

Fx Pt Quantized Batch Norm

- A bit of a hassle
  - Because it's data dependent
  - But on the upside it's only needed during training
  - Typically do via 2 passes through convolution



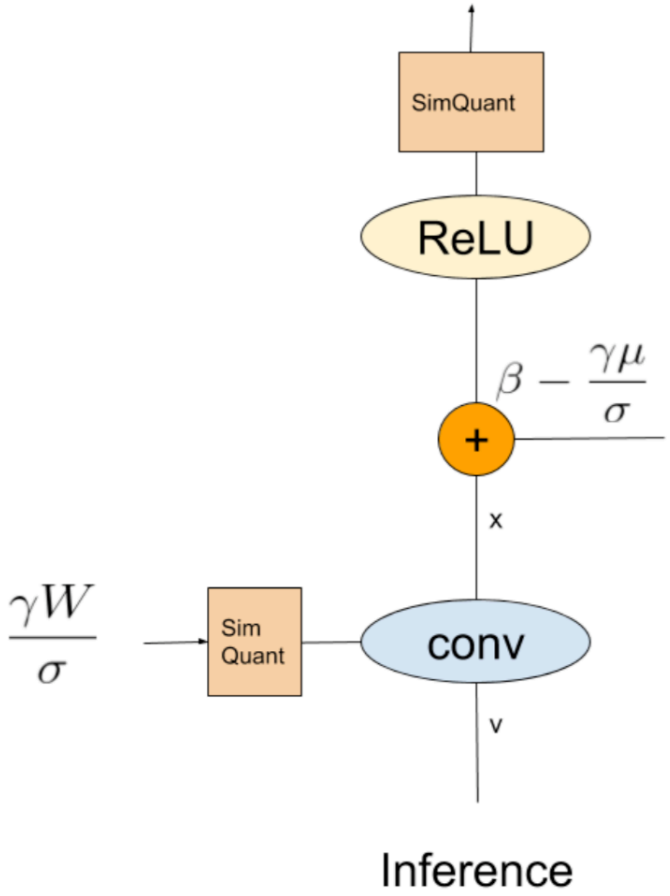


Figure from https://arxiv.org/abs/1806.08342

Range Options

- Previous examples selected scales such that we don't clip but that's not the only option
  - Clipping becomes more useful for static compute scale selection

- No clipping
  - Signed
    - -MaxAbs to MaxAbs (parameters and feature maps in general)
    - Min to Max (implies extra operations, get at most 1 extra bit)
  - Unsigned
    - 0 to Max (feature maps after ReLU)
    - Min to Max (implies extra operations)

- Clipping
  - For traditional signal processing set clip value to maximize SNR but it's different for CNNs
  - Data space is usually smooth but feature space is spikey with many small values
  - SNR as an evaluation metric is the wrong criteria by itself
  - Really care about information and final accuracy

Simulating Fx Pt In Floating Pt Hardware

- A common trick is to introduce quantize – un quantize blocks into the graph
  - Quantize $\mathbf{X}_q = \text{round}(\mathbf{X} / s_X)$ a tensor to introduce loss of information
  - Un quantize the tensor $\mathbf{X} \approx s_X \mathbf{X}_q$ to bring back to original range
  - Perform operation
  - Take care to properly handle N input 1 output and batch norm operations

Quantizing A Trained Network

- Example: quantizing a trained network with 32 bit floating point filter coefficients to 8 bit fixed point multiplicative filter coefficients and 32 bit additive filter coefficients for deployment

- Permutations
  - With or without dynamically selecting compute scale s_c
  - Note that dynamic scale selection requires extra min / max tracking at the accumulator precision

- Comments
  - Can typically tolerate some clipping in the beginning but not the end (when trying to find the max)
  - Highly grouped networks are a challenge
  - Very low precisions are a challenge; sometimes alter network structure to implicitly increase precision

Example strategy for 32 bit float to 8 bit fixed

- Run a large number of inputs through the network and compute the average of the min and max value for each feature map at every point in the network
- Consider multiplying these scales by a ramp that starts at 1.0 for the first layer and ends at 2.0 for the last layer as the net is more tolerant of clipping in the beginning than end
- Select the multiplicative filter coefficient scales for all layers using the maxAbsX approach for a symmetric range about 0
- Select the scale to quantize the input using maxX or maxAbsX
- Starting at the first layer select the scale of additive params based on the input and multiplicative parameters then select the scale of the compute to match the target output range
- Iteratively walk forward from the tail of the network to the head and repeat the selection of the scale of the additive parameters and the compute (note that this captures the linking of scales from 1 layer to the next)

Quantized Network Training / Retraining

- Permutations
  - From scratch
  - From a pre trained floating point model

- Notes
  - Likely accumulate gradients at a higher precision
  - May start training at a higher number of bits (float or fixed) then reduce the number of bits as ranges stabilize

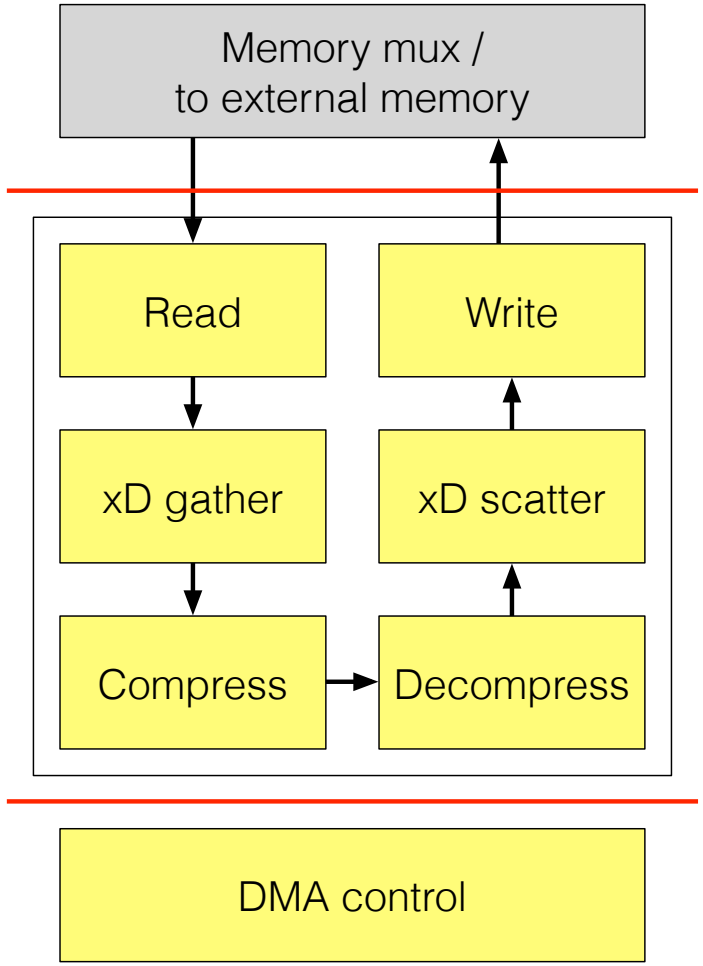
For more details see:

Quantizing deep convolutional networks for efficient inference: A whitepaper
https://arxiv.org/abs/1806.08342

Networks – Compression

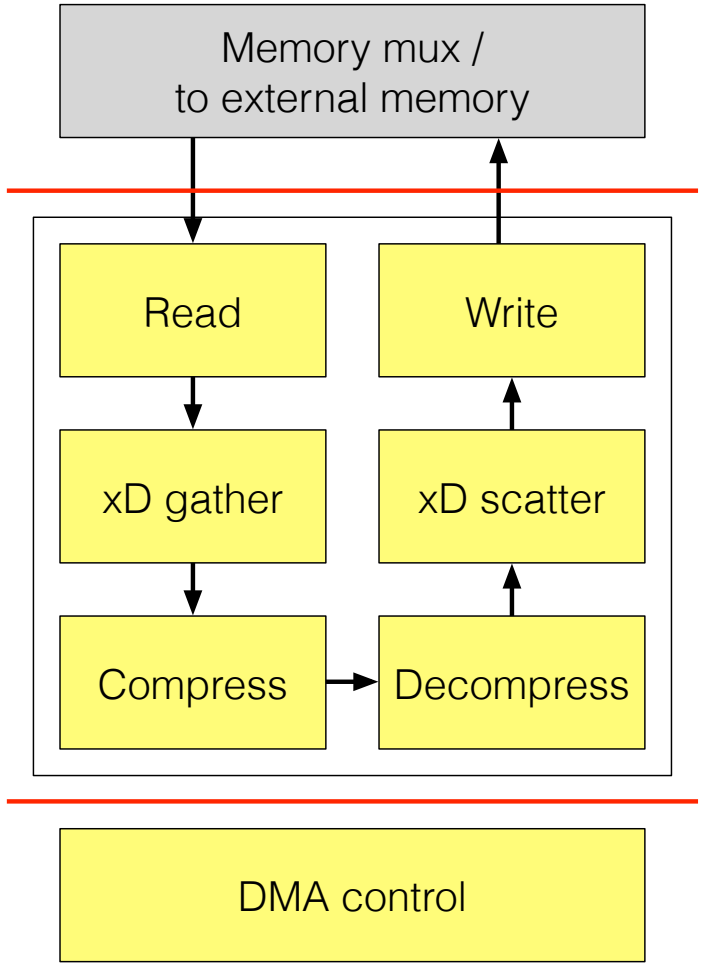
Reduce Data Movement Via Compression

- Strategy
  - Remove redundancy in parameters, feature maps and gradients to minimize the memory requirements and or bandwidth required to move memory
- Lossless or lossy
  - Application dependent tolerance of loss
  - Lossless is currently more common



Reduce Data Movement Via Compression

- Static or dynamic data
  - Static: filter coefficients (after training)
  - Dynamic: filter coefficients (during training)
    feature maps
    gradients
- Per symbol or across symbols
  - Per symbol exploits non uniform pmf
  - Across symbols exploits correlation (or other) structure

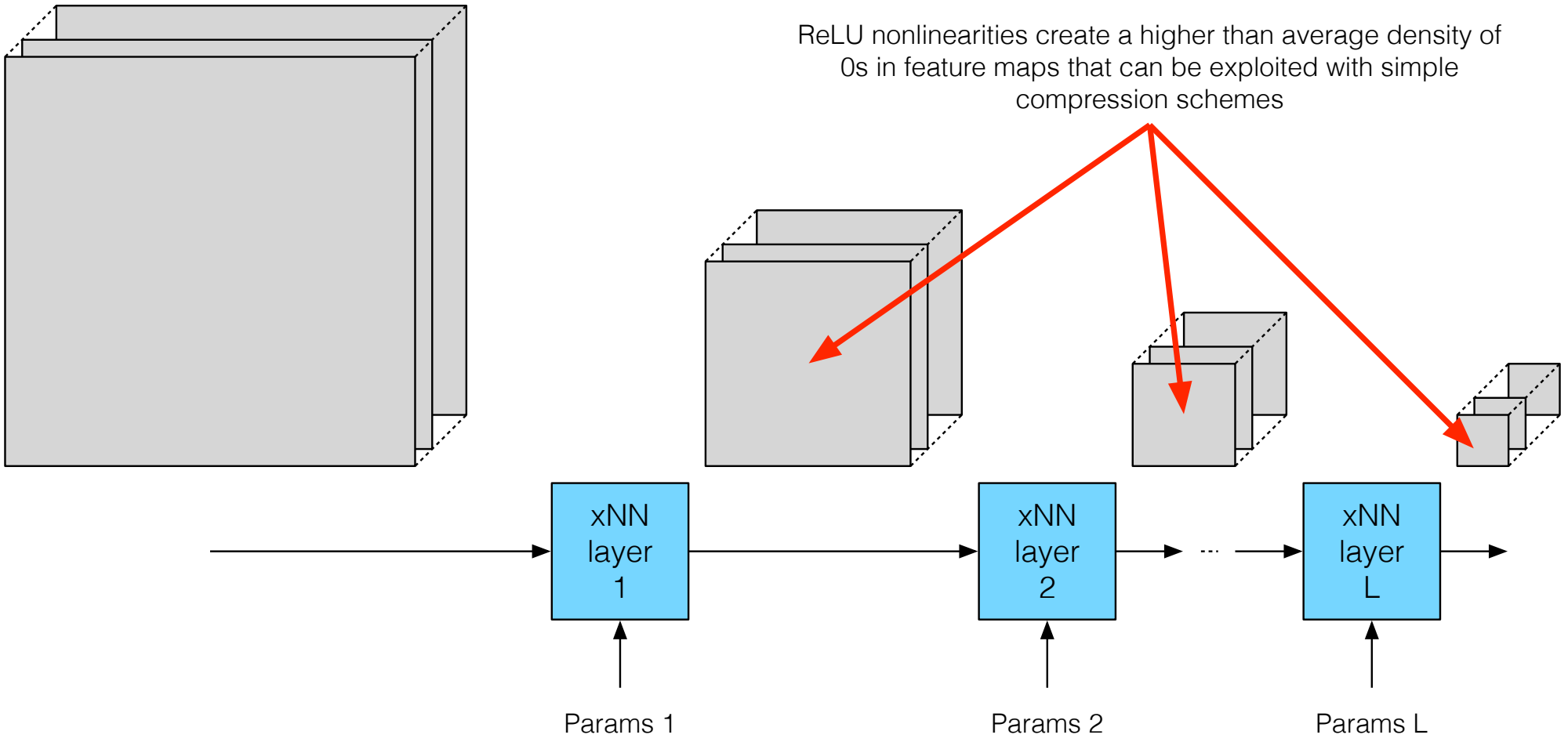


Feature Map Compression

- Exploit redundancy in feature maps
  - Values are dynamic (they change for every input and layer)

- Depending on the nonlinearity and network design relatively effective simple options are possible
  - ReLU leads to a non uniform distribution of 0s in feature maps
  - In the probability / info theory lecture we looked at Huffman coding as a method that can take advantage of non uniform densities of values
  - Can extend Huffman to include groups of 0s as symbols
  - Can add a run length encoding variant to code across symbols

Feature Map Compression



Filter Coefficient Compression

- Exploit redundancy in filter coefficients
  - Static ((typically) known ahead of time for deployments)
  - Note that this implicitly implies that maybe we should be thinking about network simplification
- Sometimes redundancy is even encouraged during training
  - Training for sparsity (re: L1 weight decay)
  - Also a potential network simplification depending on hardware capabilities

Gradient Compression

- Gradients compression comes into play during distributed training
  - Dynamic (changes for every input and every layer)
- Deep gradient compression: reducing the communication bandwidth for distributed training
  - https://arxiv.org/abs/1712.01887

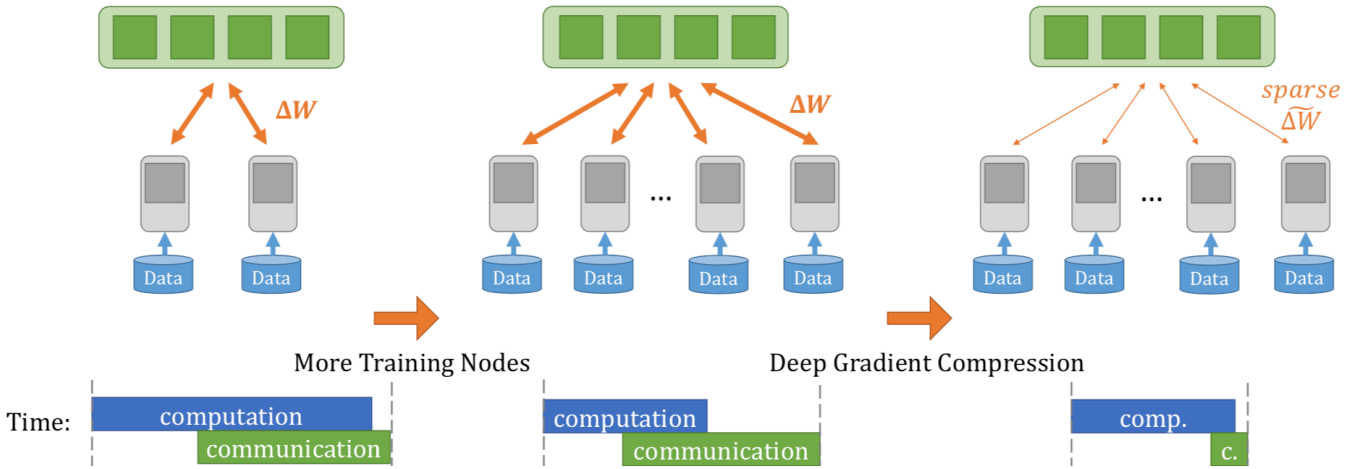
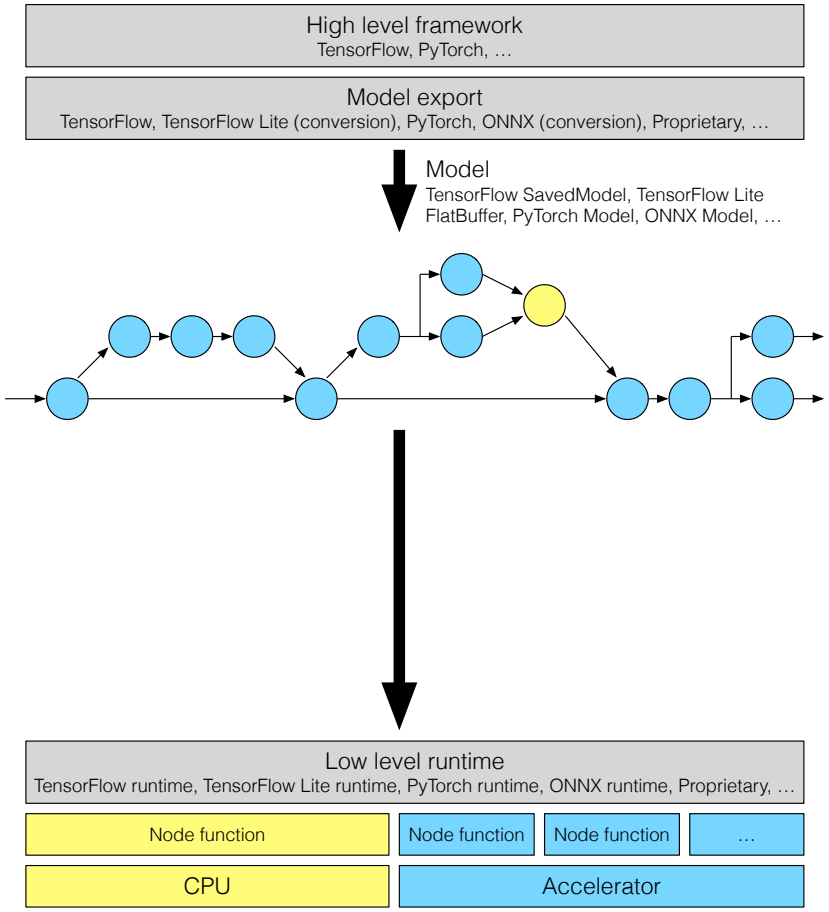


Figure from https://arxiv.org/abs/1712.01887

Software

Where xNN Software Is Going

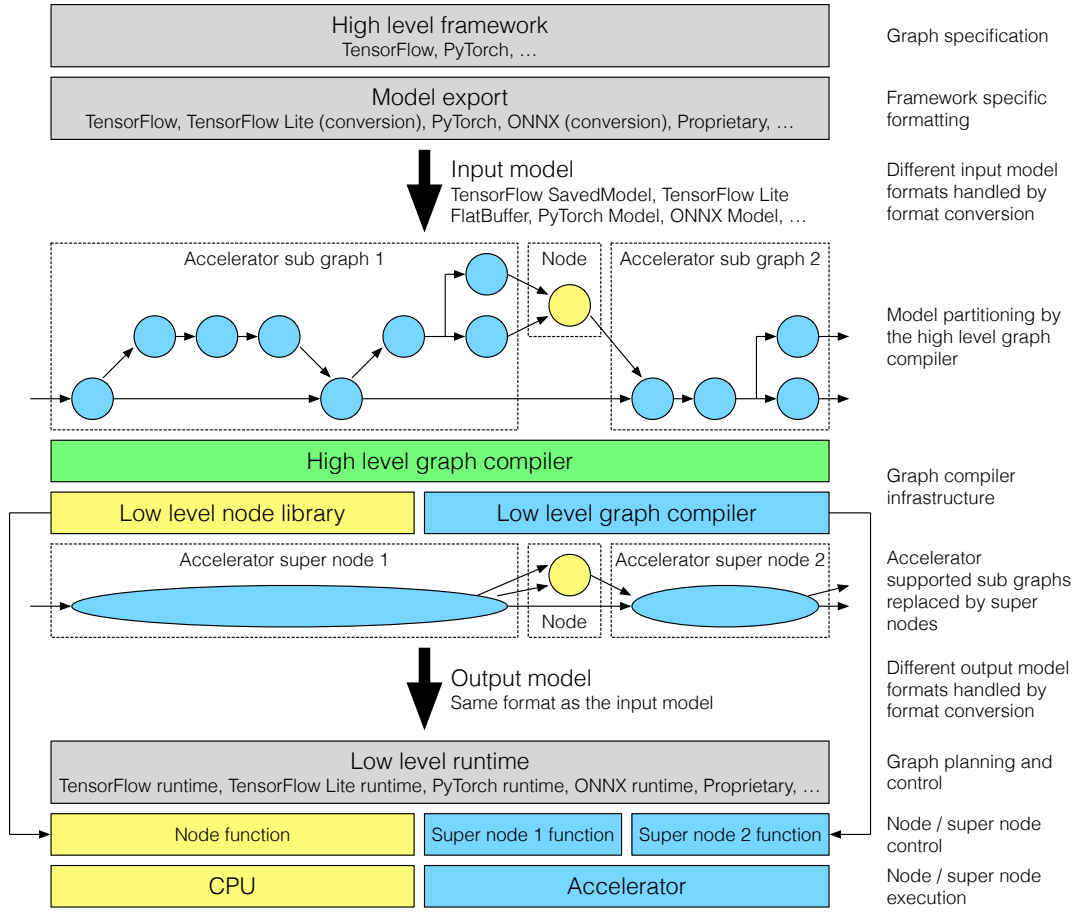


Replace accelerator supported sub graphs with super nodes

Nodes not supported by the accelerator fall back to default CPU implementations

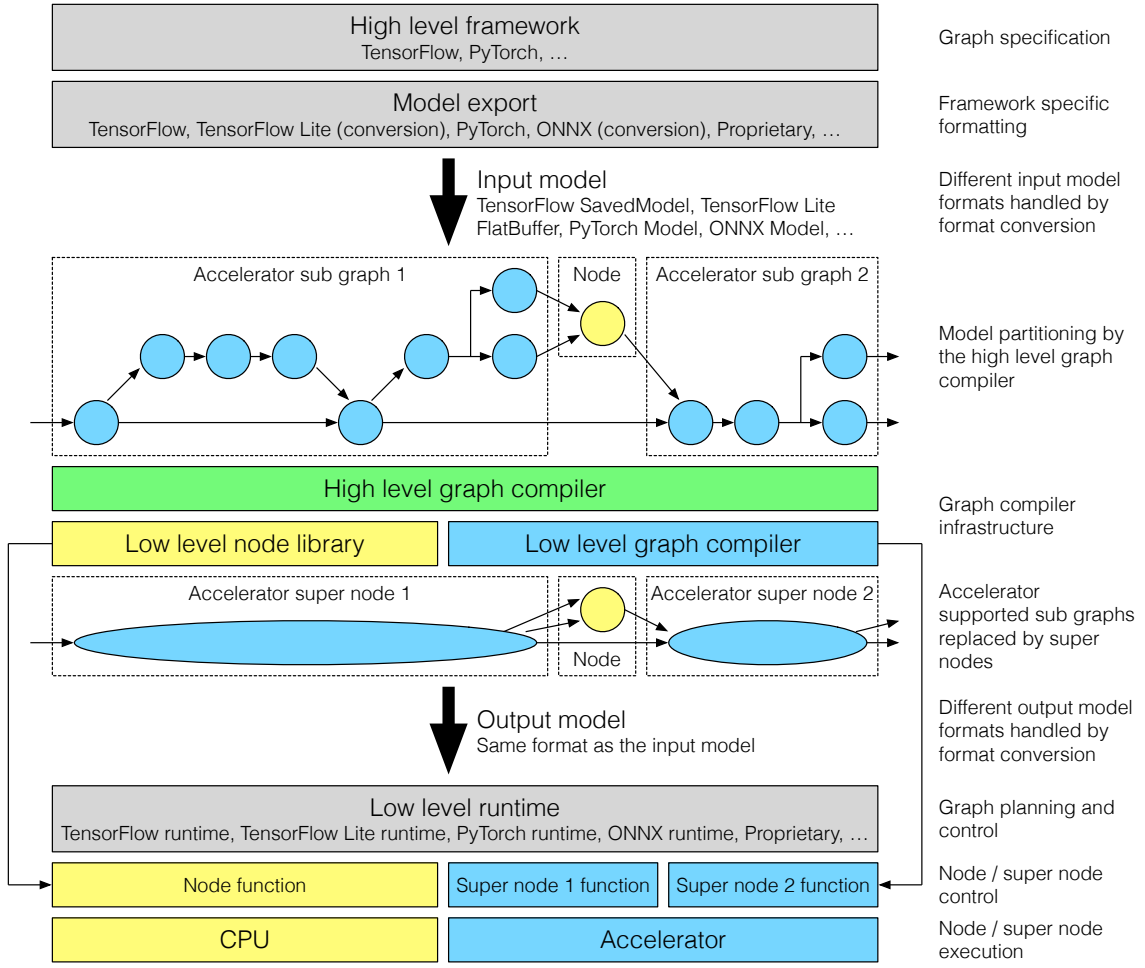
The accelerator has the freedom to choose the optimal implementation for each sub graph

This allows for specialized hardware implementations that minimize communication and have well matched computation

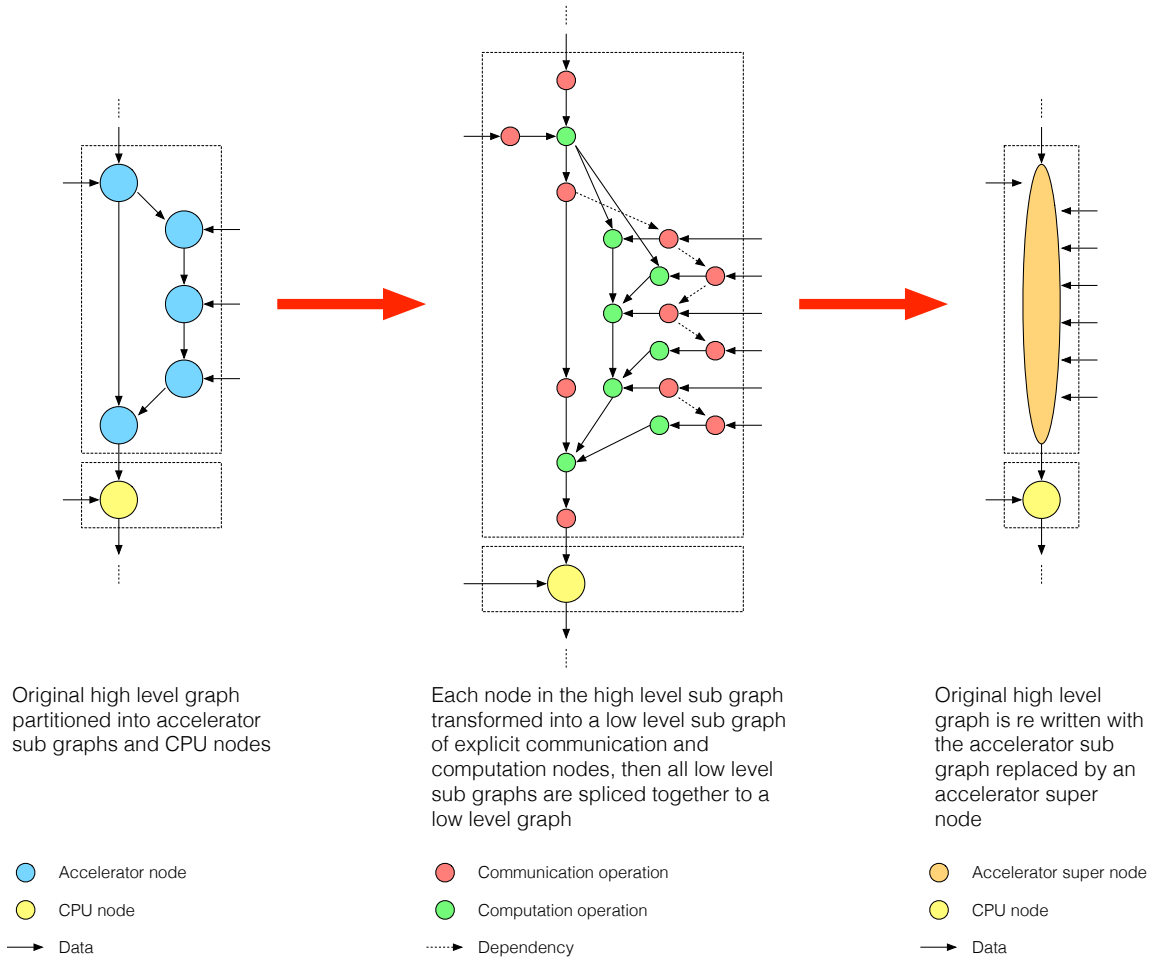


Software Outline

- Graph specification
  - High level framework
  - High level model export
- Graph compilation
  - High level graph compiler
  - Low level graph compiler
  - Low level node library
- Graph execution
  - Low level runtime

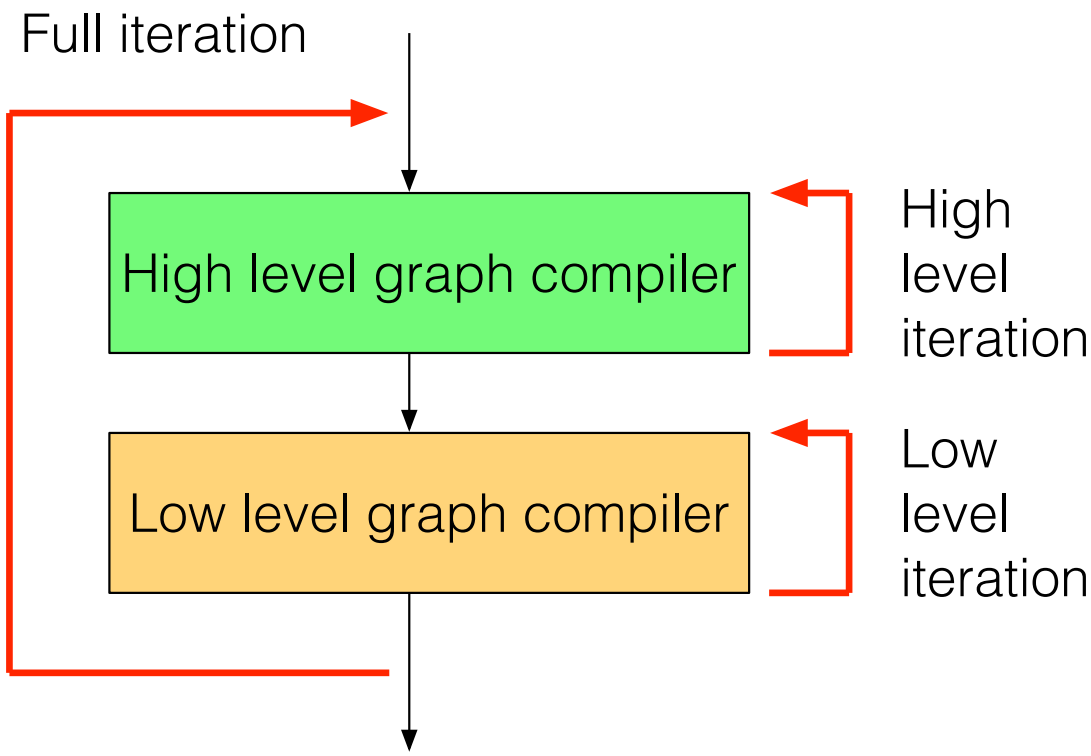


In Pictures


Original high level graph partitioned into accelerator sub graphs and CPU nodes
Each node in the high level sub graph transformed into a low level sub graph of explicit communication and computation nodes, then all low level sub graphs are spliced together to a low level graph
Original high level graph is re written with the accelerator sub graph replaced by an accelerator super node
Accelerator node
CPU node
Data
Communication operation
Computation operation
Dependency
Accelerator super node
CPU node
Data

Principles

- Abstraction
  - High level graphs are hardware agnostic
  - Low level graphs are hardware specific
- Exploit
  - Compile time vs run time information
    - Memory layout
    - Deterministic control
    - Communication selection
    - Computation selection
  - Larger pieces of throughput optimized work vs smaller pieces of latency optimized work
- Optimization is (really really) non convex
  - So typically need iterative optimization across all layers
- Note the frequent use of the word graph



Software – Graph Specification

High Level Framework

- Graph specification
  - The network is specified as a high level hardware agnostic directed acyclic graph
  - For training a data source, data path, error and update method are specified
    - The tool adds nodes to create the error gradient path and weight updates
  - For testing a data source, data path and output are specified
  - See the backup – software slides for more details on this, specifically with respect to the details of various commercial packages

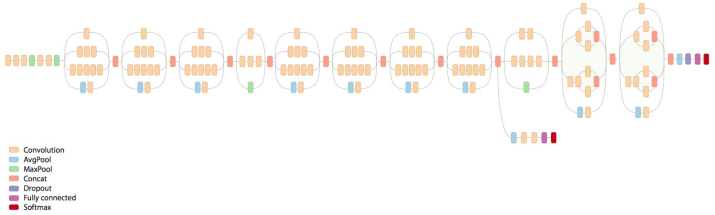
- Different execution methods
  - Graph execution: initial software had separate steps for graph specification, graph compilation and graph execution; this is optimal from a performance perspective but counterintuitive to typical development expectations in an interpreted language
  - Eager execution: later software packages followed the expectations of an interpreted language and executed code as written; this is better from a development perspective but sub optimal from a performance perspective
  - Recent frameworks are moving towards supporting both styles

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```



Figures from https://www.tensorflow.org/tutorials/ and https://ai.googleblog.com/2016/08/improving-inception-and-image.html

High Level Framework

- On the topic of memory
  - The locations of tensors can be static or dynamic
  - The contents of tensors can be static or dynamic
  - The management of tensors can be internal or external

- This information needs to be either available to or infer ably by the graph compiler for optimal performance

- Examples
  - Different input images (dynamic contents) during testing can be in different places in DRAM (dynamic location) determined by a host application (external management)
  - Trained weights (static contents) during testing can be in the same place in DRAM (static location) determined by the graph compiler (internal management) within an allocated memory pool

High Level Model Export

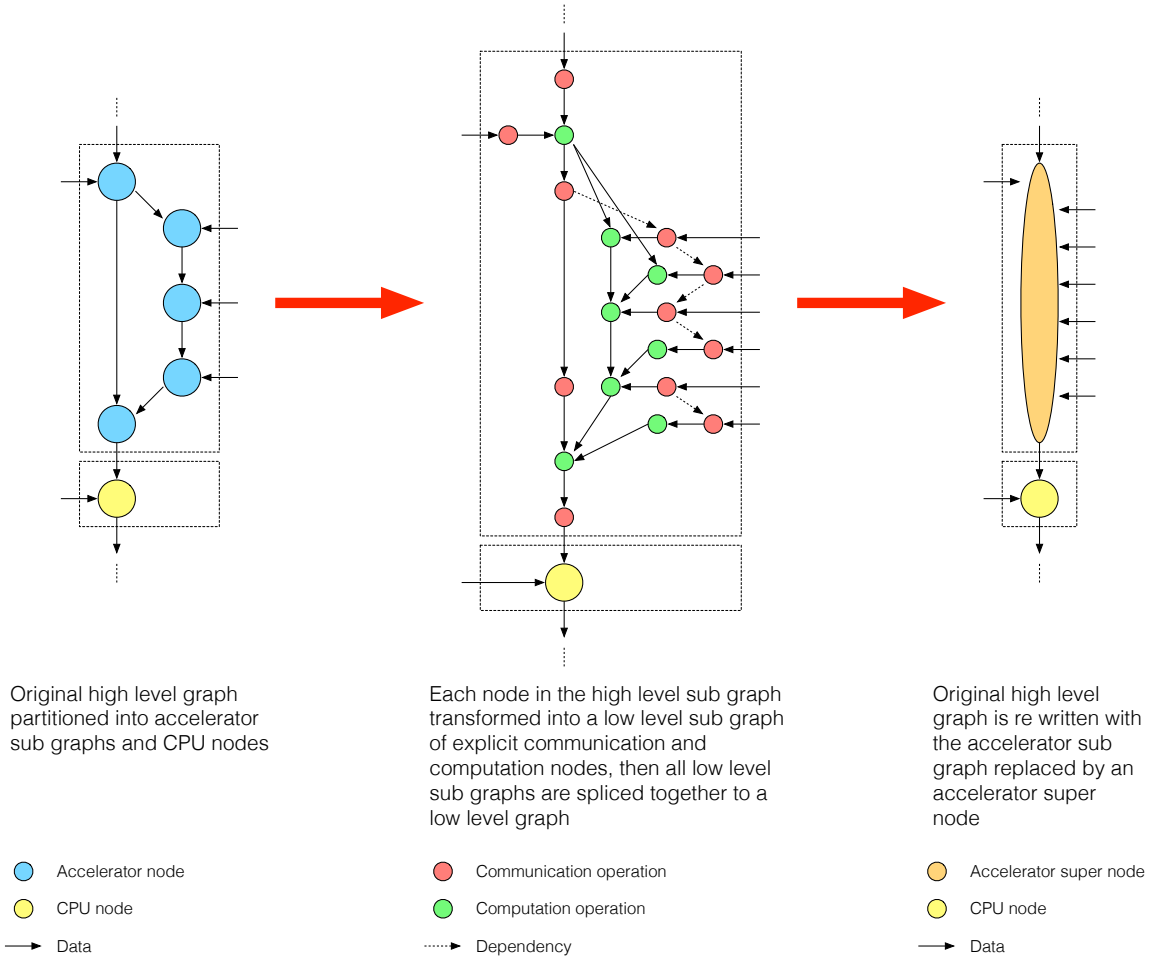
- Frameworks work with models in different formats
  - During training this is typically an in memory representation
  - After training this is typically thought of as a static graph
  - Use the word conversion when it's not the native format

- Sometimes some graph optimization is done during the export process

Software – Graph Compilation

Flow

- High level graph compiler
  - Format conversion
  - Domain agnostic and domain specific optimization
  - Sub graph partitioning
  - Graph re writing (after below steps)
- `For each high level sub graph targeted for the accelerator`
  - Low level graph compiler
    - Memory planning
    - For each high level node
      - Low level sub graph code generation
    - Splice together all low level sub graphs
- `For all remaining nodes`
  - Low level node library

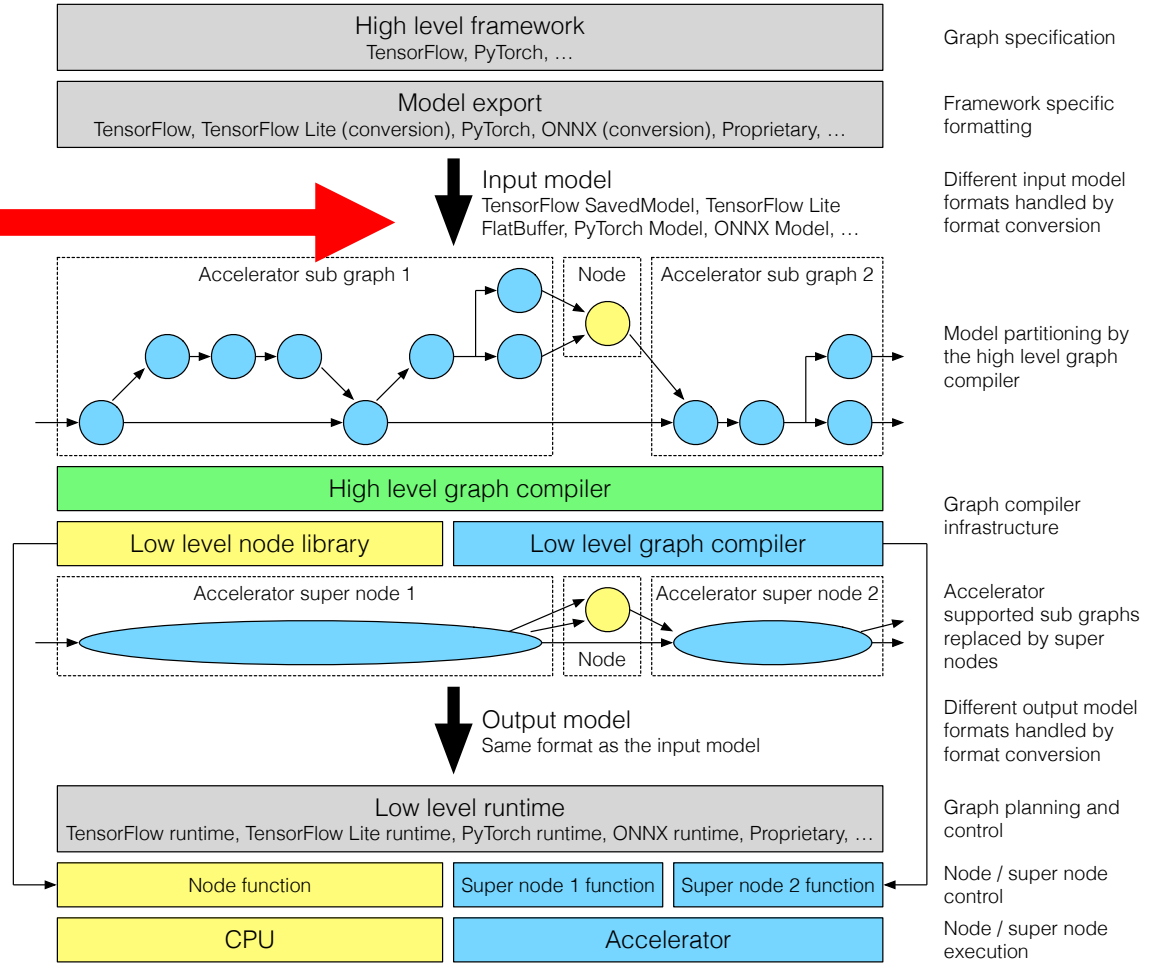


High Level Graph Compiler

Format conversion

- If you're building custom hardware and want to support many high level frameworks and low level runtimes, it's most common to
  - **Bring the high level graph to a common format**
  - Extract out supported sub graphs
  - Determine optimal implementations for supported sub graphs
  - Re write the high level graph with supported sub graphs replaced by super nodes

- There's a lot of interest in defining common high (and low) level graph formats
  - ONNX model
  - MLIR
  - …



High Level Graph Compiler

Optimization

- Domain agnostic optimization
  - Remove unneeded edges and nodes
  - Constant folding and constant propagation
- Domain specific optimization
  - Hardware agnostic
    - Remove dropout and scale associated layer weights
    - Absorb batch norm into conv and bias
  - Hardware aware
    - Transform data layouts
    - Node fusion, tiling and grouping
    - Post training quantization

High Level Graph Compiler

Sub graph partitioning

- Assign different sub graphs to different hardware back ends

- Accelerators typically use some degree of specialization to achieve high levels of performance at the expense of less flexibility
  - Identify sub graphs with all nodes fully supported by the target accelerator
  - Assign that sub graph to the accelerator
  - Repeat for all sub graphs

- Common to include a CPU back end that can be used as a fall back to maintain full library compatibility
  - All nodes not assigned to accelerator sub graphs are assigned to the CPU

Low Level Graph Compiler

For each high level sub graph – memory planning

- Given
  - High level sub graph implying tensor lifetimes
  - The location type and content type for all externally managed tensors
  - Internal and external memory pools
- Determine
  - An internal or external memory location for every tensor (use placeholder addresses for all externally managed tensors)
  - Internal and external memory pool availability for every high level node
- Notes
  - Memory planning is very important for high performance systems with on device memory and also low performance systems with limited amounts of resources
  - Any tensor in external memory that's an input to a compute node will need to be moved from external memory to internal memory by a communication node before being consumed by the compute node
  - Any tensor in external memory that's an output from a compute node will need to be moved from internal memory to external memory by a communication node after being generated by the compute node

Low Level Graph Compiler

For each high level sub graph – for each high level node in the high level sub graph – low level sub graph code generation

- Decompose each high level node into low level control instructions and a low level sub graph of communication and computation nodes with associated instructions
  - Memory constraints come from the memory planner
  - Low level control instructions move instructions in internal memory to the control, computation and communication instruction queues and start execution of instructions in the control, computation and communication instruction queues
  - Low level communication nodes move instructions and data between external memory and internal memory through the communication accelerator; typical communication accelerator specific instruction format is configure then execute for many cycles for maximum compactness and efficiency
  - Low level computation nodes move input data from internal memory to the computation accelerators, generate output data and move move output data from the computation accelerators to internal memory ; typical compute specific accelerator instruction format is configure then execute for many cycles for maximum compactness and efficiency

- Different low level sub graph decompositions are typically possible for each high level node, select the 1 with the best performance
  - For supported high level nodes have a library of optimal implementations for different parameter combinations, input and output memory locations and available internal and external memory pools
  - TVM style approaches attempt to auto generate these with a combination of generic implementation strategies, hardware intrinsics and ML based auto tuning

Low Level Graph Compiler

For each high level sub graph – splice together all low level sub graphs

- This step splices together all of the low level sub graphs, each corresponding to 1 high level node, into a single low level graph that implements the high level sub graph

- A few additional things are typically done at this point
  - The complete low level graph is re partitioned into initialization and execution low level sub graphs
    - The low level initialization sub graph can be executed 1x before multiple inputs, implying no dynamic dependencies
    - The low level execution sub graph is executed for each new input
  - The low level initialization and execution sub graphs are further re partitioned into smaller low level sub graphs
  - Communication nodes are added to move packets of instructions for all low level nodes in each re partitioned low level subgraph such that instructions are present when nodes are ready to execute

- The result can be thought of as a super node function (from the perspective of the low level runtime)

Low Level Graph Compiler

For each high level sub graph – splice together all low level sub graphs

- A preview of the hardware section
  - Nodes in the low level sub graph will map 1 to 1 to computation and communication primitives in the DSA; instructions for these nodes will run on the computation and communication queues
  - Control instructions will run on the control queue and sequence through the low level nodes

Low Level Node Library

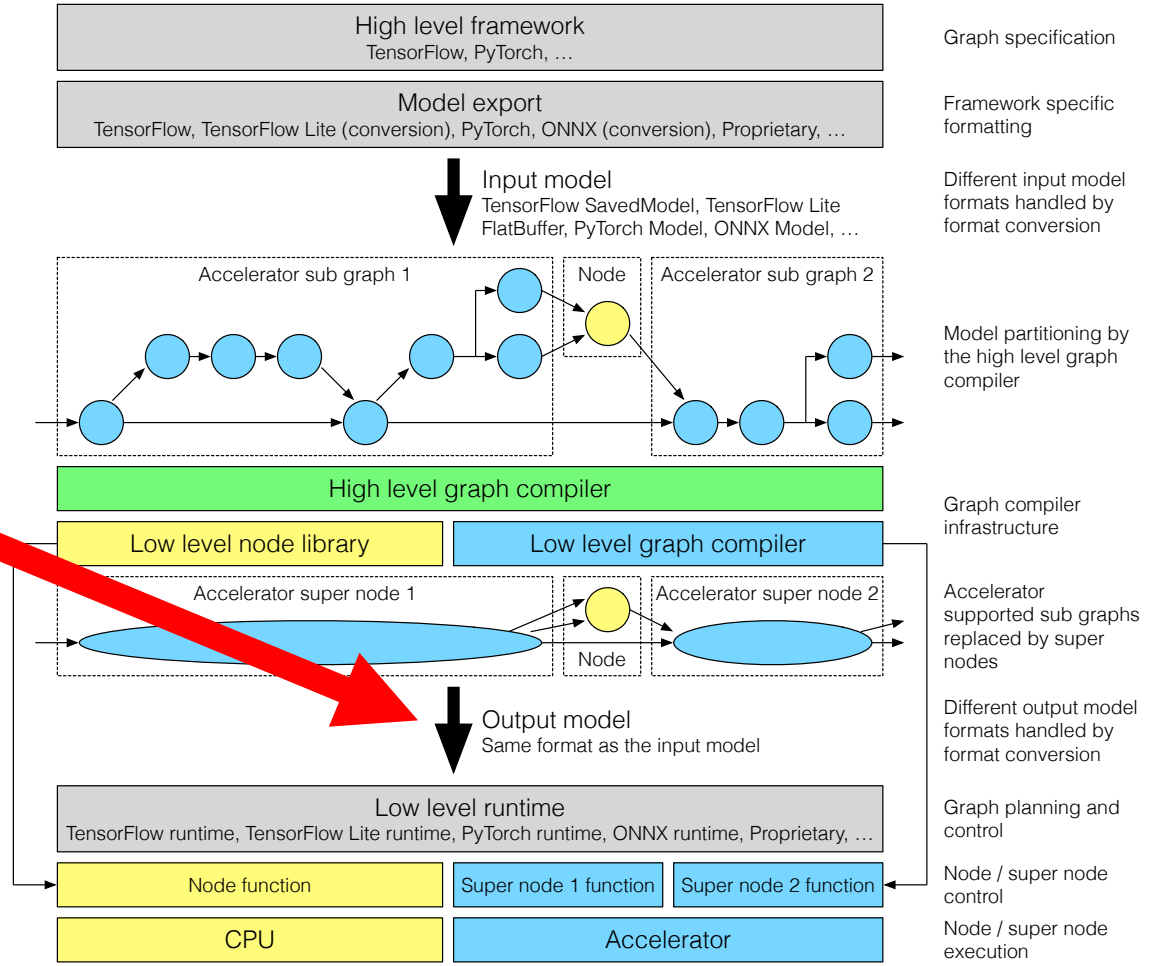
- For CPUs and other backends in an eager style execution mode, it's typical to map each high level graph node to a library function
  - Many nodes (e.g., CNN style 2D convolution) have many potential implementations in a library
  - Typical to select the implementation that has the highest level of performance while meeting the parameter constraints of the specific node realization

High Level Graph Compiler

Graph re writing

- If you're building custom hardware and want to support many high level frameworks and low level runtimes, it's most common to
  - Bring the high level graph to a common format
  - Extract out supported sub graphs
  - Determine optimal implementations for supported sub graphs
  - **Re write the high level graph with supported sub graphs replaced by super nodes**

- There's a lot of interest in defining common high (and low) level graph formats
  - ONNX model
  - MLIR
  - …



Software – Graph Execution

Low Level Runtime

- The low level runtime has initialization and execution phases

- During initialization (1x)
  - Create a schedule of nodes to implement the graph while satisfying dependencies
  - Determine memory locations for all externally managed tensors
  - Map all nodes to appropriate backends and allow backends to perform initialization operations
    - For the accelerator: link static externally managed addresses to the accelerator
    - For the accelerator: run the initialization sub graphs

- During execution (per input)
  - Cycle through the schedule of nodes that implement the graph
  - Call the appropriate backend for each node
    - For the accelerator: link dynamic externally managed addresses to the accelerator
    - For the accelerator: run the execution sub graphs

Hardware

Hardware Outline

- Physics
- System on a chip architecture
- Domain specific architecture
  - Overview
  - Memory
  - Control
  - Communication
  - Computation
- Network architecture

Question

- What is the best hardware design?
  - The brain is an existence proof of what can be computed with 20 W of power and 3 lbs of material
  - But it's not a limit of what can be built
- Coding theorists were lucky and Shannon gave them a limit
- Approximate limits are known for some small pieces of hardware design
  - Comparators
  - DRAM bit cells
  - Individual multipliers
- It's more difficult to answer in the context of a full system with many variables
  - However, we should always have this question in the back of our minds when designing hardware

Hardware – Physics

Moore's Law

- The number of transistors in an integrated circuit doubles every ~ 2 years at a constant cost
  - Previously a little faster
  - Now a little slower
- Note
  - Doesn't say anything about speed
  - Doesn't say anything about power

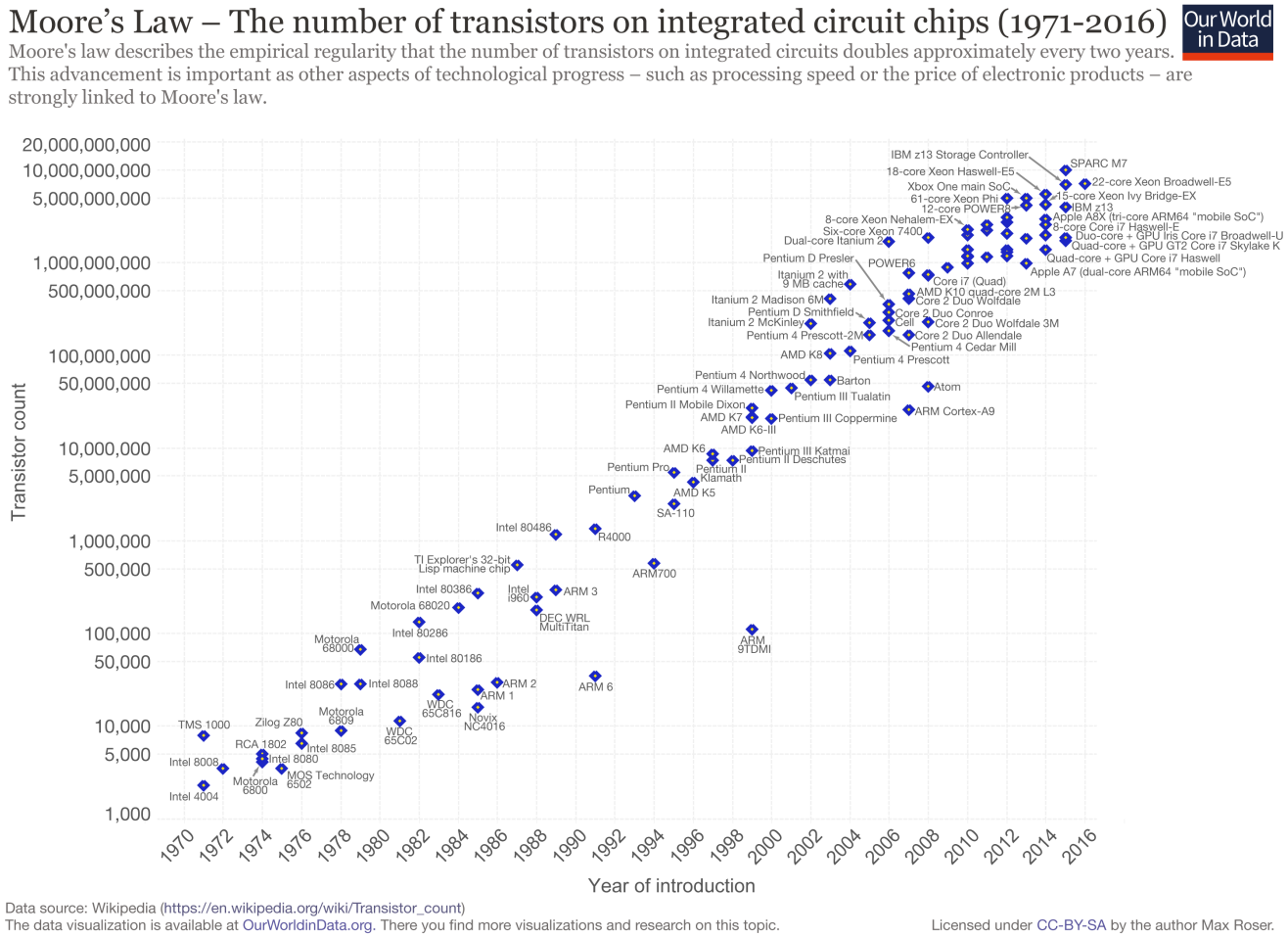


Figure from https://en.wikipedia.org/wiki/Moore%27s_law

Dennard Scaling

- Transistor power density used to be proportional to area but no longer is
  - It was from ~ 1974 – 2006 when energy was dominated by switching frequency (Dennard scaling)
  - But it no longer is (sadness)
  - The problem is that at smaller transistor sizes the threshold voltage and current leakage limits voltage scaling
  - Prior to ~ 2006 improvements in scaling feature sizes and voltage overwhelmed everything else
  - Now need better architecture designs to advance performance

Approximate physics

- L = transistor feature size
- V = voltage
- C = capacitance per transistor ($\propto$ L)
- D = area density ($\propto 1/L^2$)
- E = energy per transistor use ($\propto CV^2$)
- f = frequency ($\propto 1/L$)
- P = power per area ($\propto DEf$)

Approximate process technology

- In 1 generation L is scaled by ~ 0.7
- In 2 generations L is scaled by ~ 0.7*0.7 = 0.49 ≈ 1/2

2 gens with voltage scaling: L' = L/2, V' = V/2 and same area

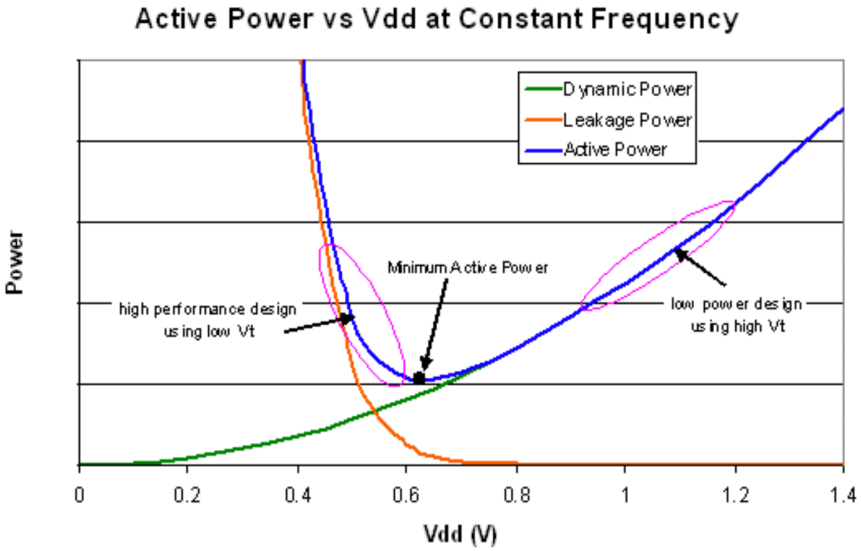
- C' = C/2
- D' = 4D — 4x transistors
- E' = E/8
- f' = 2f — 2x frequency
- P' = P — 1x power

2 gens without voltage scaling: L' = L/2, V' = V and same area

- C' = C/2
- D' = 4D — 4x transistors
- E' = E/2
- f' = 2f — 2x frequency
- P' = 4P — 4x power (**bad**)

Dark Silicon And Dark Memory

- Dark silicon
  - A consequence of the end of Dennard scaling
  - Only a fraction of a device can be active at one time because of increased energy per unit area vs power dissipation limits
  - This gets worse as process geometries continue to shrink
  - The result is that more and more of the device is off at any given time
  - Consequence: design accelerators to be as efficient as possible for key tasks



- Dark memory
  - A consequence of the end of Dennard scaling
  - Only a fraction of DRAM and local device memory can be active at one time because of increased energy per unit area vs power dissipation limits
  - This gets worse as process geometries continue to shrink
  - The result is that more and more of the memory is idle at any given time
  - Consequence: maximize data locality to minimize memory and data movement

Figure from https://www.design-reuse.com/articles/20296/power-management-leakage-control-process-compensation.html

Power Is The Problem

- A list of what consumes power from most to least
  - Physical movement
  - Long distance communication
  - Off device communication
  - On device communication
  - Computation

Example power estimates in 28 nm (from a public presentation from ARM)

16 bit integer MAC	1	pJ
32 bit integer MAC	4	pJ
32 bit float MAC	6	pJ
64 bit float MAC	20	pJ
Read from on chip SRAM	1.5	pJ/B
Read from off chip DRAM	250	pJ/B
Wires 20 mm 50% transitions	7	pJ/B
Chip to chip parallel link	25	pJ/B
Chip to chip serial link	50	pJ/B

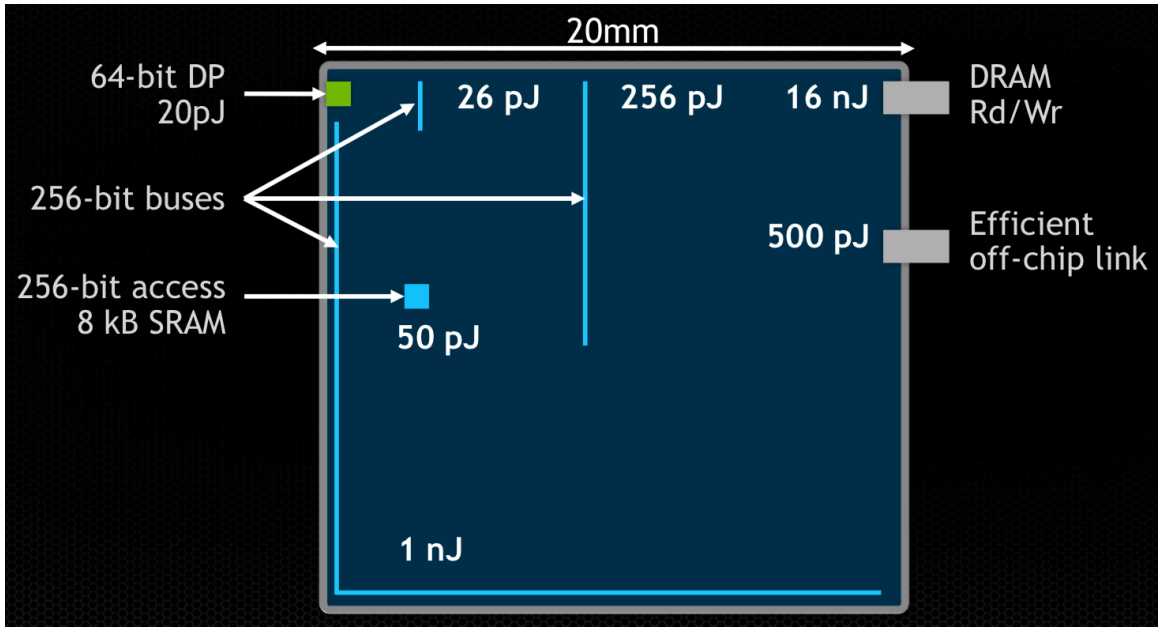
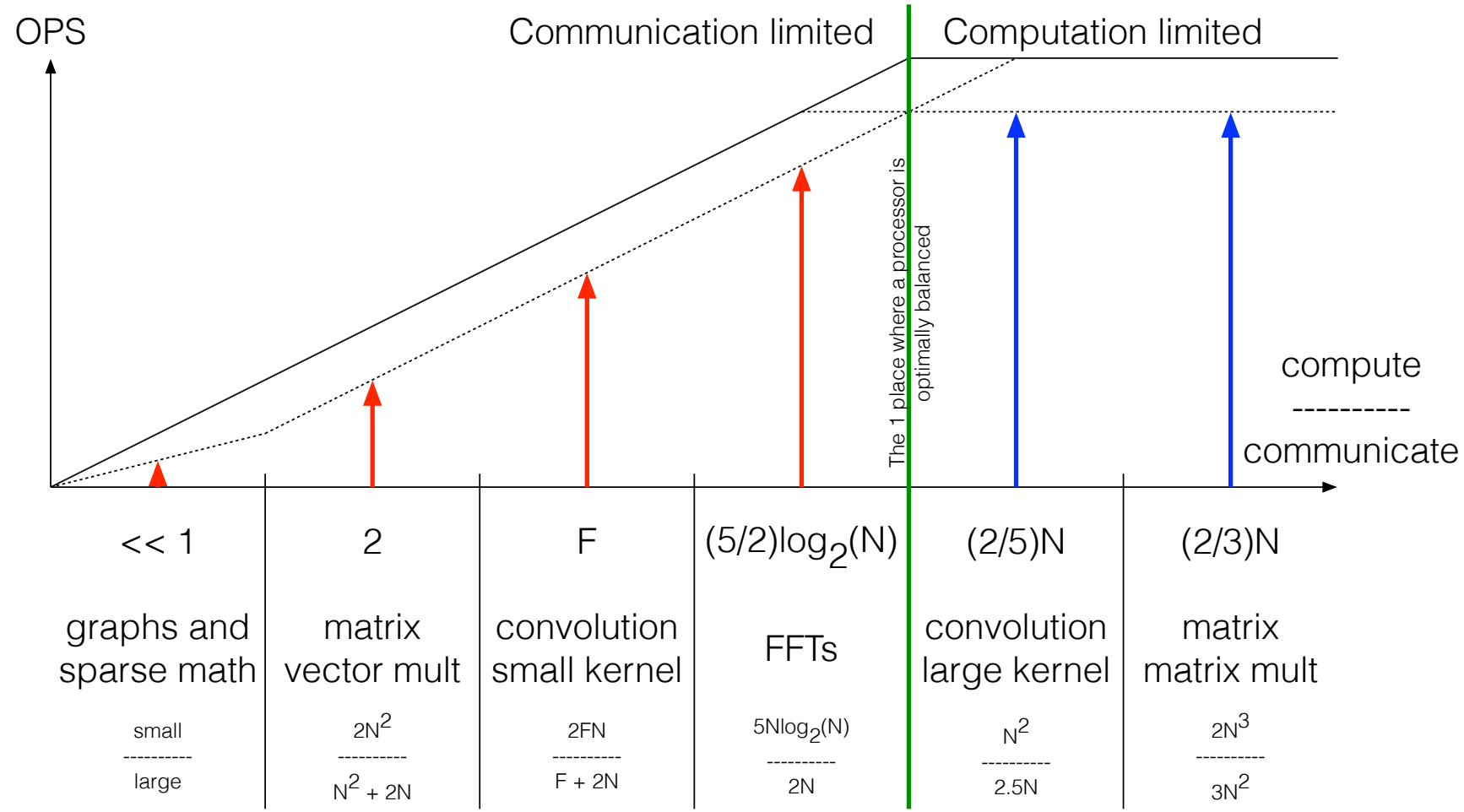


Figure from B. Dally SC415 Nvidia the path to exascale

Roofline Model



Putting 1 And 1 And 1 And 1 Together

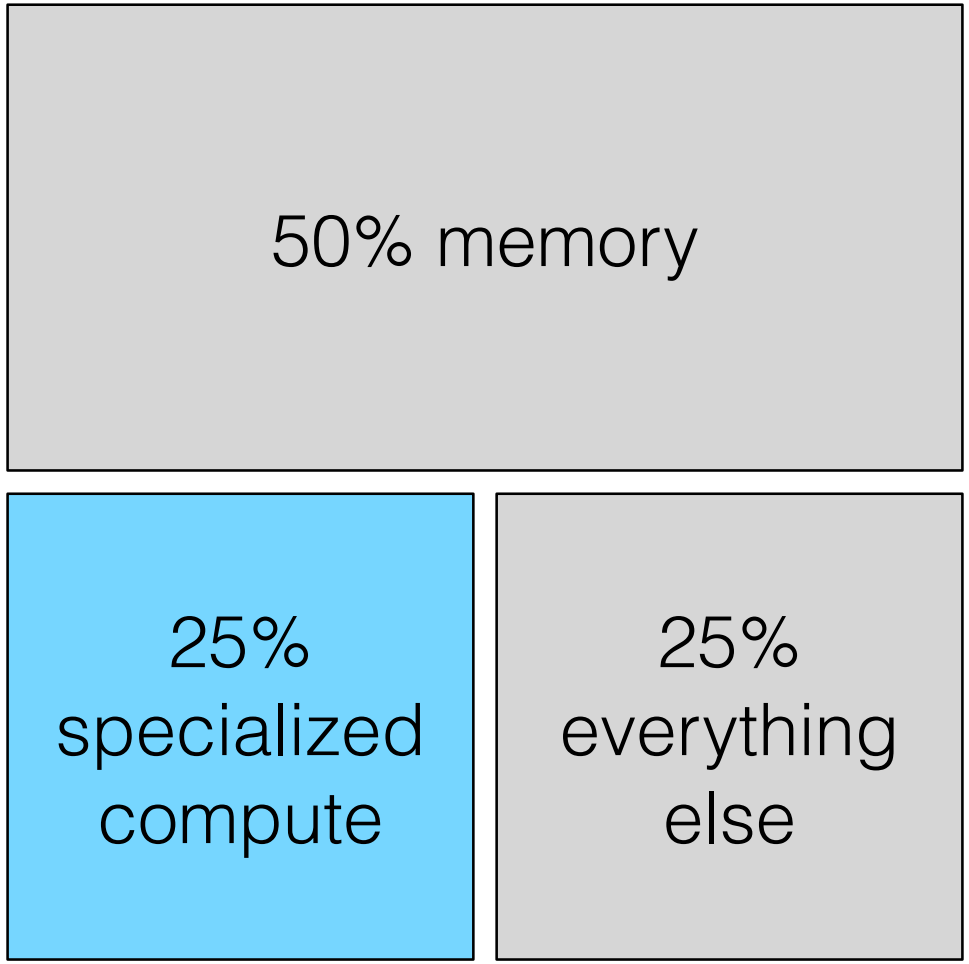
- Power is the problem, only part of the device can be on at a time, moving data takes the most power and compute is limited by data movement

- The implication of this thought chain with respect to how to design optimal hardware
  - Minimize off device data movement
    How: include sufficient on device memory
  - Minimize on device data movement
    How: data locality and accelerator reuse of data
  - Optimize compute
    How: exactly matched to algorithm, parallel to and sized for data movement

Trends

- Big compute device trends ≈
  - 50% memory
    - If off device data movement suddenly became very low power and very high throughput then this number will reduce
  - 25% optimized compute
  - 25% everything else

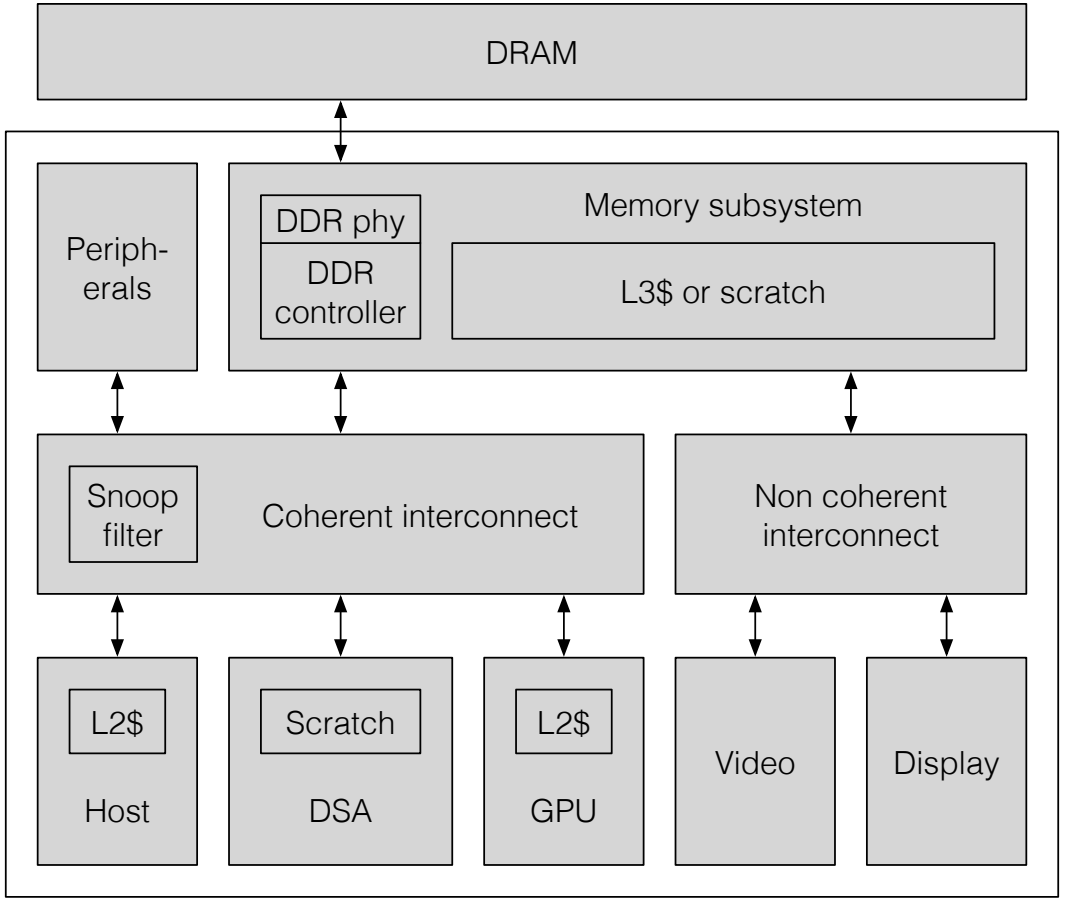
- Want the network designer to design the easiest networks to run as possible
  - But at the end of the day need to run the network
  - So how to support generality to future proof while still providing optimized compute



Hardware – System On A Chip Architecture

A Generic SoC Architecture

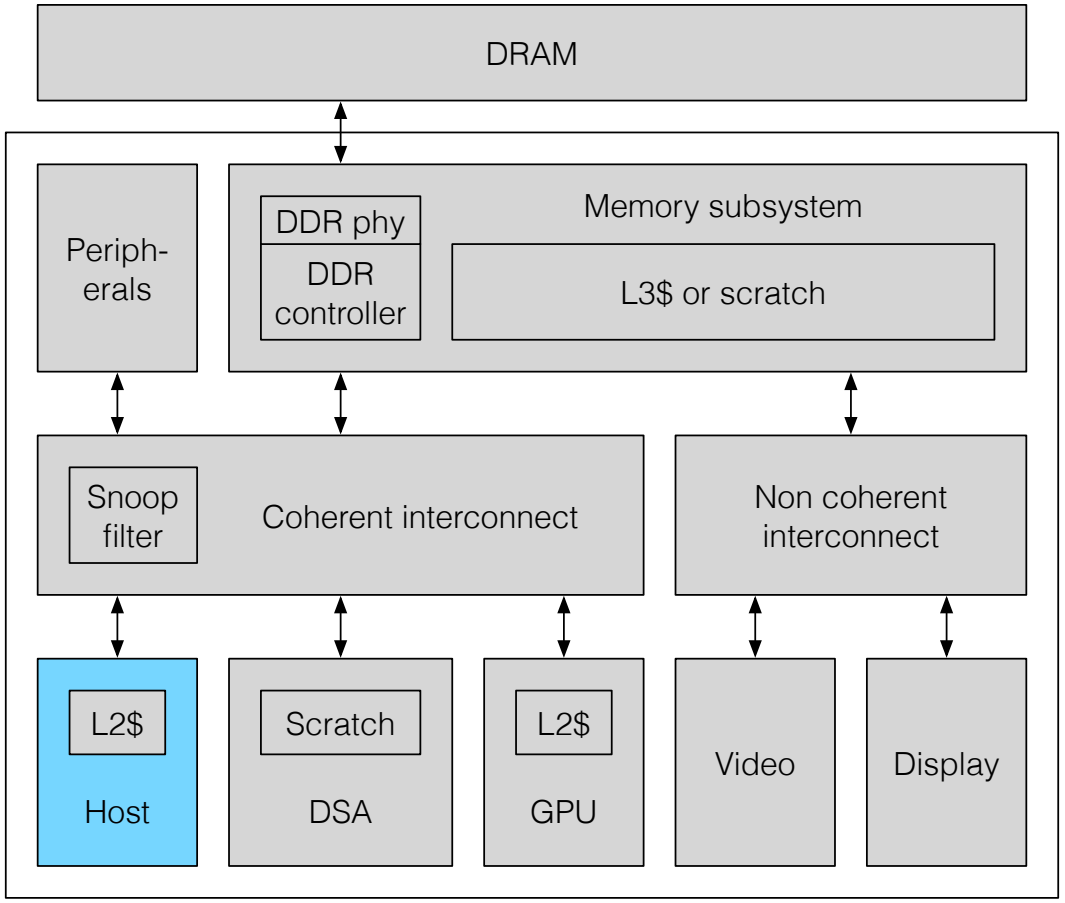
- External DRAM (not part of the SoC)
- Coherent
  - Host L2$
  - GPU L2$
  - L3$
  - DRAM (part)
- IO coherent
  - Peripherals
  - DSA
- Non coherent
  - Video
  - Display
  - L3 scratch
  - DRAM (part)



Host

- General compute that can do everything
  - Example architectures: x86, ARM, RISC-V, ...
  - This is what you want for optimizing the performance of large amounts of runtime dynamic hardware agnostic code
  - This is not what you want for optimizing the performance of small amounts compile time static hardware specific code
  - This is 1 possible get out of jail free card in the event that some portion of a network design does not map to the DSA
  - But in practice we'll use it to run a high level OS and offload pixels to the GPU and big compute to the DSA

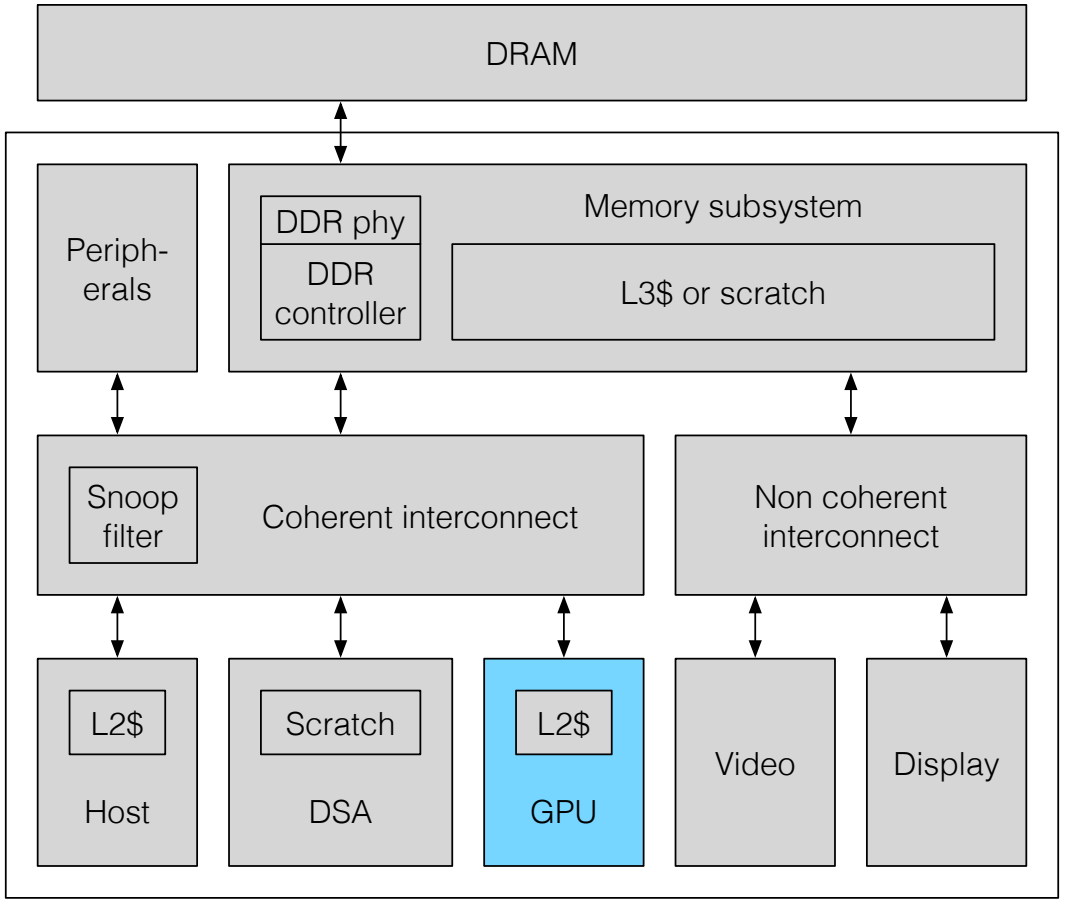
- Intelligence in hardware == limited optimization horizon, extra power, extra area and lower frequency
  - Cache
  - Branch prediction
  - Out of order processing
  - Speculative execution



GPU

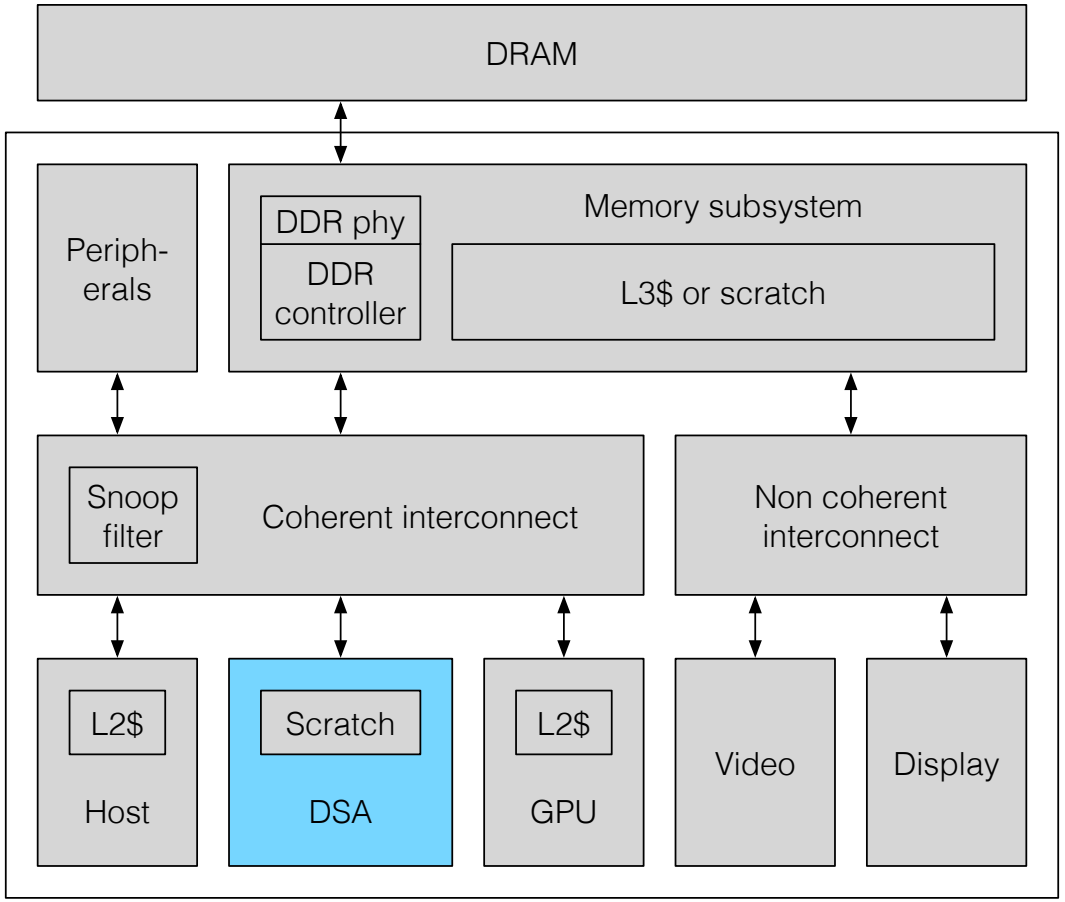
- A GPU is optimal for figuring out what pixels to put on a screen

- A GPU is not optimal for large matrix operations with static compile time graphs
  - Lots of parallel 3x3, 3x4 and 4x4 matrix multiplication is ok but suboptimal in terms of a dedicated architecture (re: N/3 compute to data movement ratio)
  - But in the absence of access to optimized hardware it's a convenient mechanism for training CNNs and it makes up for it architecture shortcomings with sheer size for applications that are less power constrained
  - And you can always use it to play video games when you're not training



Domain Specific Architecture

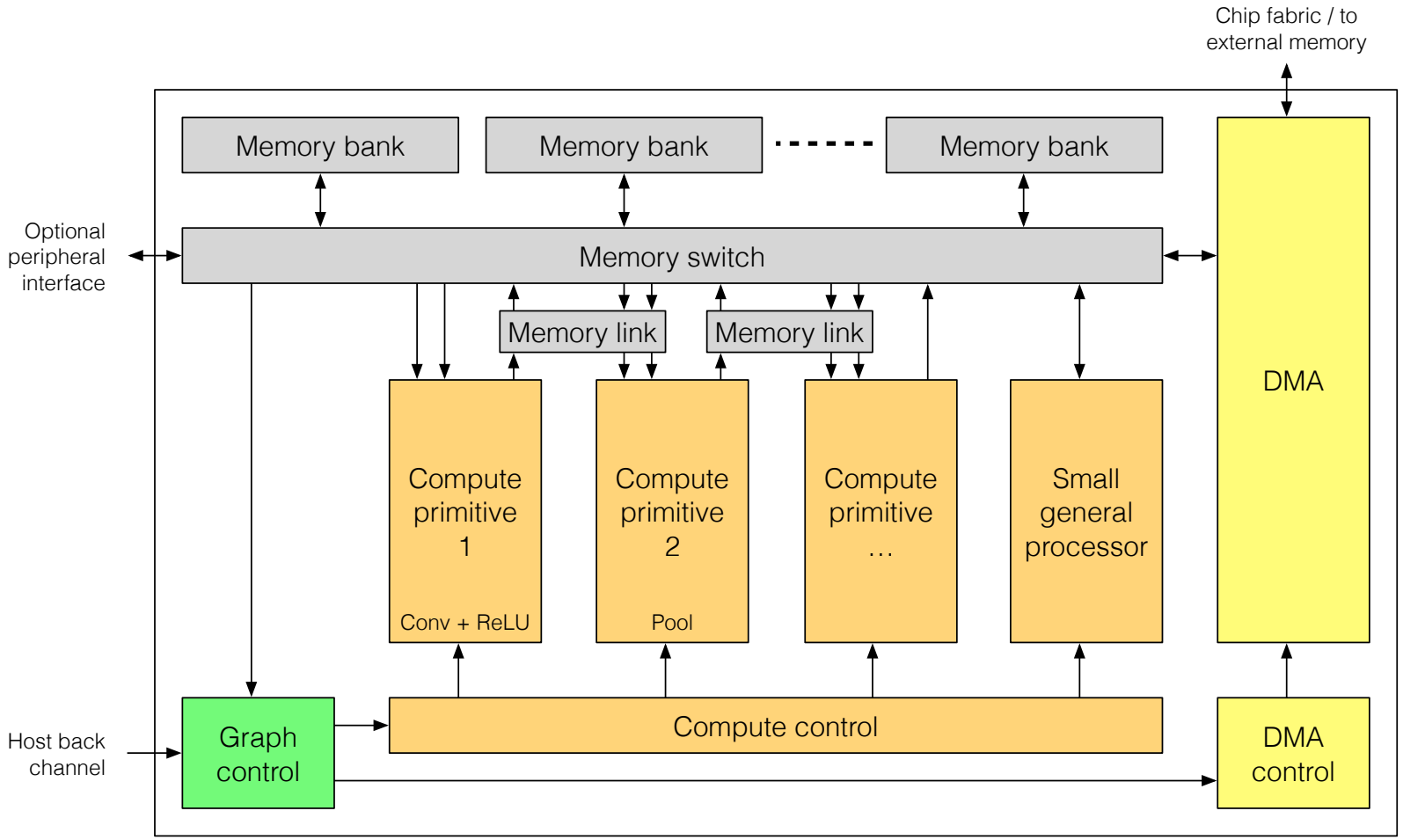
- Domain specific architecture
  - Hardware optimized for a specific application
  - Includes control, memory, data movement and compute
  - Can potentially give same benefits of multiple gens of Moore
- Can include on the coherent or non coherent interconnect depending on the application
  - Coherent
  - IO coherent
  - Non coherent
- Question: What is the optimal DSA for xNNs?
  - What type of memory is needed?
  - What type of control is needed?
  - What type of communication is needed?
  - What type of computation is needed?



Hardware – Domain Specific Architecture – Overview

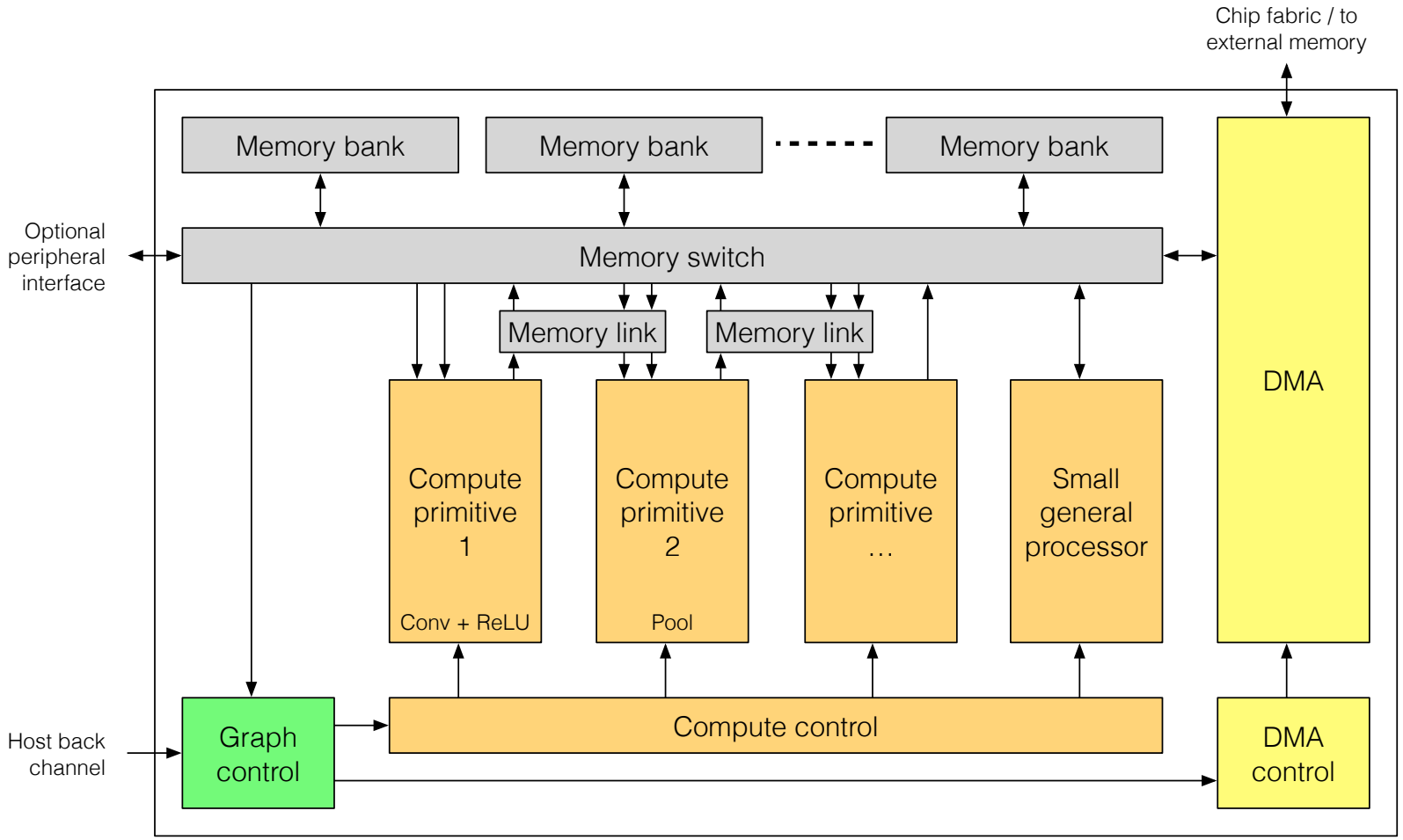
Computational Primitive Defined Domains

- Goal: keep domain specific optimality while allowing a high amount of generality
- Strategy: define the domain in terms of fundamental math, not an application
- Components
  - Memory
  - Control
  - Communication
  - Computation
- Control, communication and computation all operate in parallel



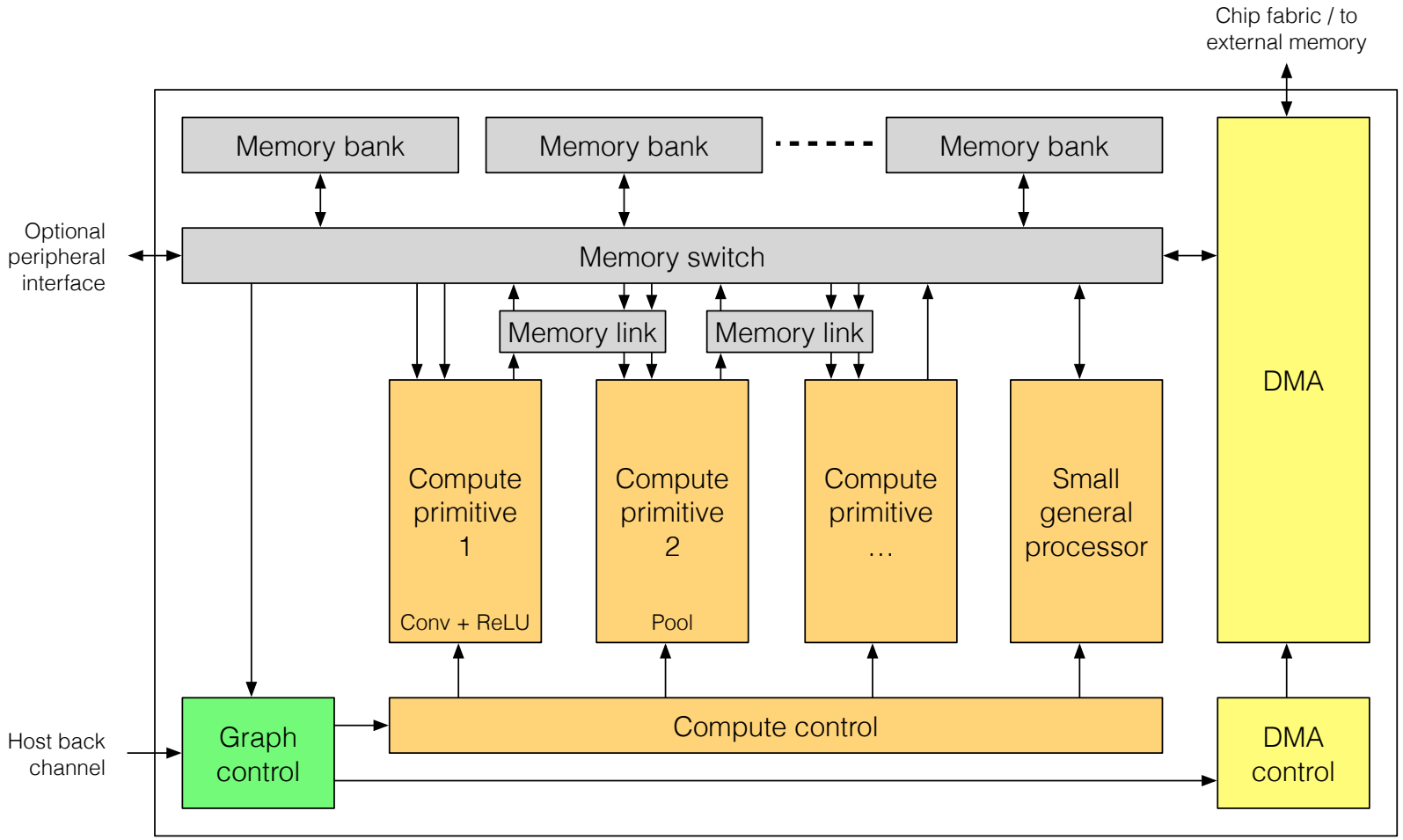
DSA Components

- Memory
  - Sized to allow most feature maps to remain on device
  - Frequency, banking and width matched to compute and communicate accelerators
- Control
  - Loads instructions from internal memory to control, compute and communication instruction queues
  - Starts execution of instructions in control, compute and communicate instruction queues and clears dependencies when complete



DSA Components

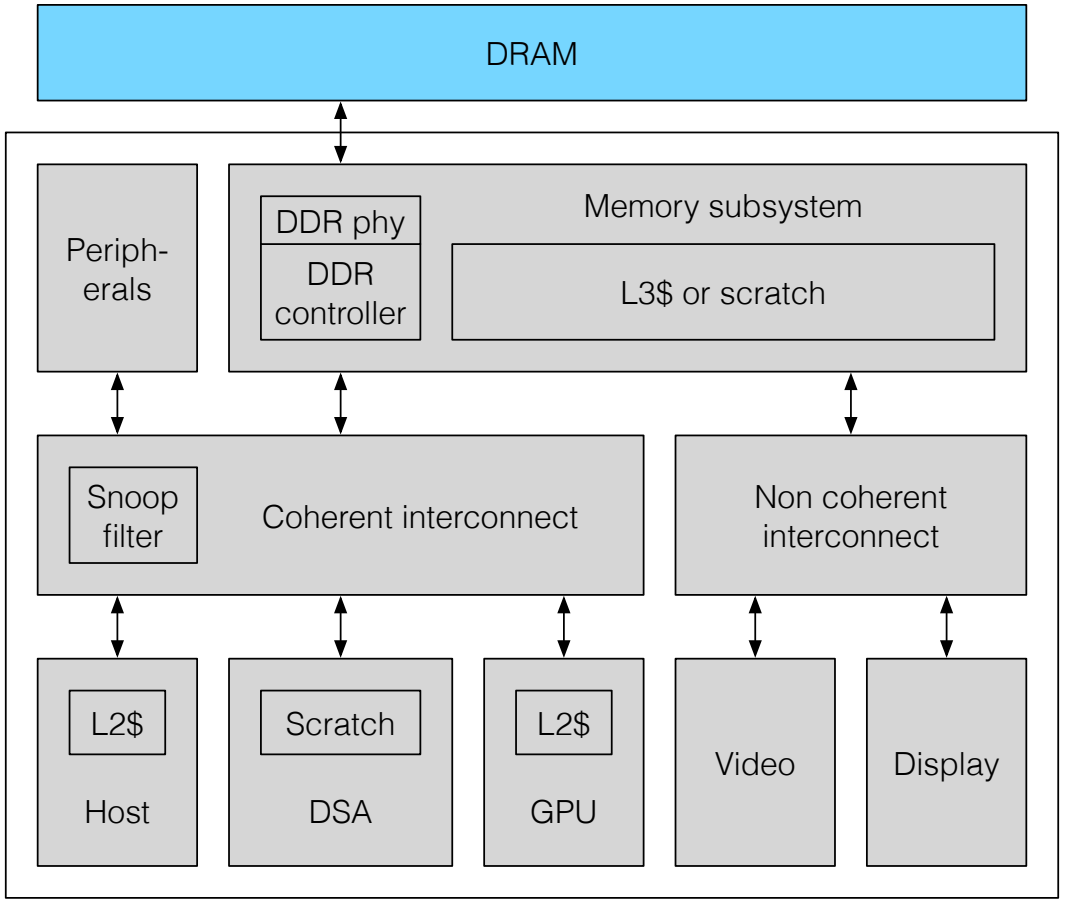
- Communication
  - Transform primitive approach to communication using a configure then execute instruction format
  - Read data from internal (external) memory, transform and write data to external (internal) memory
- Computation
  - Computational primitive approach to acceleration using a configure then execute instruction format
  - Load input data from internal memory, compute and write output data to internal memory



Hardware – Domain Specific Architecture – Memory

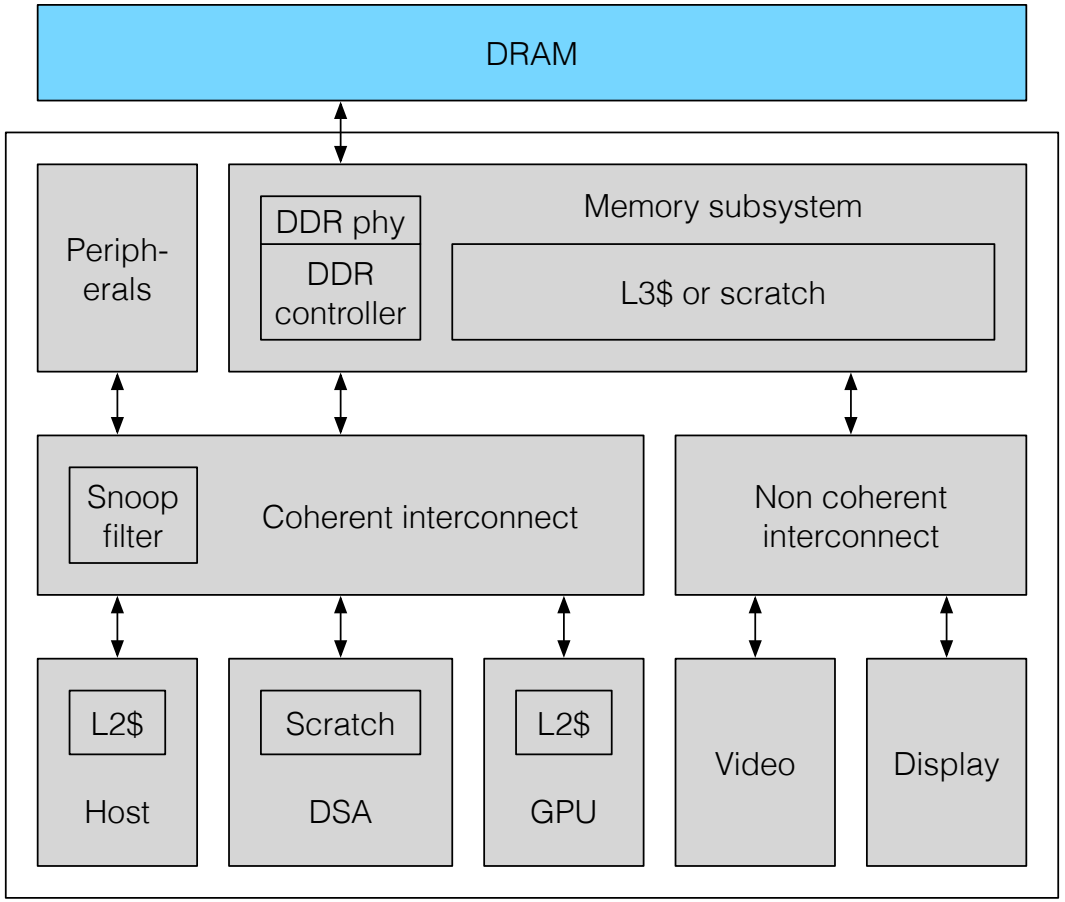
External Memory

- DRAM
  - Use for off device volatile data storage
  - Most common types are DDRx SDRAM
- Memory cell
  - Data is stored as charge on a capacitor representing a bit
  - Memory cells require 1 transistor and capacitor per bit
  - Because charge leaks from the capacitor DRAM needs an external circuit to continually refresh the data
- Organization
  - (Banks * rows * columns) x bits
  - Commonly most efficient with ~ 64 B alignment and multiple of 64 B accesses; specific alignment and access size is a function of the specific memory
  - This affects data arrangement and memory accesses



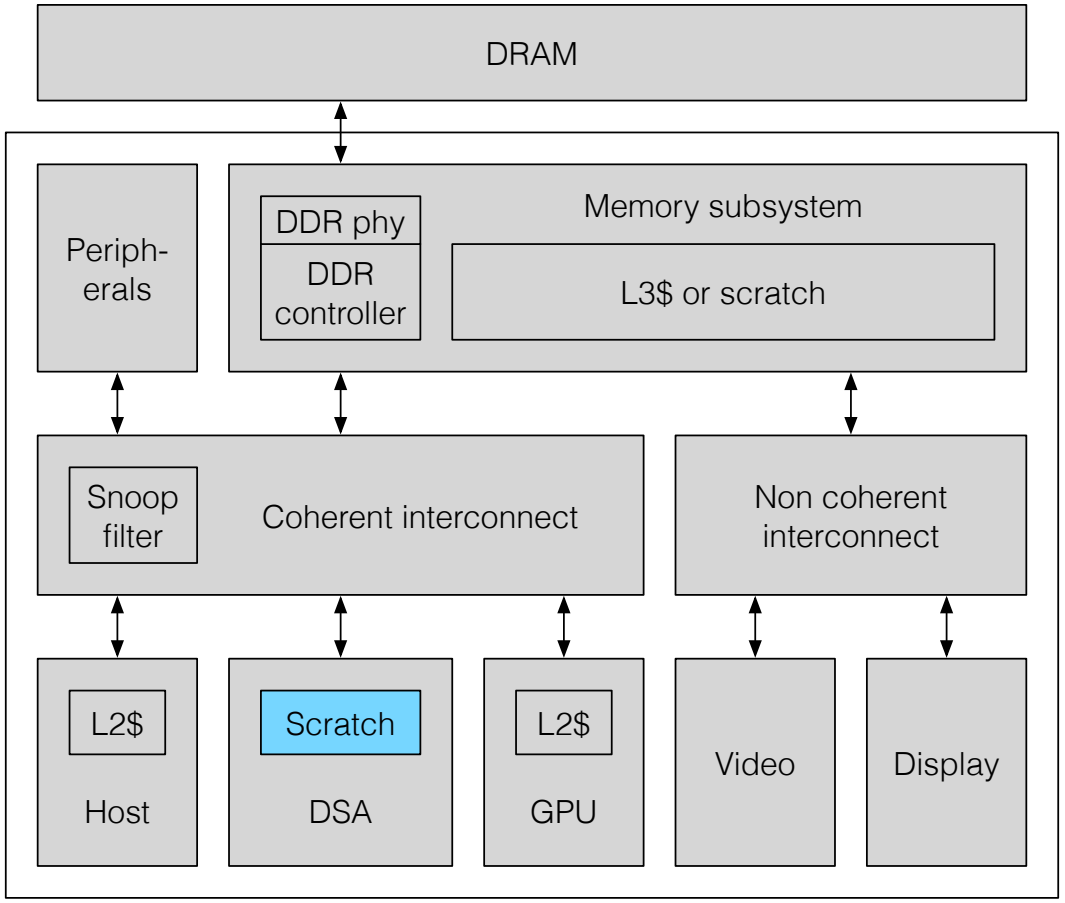
External Memory

- Comments
  - High off device latencies can usually be hidden in throughput optimized compute using on device memory
  - Cheaper per bit but slower than SRAM
- Typical uses of external memory for xNNs
  - For our purposes assumed to be ~ infinite (unless working with extremely large networks or extremely small systems)
  - Dynamic network inputs (unless coming from a direct peripheral interface or another linked graph in the session)
  - Dynamic network outputs (unless a linked graph in the session)
  - Filter coefficients (unless all very small)
- Occasionally uses of external memory for xNNs
  - A buffer for intermediate feature maps when they're too big to fit on device



Internal Memory

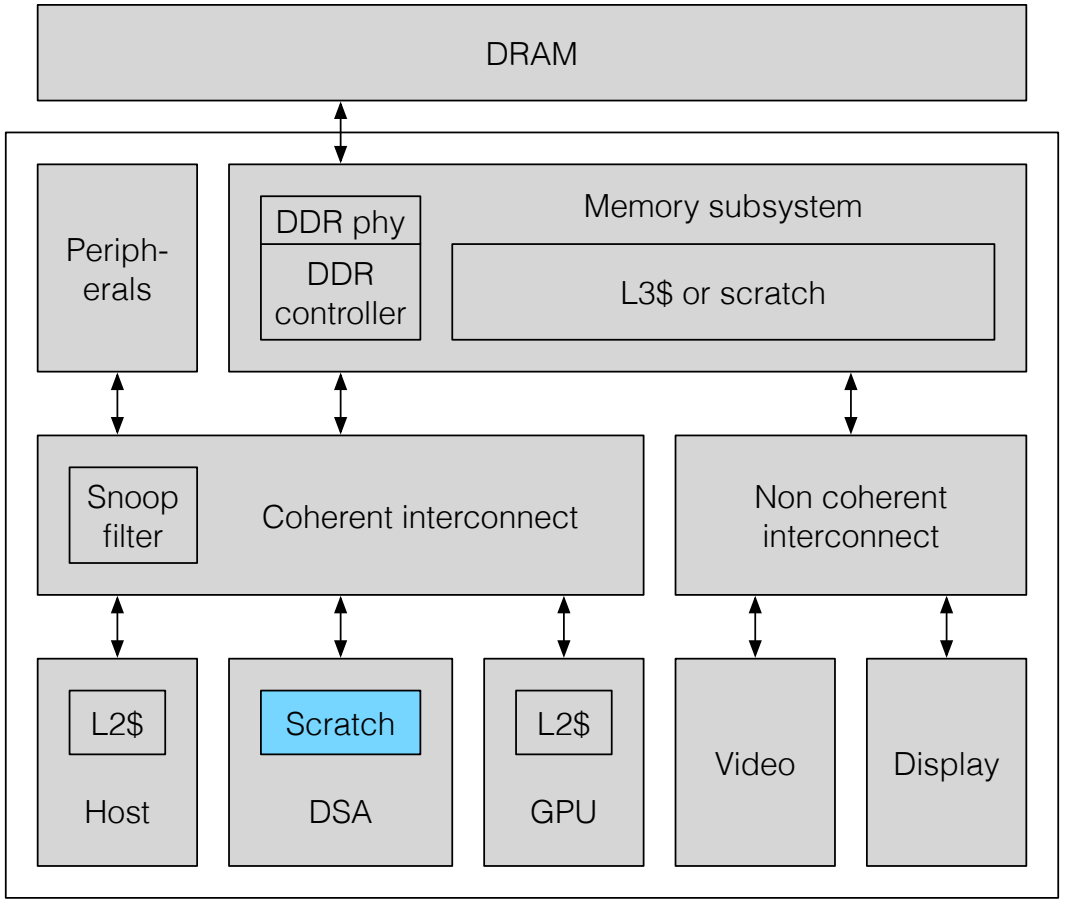
- SRAM
  - Use for on device volatile data storage
  - Compute out of SRAM (maybe 1 step removed)
- Memory cell
  - Data is stored in a flip flop representing a bit
  - Memory cells require 4 - 6 transistors per bit
- Organization
  - Divided into multiple banks where each bank can be thought of as a 2D array of bits / bytes
  - Access are most efficient that read a row at a time
  - Applications spread data across multiple banks for multiple simultaneous read / write operations
  - Either use bank randomization or coordinated memory arrangements to minimize delays caused by multiple simultaneous read / write operations to the same bank



Internal Memory

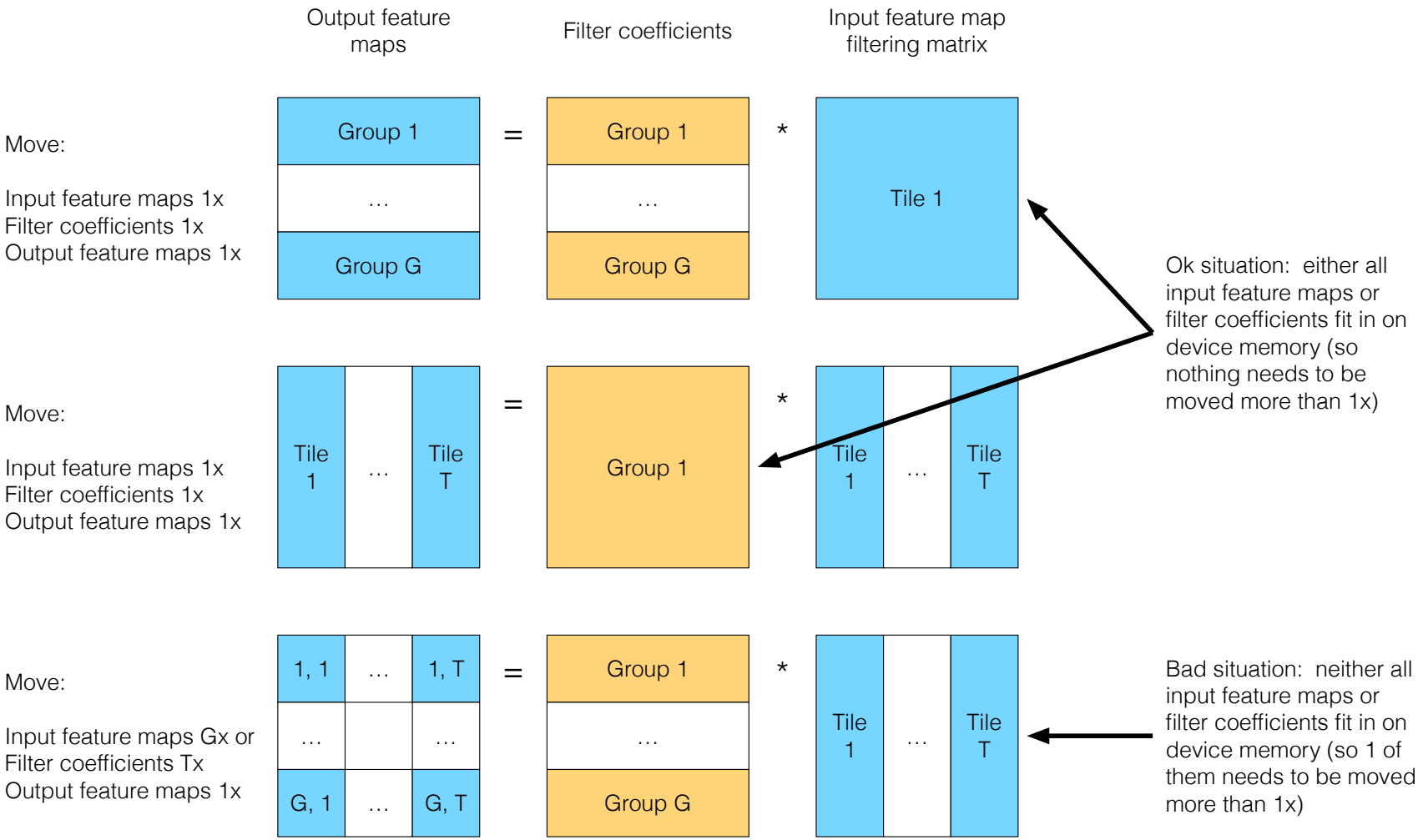
- Example
  - 2^4 = 16 banks of 2^10 = 1024 rows of 2^6 = 64 bytes
  - Total memory = 2^4 * 2^10 * 2^6 = 2^20 ~ 1 MB
  - Up to 16 parallel accesses are possible (for single port designs, though typically would design for many fewer simultaneous read / write access to avoid collisions and there are also implications wrt the memory mux / switch connecting memory banks to IPs)
  - Accesses are most efficient when the starting address is a multiple of 64 bytes and 64 bytes are read at a time

- Typical uses of internal memory for xNNs
  - Finite (though frequently occupying a large fraction of an optimal device)
  - Input and output feature maps for internal graph edges
  - Filter coefficients for the current layer

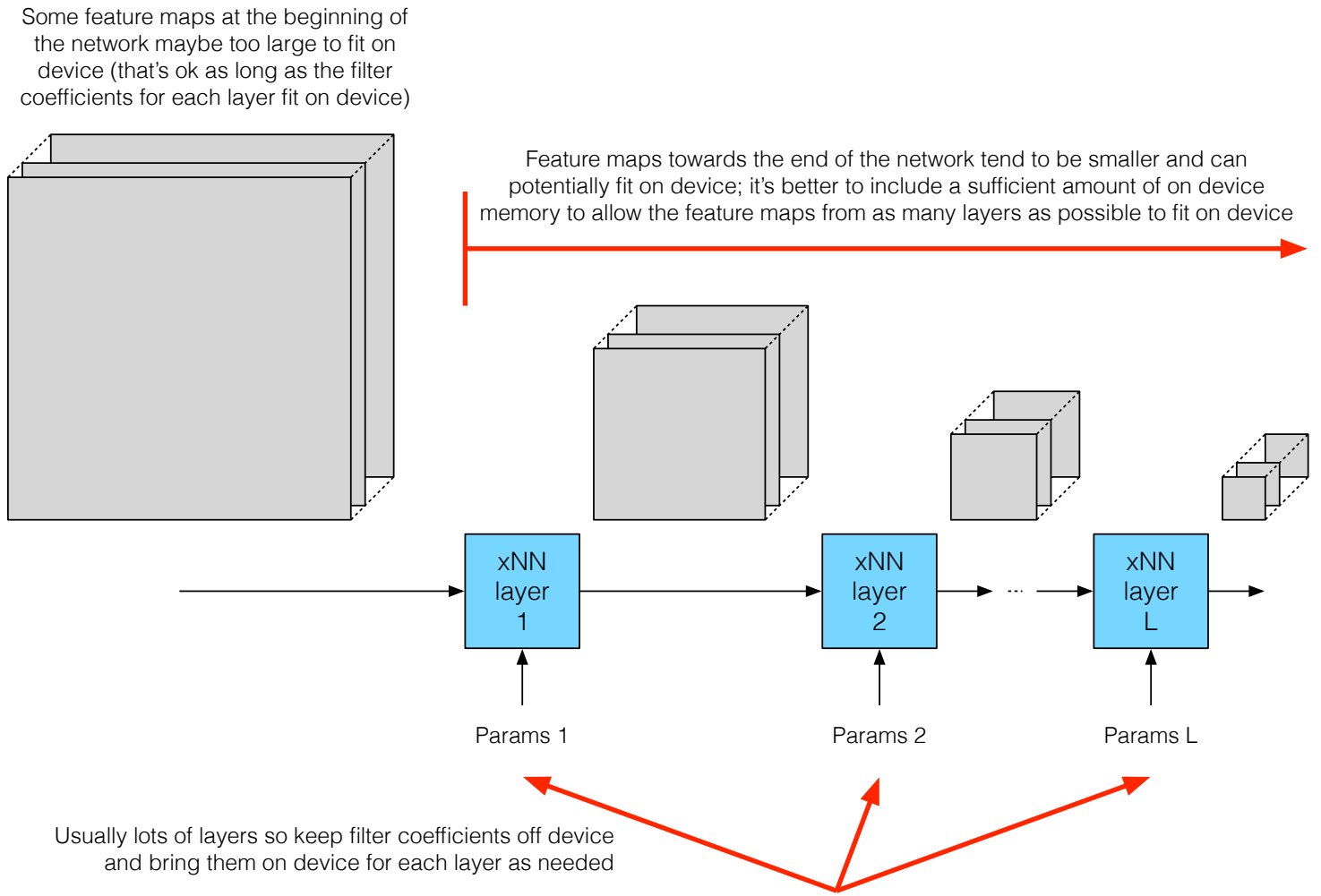


How Much Memory Is Ok

Sufficient on device memory such that for each layer (considered individually) either all feature maps or all filter coefficients fit on device



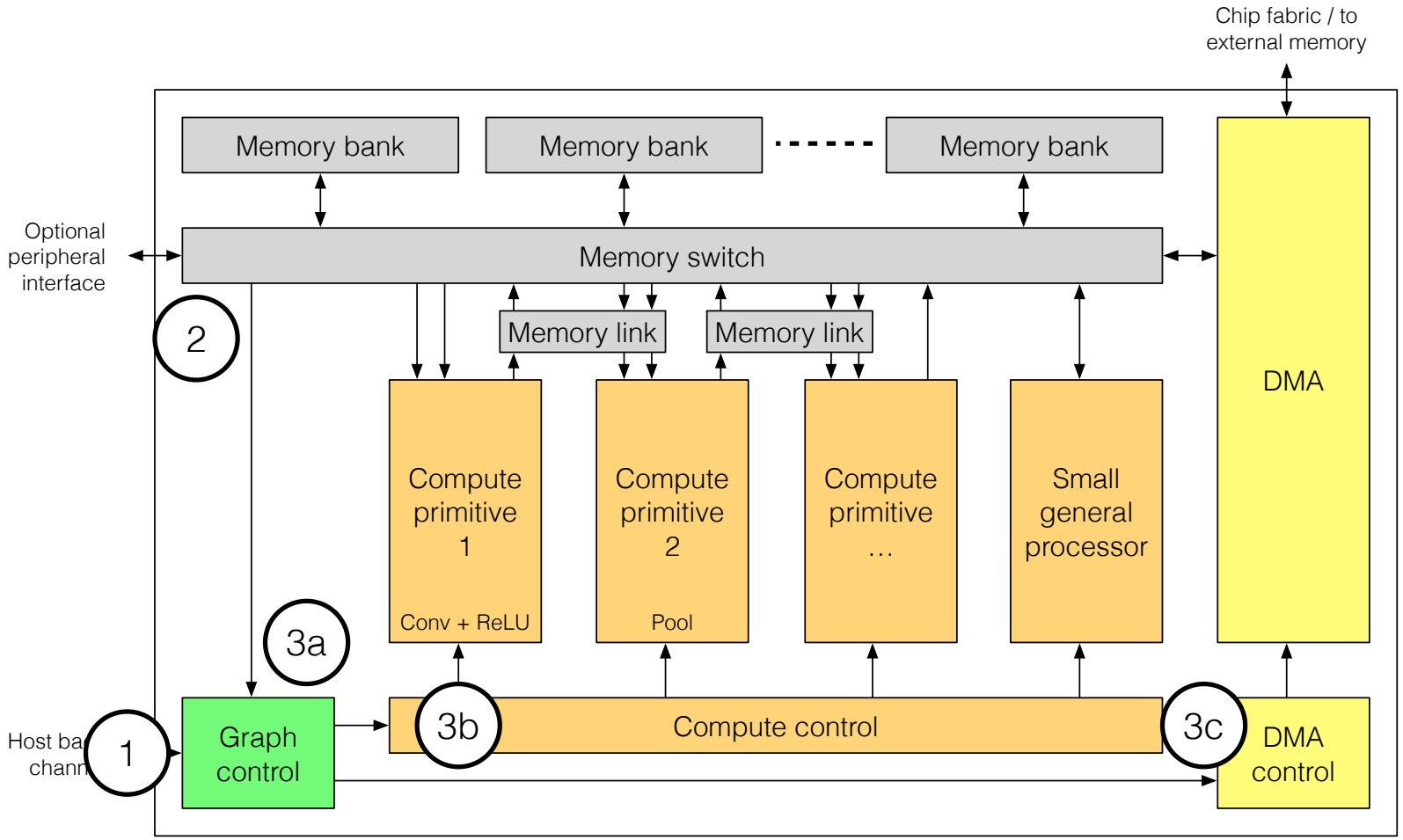
How Much Memory Is Better



Hardware – Domain Specific Architecture – Control

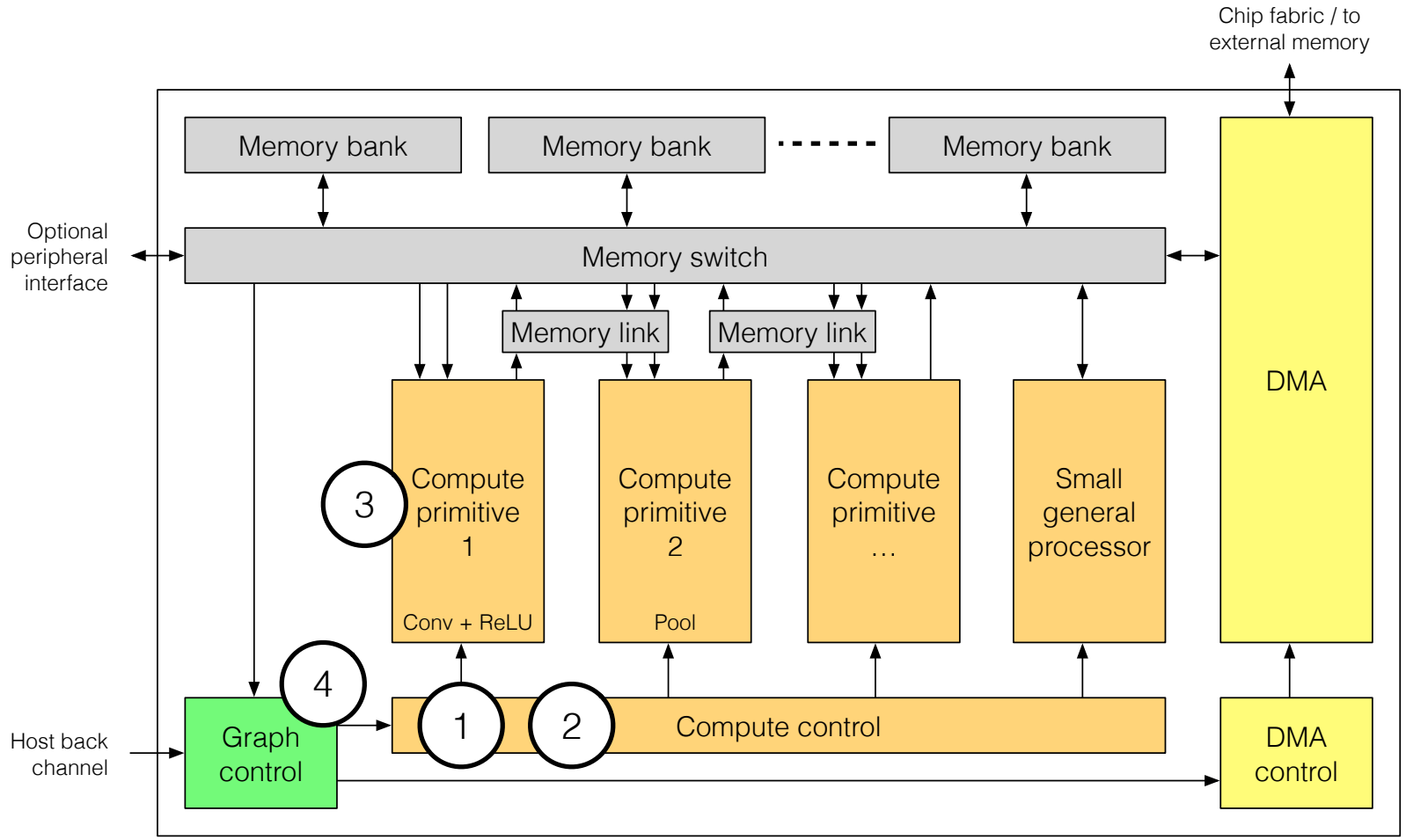
Graph Control

- Graph control does the following
  - 1 Finds the first node descriptor with all dependencies satisfied
  - 2 Reads a block of local memory specified by the node descriptor
  - 3 Writes the memory block to the node descriptor specified control component
  - 3a The control component can be the graph control load more nodes
  - 3b The control component can be the compute control to load instructions for a compute primitive
  - 3c The control component can be the DMA control to load instructions for the DMA



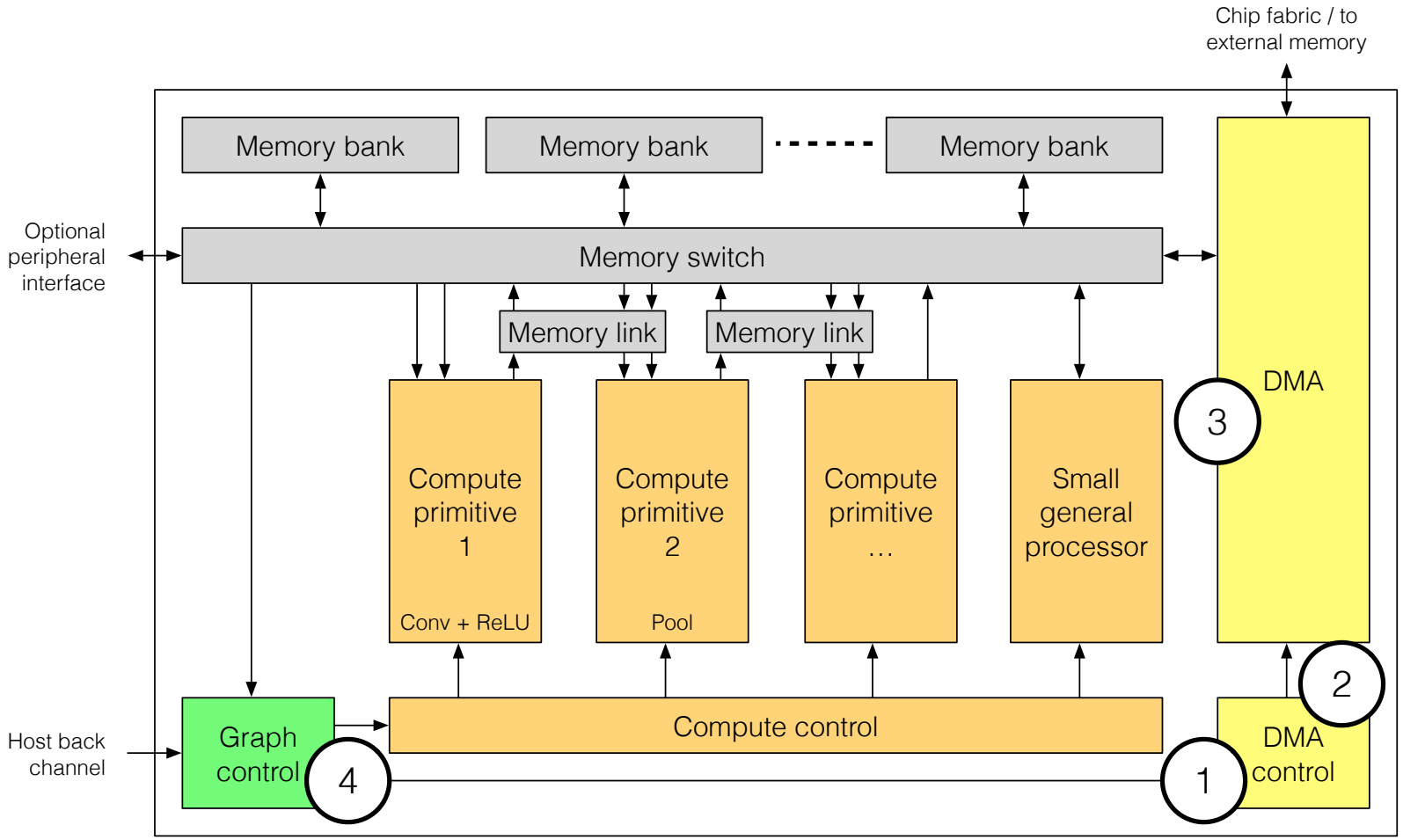
Compute Control

- Compute control does the following
  - 1 Reads the next compute instruction
  - 2 Generic looping instructions are processed by the compute control itself and create a deterministic pattern / looping sequence for the next set of instructions
  - 3 Specific compute instructions are passed to the appropriate compute accelerator; typically these are composed of compute specific state machine initialization and execution
  - 4 Generic return instructions are processed by the compute control itself and send a message back to the graph control (e.g., to indicate a node is complete and to clear dependencies in the node descriptor)



DMA Control

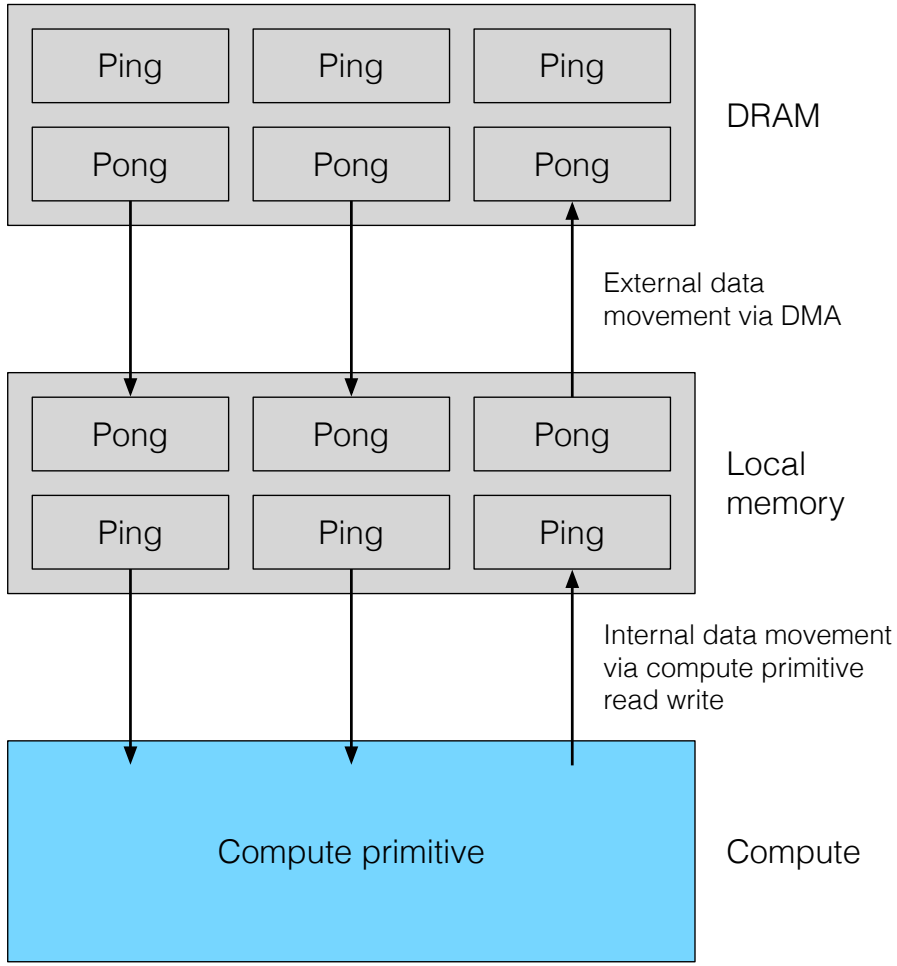
- DMA control does the same things that compute control does, just for the DMA (vs the compute primitives)



Hardware – Domain Specific Architecture – Communication

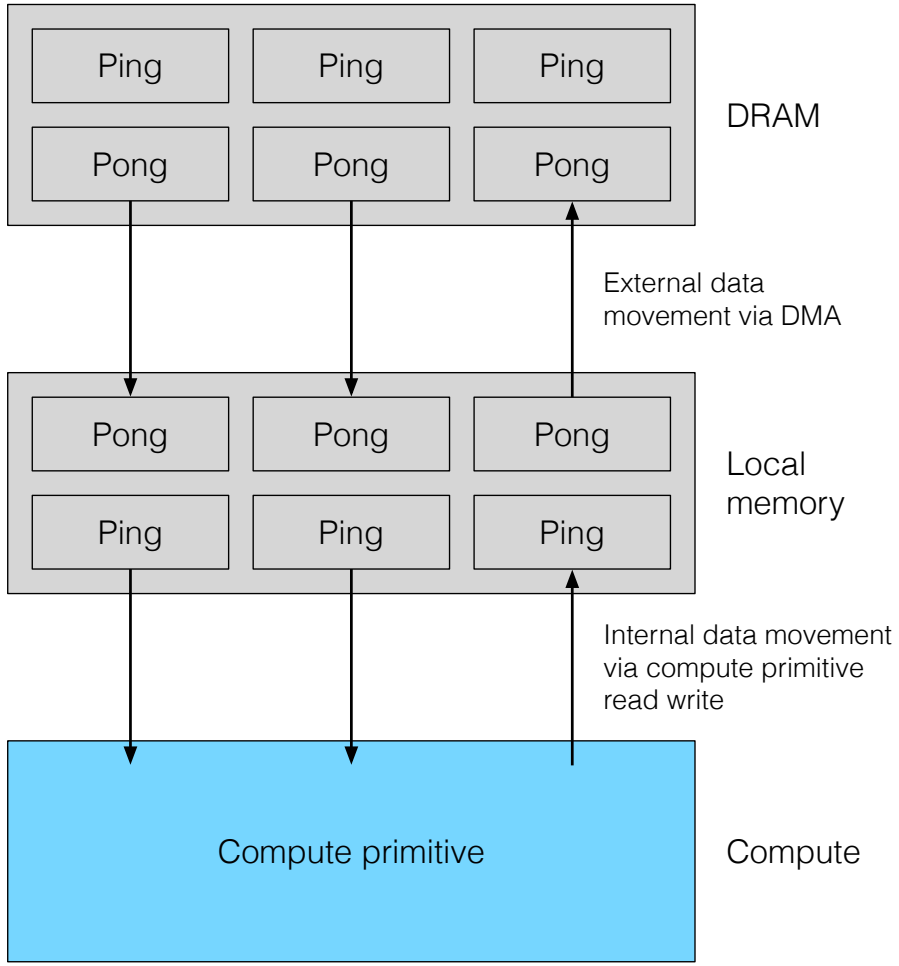
CNN Strategy

- Create ping pong buffers in DRAM and local memory if needed to allow continual parallel compute
- Attempt to keep feature maps on device and bring in filter coefficients as needed per layer
  - This removes the need for feature maps to be involved in the ping pong scheme
  - Great if all filter coefficients fit on device too, but this is usually not the case
- Do the following in parallel
  - External to local memory data movement
  - Local memory to compute data movement
  - Compute



CNN Strategy

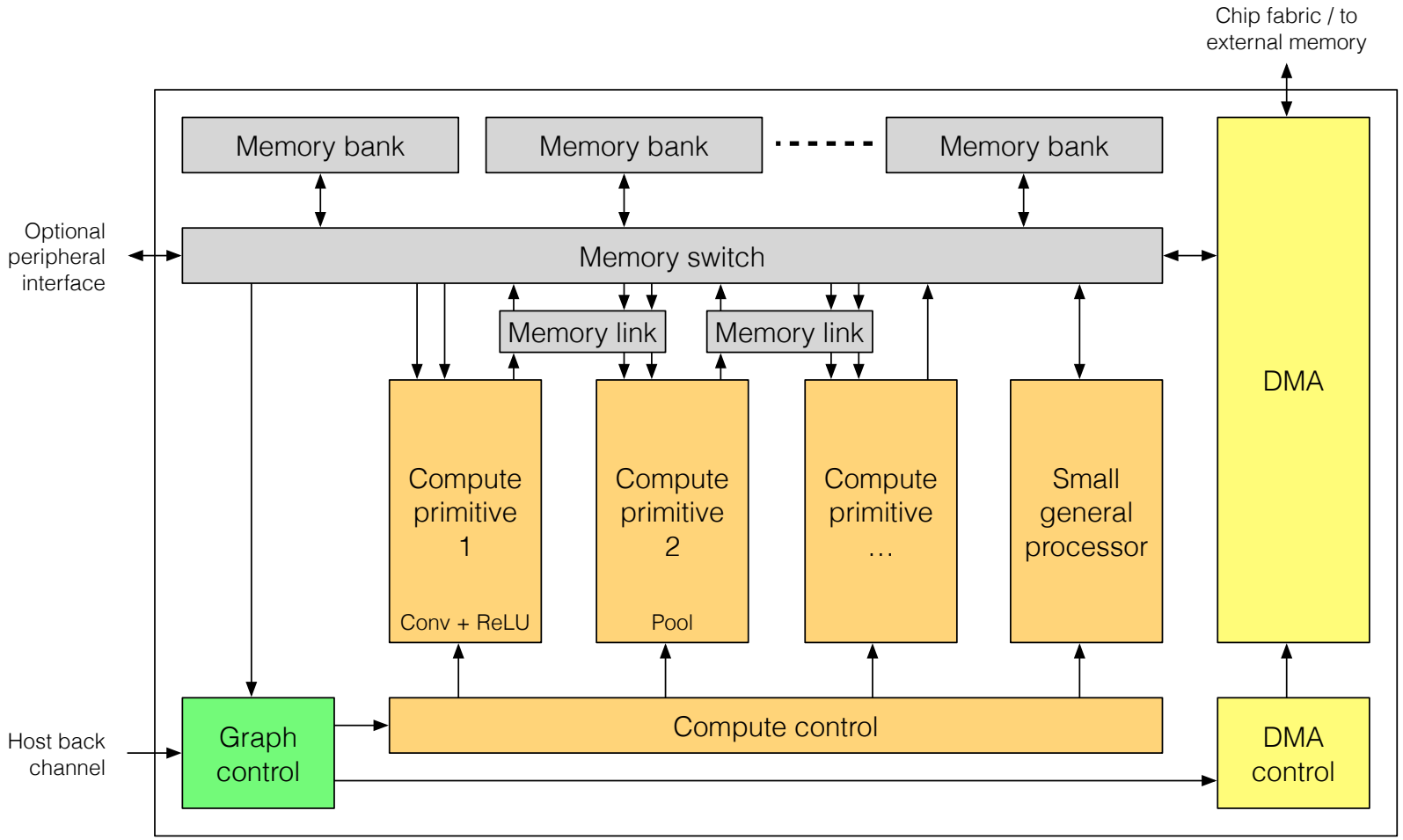
- Typically, external memory bandwidth is much less than internal memory bandwidth
- This implies 2 options for efficiency
  - Need to keep some fraction of data on device
  - Need to have a high level of data reuse once on device



DMA

- External – internal data movement is the job of the DMA
  - Read from local memory and write to external memory
  - Read from external memory and write to local memory
  - Operates in parallel to graph control and compute control to allow parallel ping pong data movement and compute

- External memory is typically DRAM attached to the current device but can also be a memory space on a different device

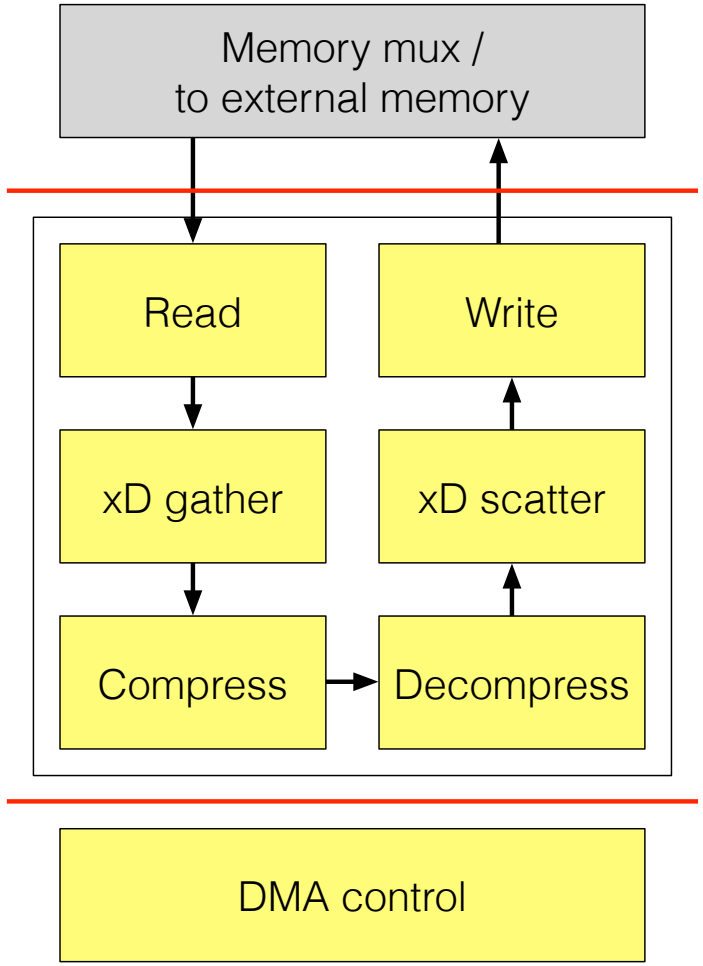


DMA

- Example data flow and control are listed below, other options are possible
  - Note that all data that moves between DRAM and the DSA local memory moves via the DMA

- Data flow
  - xD gather: vector read from local / DRAM memory
  - Transform: compress / decompress
  - xD scatter: vector write to DRAM / local memory

- Control via instructions from the DMA queue manager
  - State machine initialization
  - State machine execution



Hardware – Domain Specific Architecture – Computation

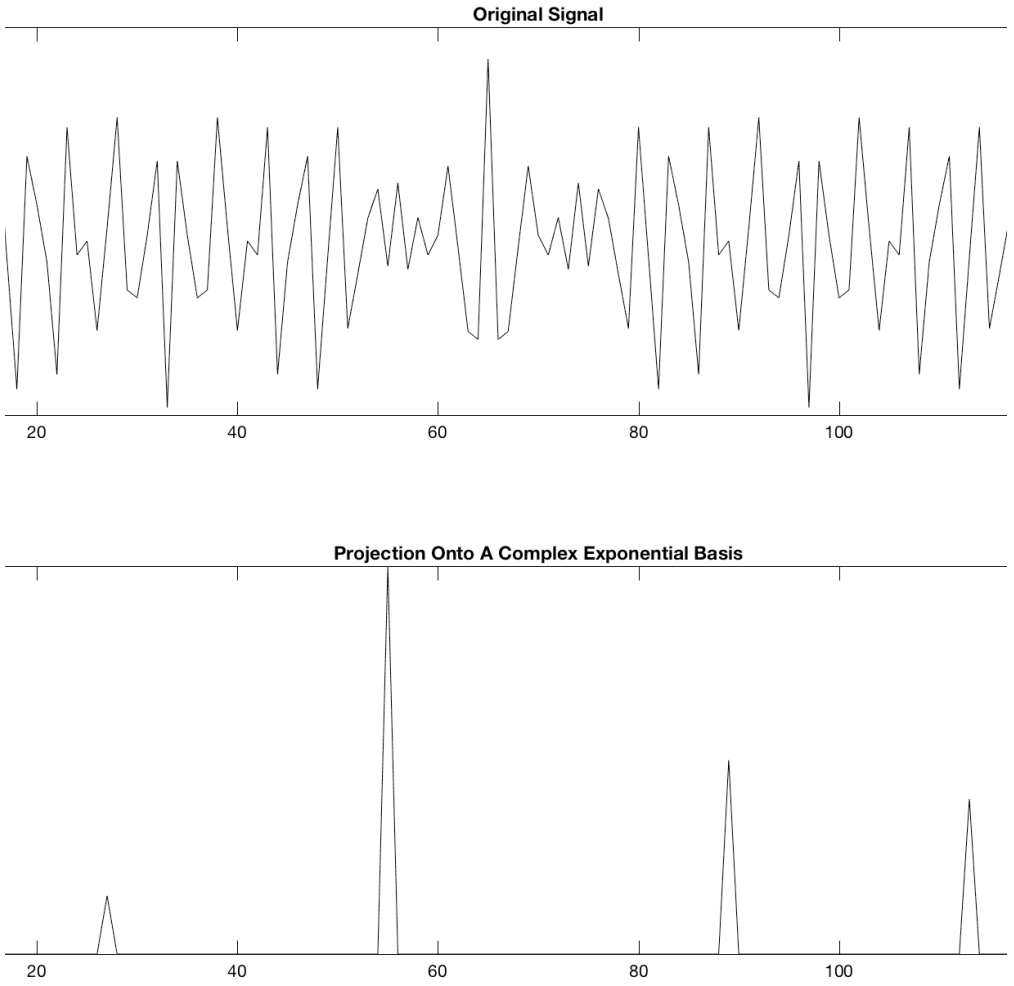
You Don't Put High Level Algorithms In Gates

- Unless they're part of the fixed portion of a standard
  - The transmitter part of a communication standard
  - The decoder part of a compression standard
  - …
- Algorithm designers can change their minds like you change your socks (relatively frequently)
- Hardware designers create silicon at a much much slower and more expensive pace
- The question: how do you get the efficiency of a dedicated accelerator but still enable generality with respect to high level algorithms?

An Analogy For Thinking About Compute

A small detour before answering the question; a FFT projects a signal onto a complex exponential basis and tells you how much of each basis component was in the original signal

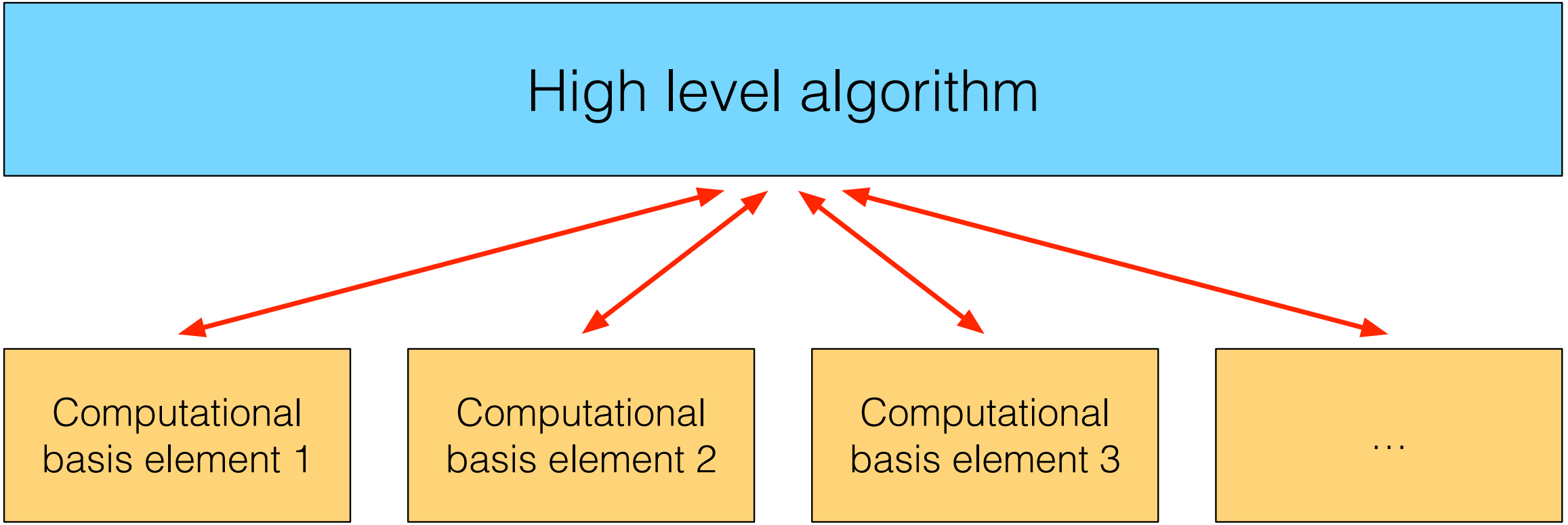
FFT



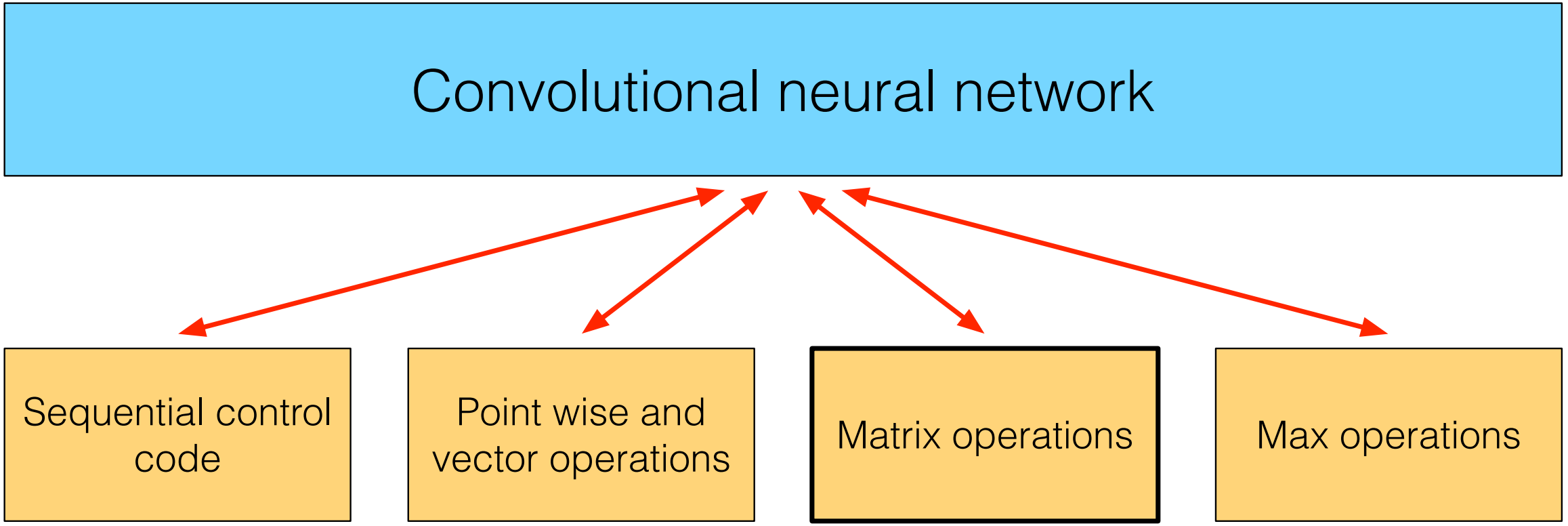
Dear observant class member: yes the top figure is a bit of a cheat (just the abs of the sequence) because I was too lazy when making figures to construct a real even signal; but the point of the slide is valid

A Computational Basis For Algorithms

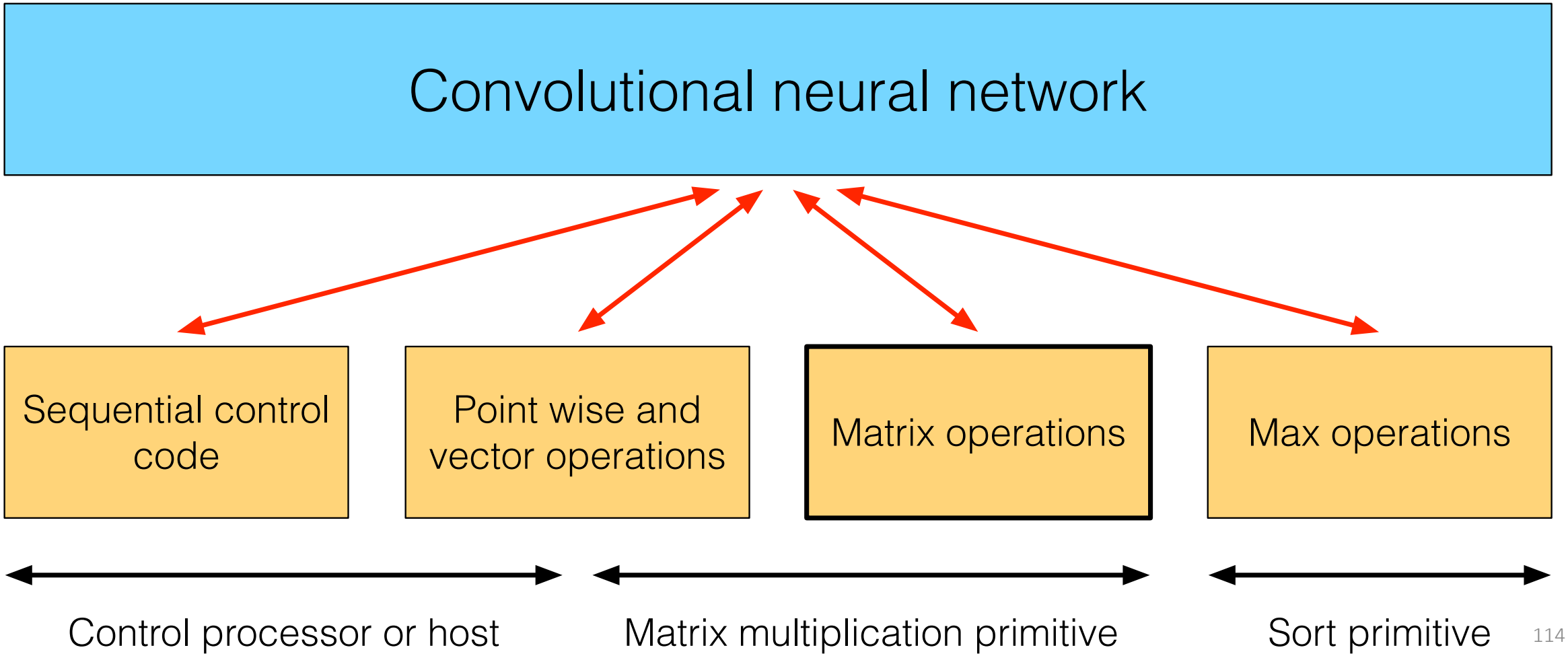
Answer to the question: use the same strategy for designing hardware, decompose high level algorithms onto a computational basis and provide optimized implementations of key computational basis elements (get ASIC efficiency while maintaining algorithmic generality)



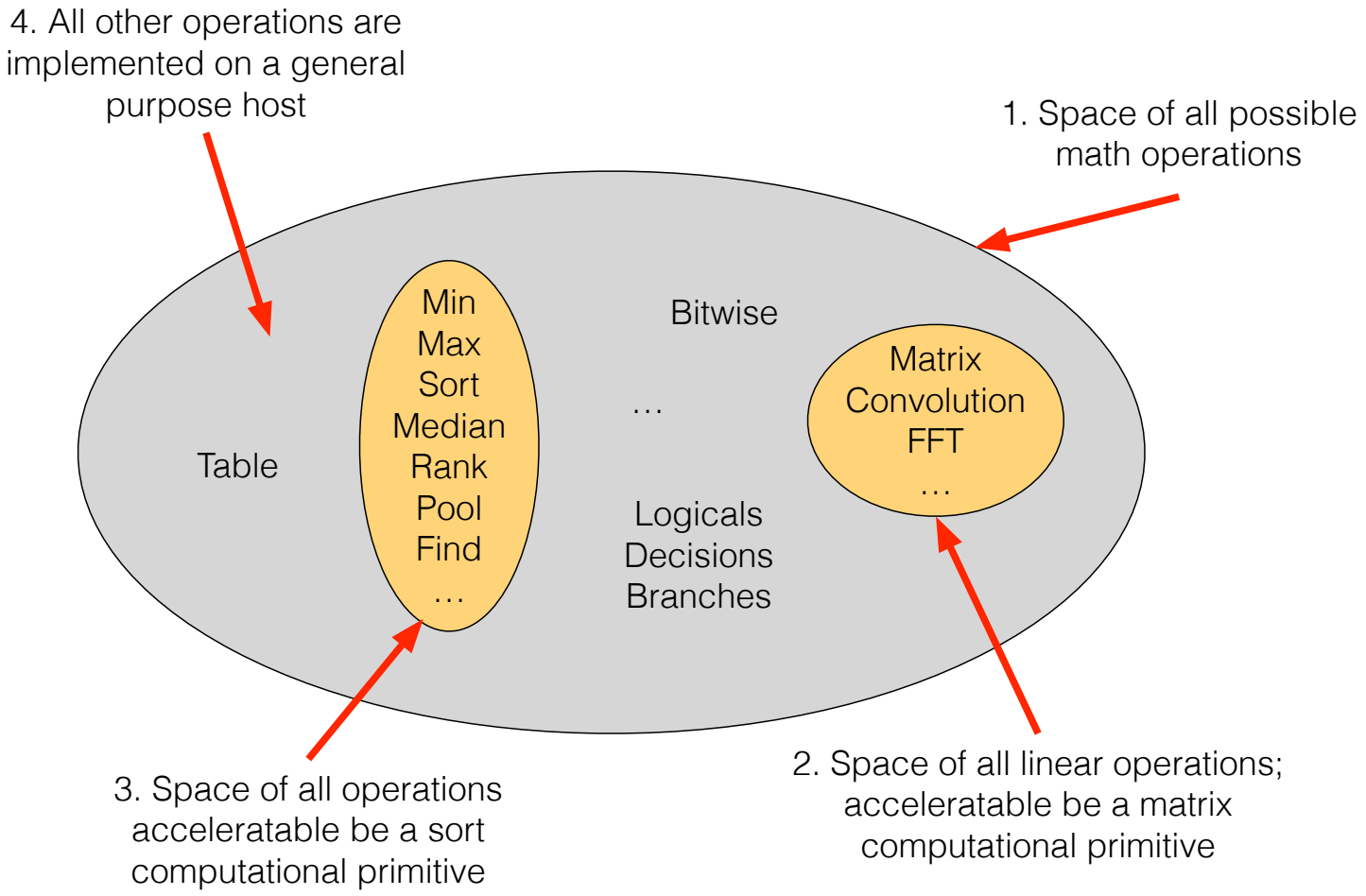
Projecting A xNN Onto A Computational Basis



Projecting A xNN Onto A Computational Basis



An Alternate View Of The Last Few Slides



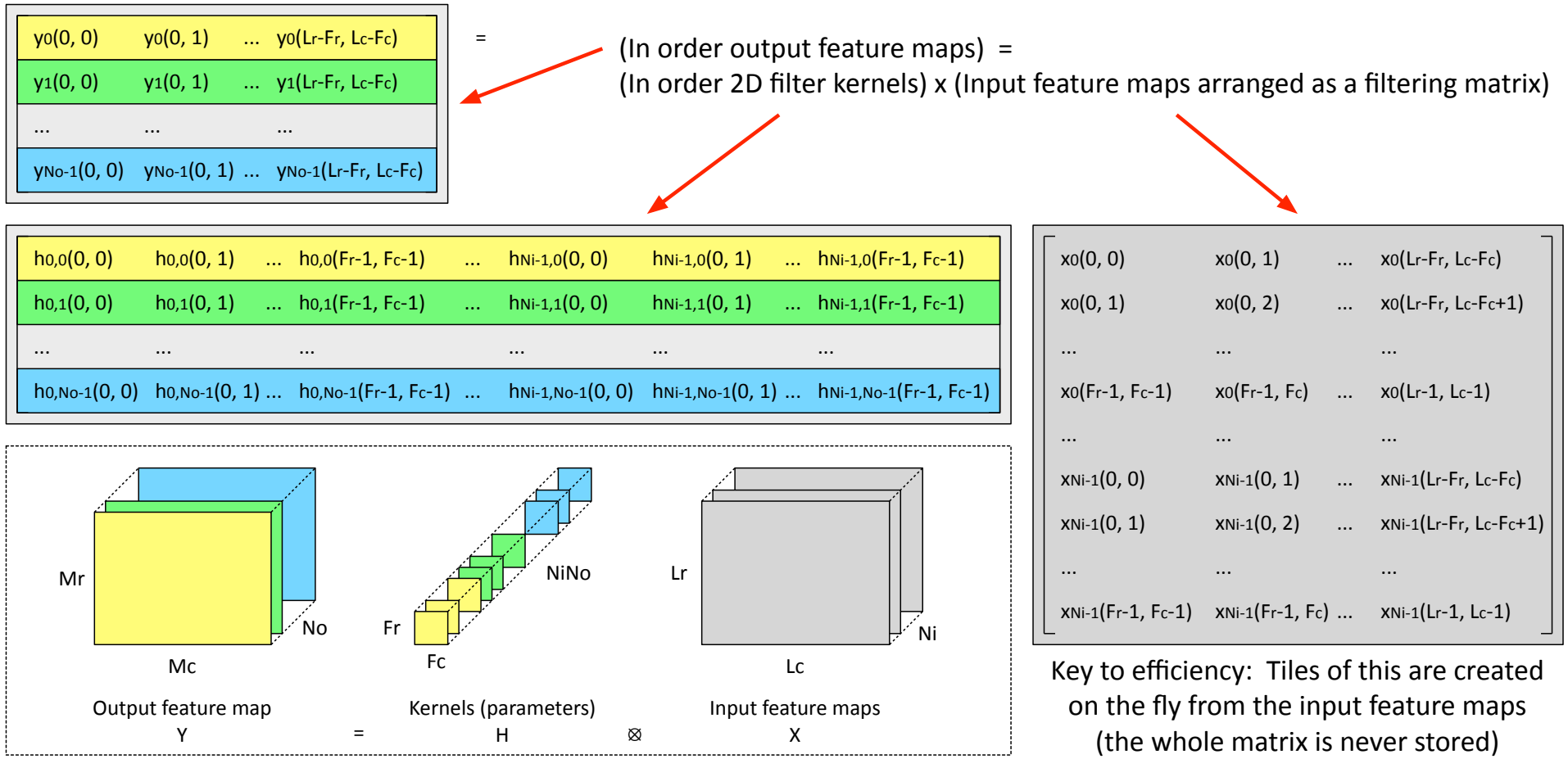
Reminder: Matrix Compute Is Special

- Scalar: c += a b
  - 1 MAC, 3 pieces of data
  - Arithmetic intensity = 1/3
- Vector (N x 1 point wise): **c += a ⊙ b**
  - N MACs, 3N pieces of data
  - Arithmetic intensity = N/3N = 1/3
- Matrix (M = N = K): **C += A B**
  - N^3 MACs, $3N^2$ pieces of data
  - Arithmetic intensity = $N^3/3N^2$ = N/3

Why are bubbles spherical shaped? Why choose a square matrix? Because square matrix sizes maximize the compute to data ratio (max M*N*K given M*N + M*K + N*K = constant → M = N = K)

Why is 1 large accelerator better than many small accelerators? Because it minimizes the excess data movement and delay for inherently sequential operations

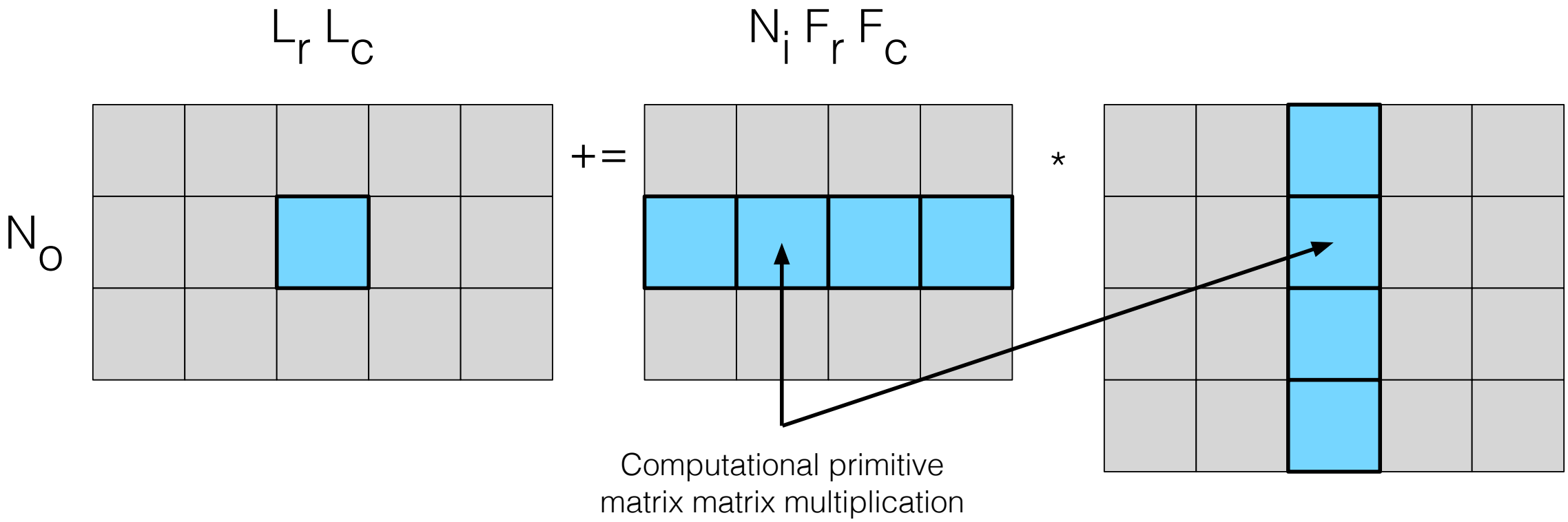
CNN Style 2D Conv Is Large Matrix Mult



Key to efficiency: Tiles of this are created on the fly from the input feature maps (the whole matrix is never stored)

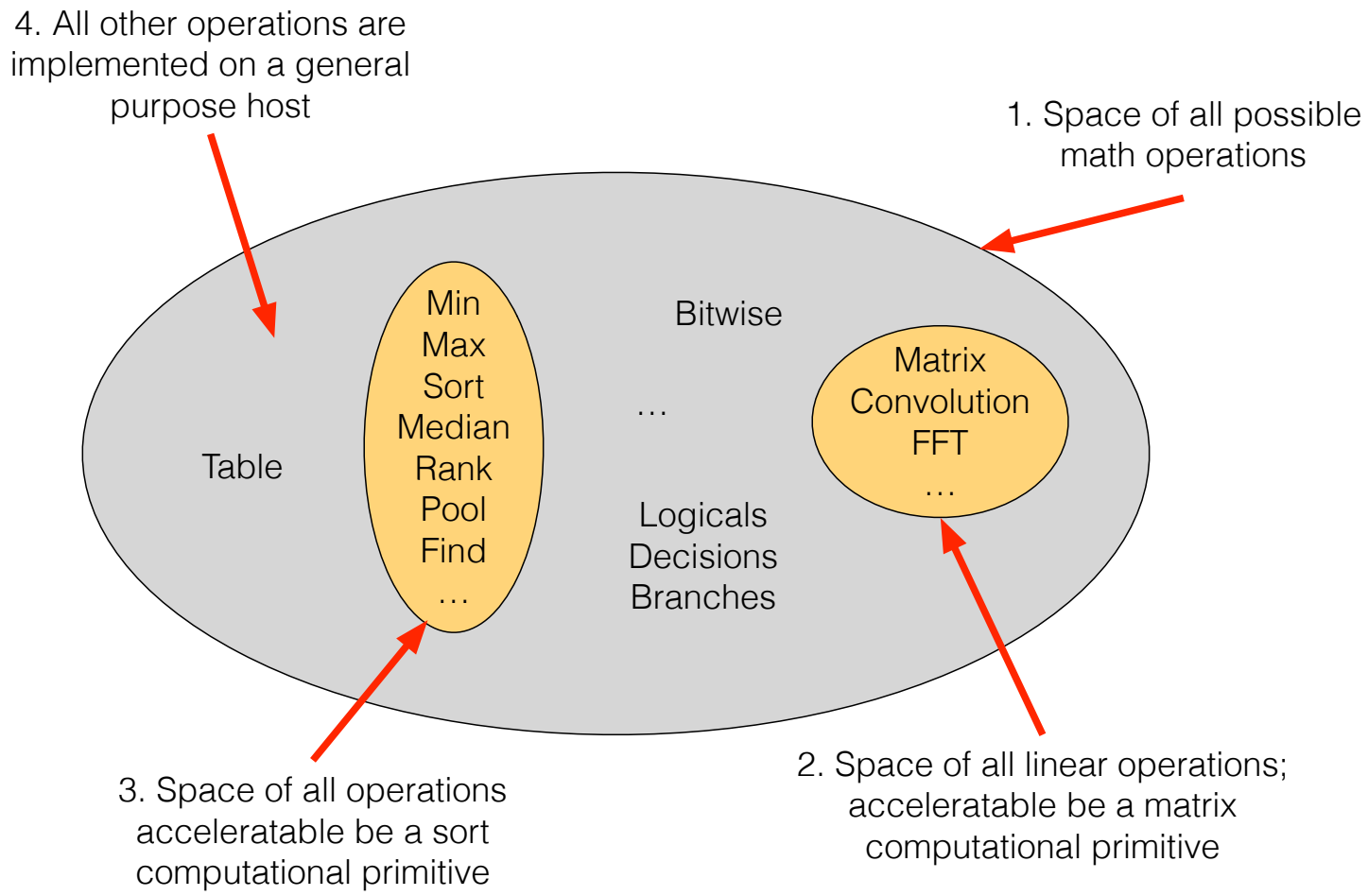
Consider the extremes
- If Fr = Fc = 1 this reduces to matrix matrix multiplication as would be used by a fully connected layer processing multiple regions or batches
- If Fr = Lr, Fc = Lc this reduces to matrix vector multiplication; not possible for any method to accelerate well because bandwidth limited

Large Matrix Mult Via Tiled Matrix Mult



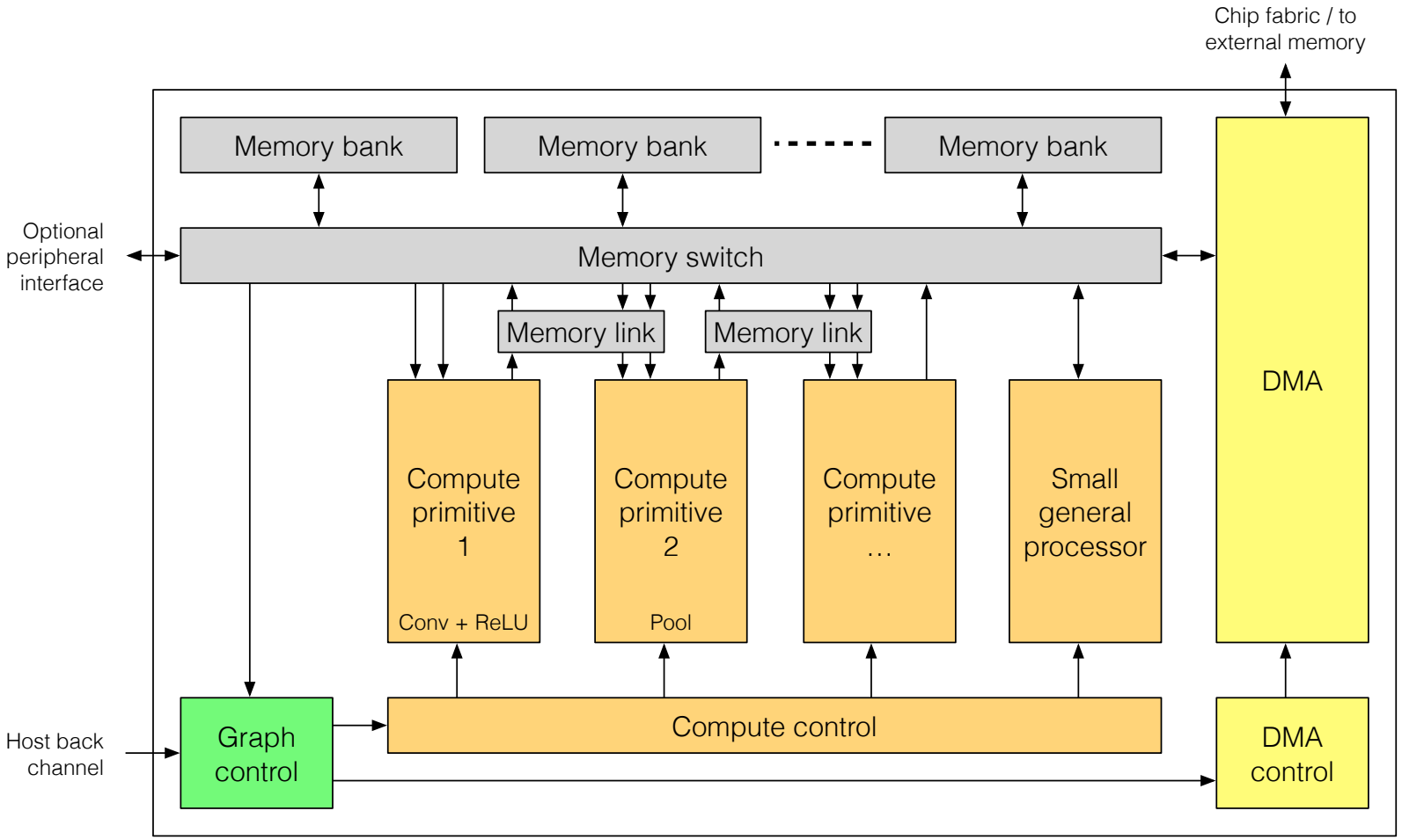
Computational Primitive Candidates

- Matrix multiplication is a given
- Sort seems appropriate
  - Or maybe a general divide and conquer primitive
- After that it's more open …
  - Perhaps an ISP if vision focused
  - Perhaps some if machine for MCTS optimization of RL focused
  - …



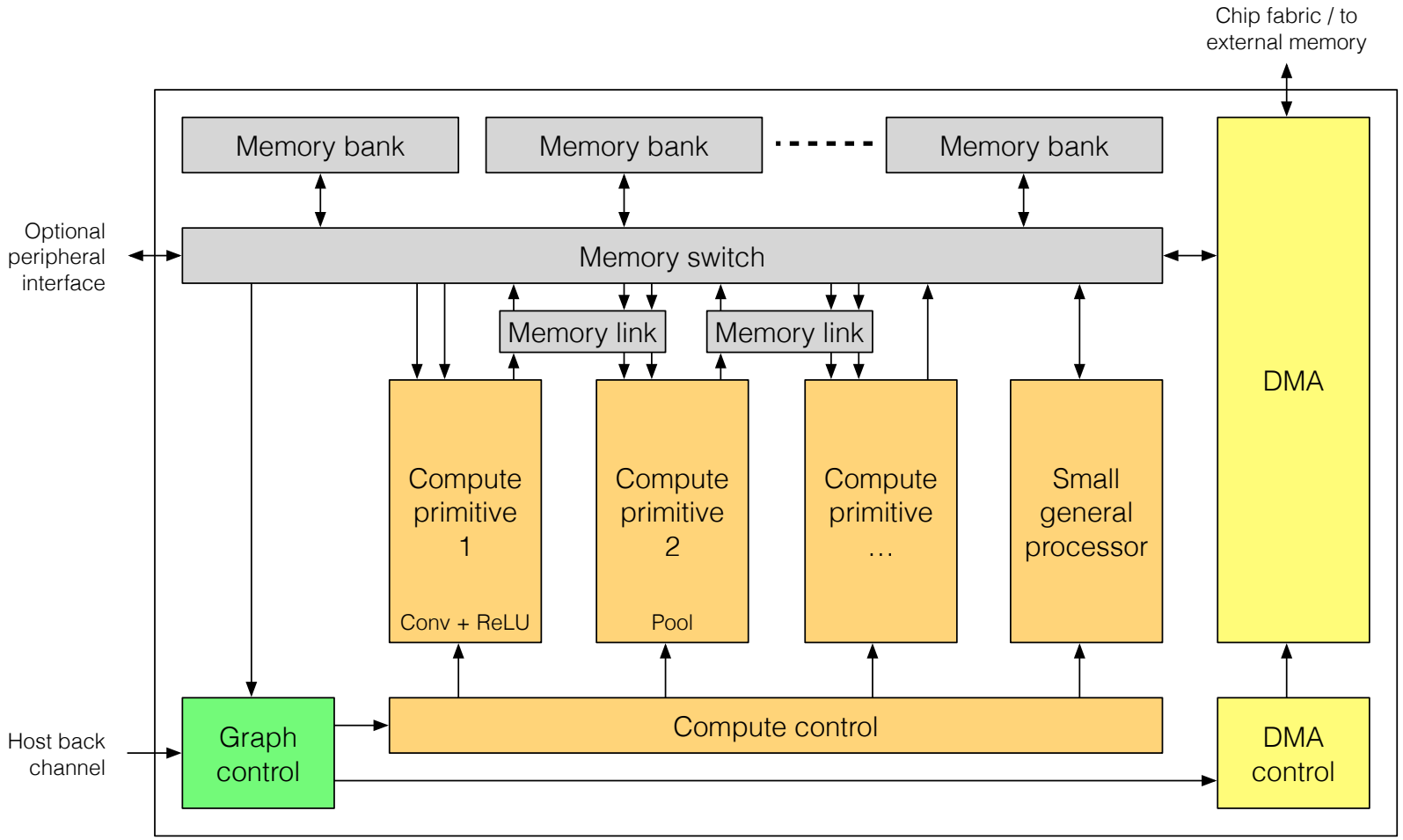
Serial Or Parallel Execution Of Compute

- Serial implies all bandwidth can make each operation go as fast as possible
- Can link operations if the extra memory for the next operation is small relative to the memory for the first operations (e.g., pooling)
  - Even better if it's fully localized (e.g., ReLU)



Serial Or Parallel Execution Of Compute

- When bandwidth limited usually want to use the largest computation primitive possible for the bandwidth constraint
  - For matrix multiplication that means using the largest matrix possible (though remember tiling efficiency)
  - For FFTs that means using the largest FFT possible
- Sometimes the problem size is too small for the extra serial bandwidth and parallel execution is appropriate



Amdahl's Law

- Define
  - p = the fraction of tasks in a program benefiting from acceleration
  - s_{task} = speedup of the task
  - $S_{program}$ = speedup of the whole program
- Amdahl's law
  - $S_{program} = 1 / ((1 - p) + (p/s_{task}))$
- xNNs have many layers
  - CNN style 2D convolution dominates the compute of CNNs and to a 1st order approximation you should do everything you can to make it run as fast as possible (give it most bandwidth)
  - But if you get really really good at making that go fast, it's possible for other operations to start to become a more significant part of the execution time
  - It's why we put control and data movement in parallel
  - It's why you may want to have the option of allocating bandwidth to pool completed output feature maps while the matrix primitive is finishing up other output feature maps

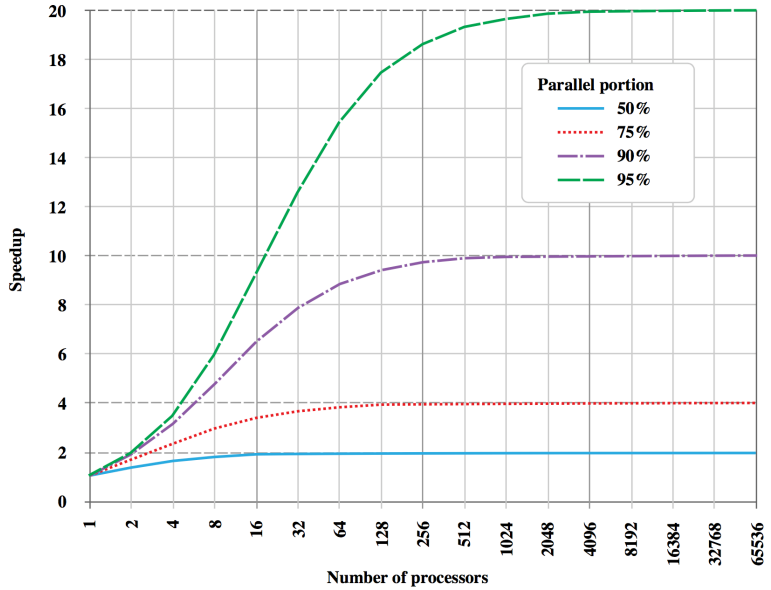
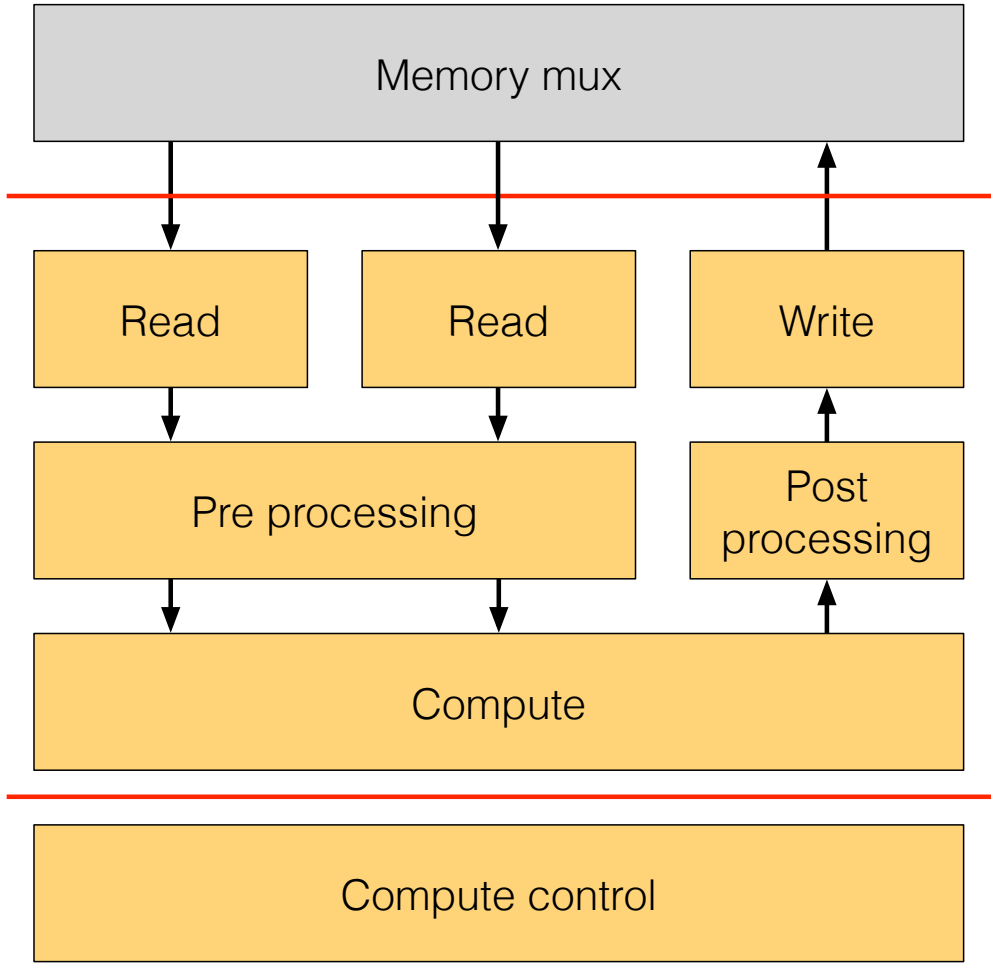


Figure from https://en.wikipedia.org/wiki/Amdahl%27s_law

Computational Primitive Template

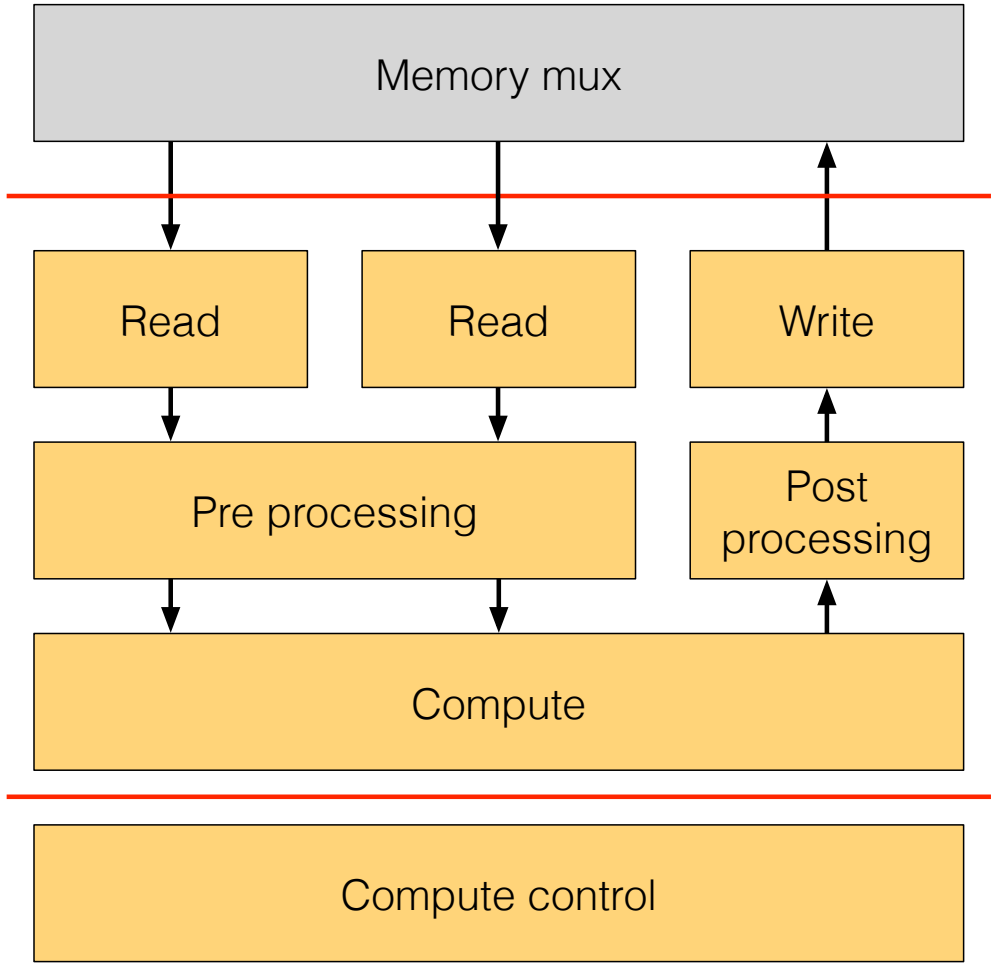
- Data flow
  - Read from local memory
    - Optimal alignment and length
  - Pre processing
    - Simple limited set of data transformations
    - Allows optimized regular compute structure
    - Enable generality within the primitive class
  - Compute
    - Computational primitive computation
  - Post processing
    - Simple limited set of data transformations
    - Allows optimized regular compute structure
    - Enable generality within the primitive class
  - Write to local memory
    - Optimal alignment and length
- Control
  - Initialize state machines
  - Execute (advance) state machines



Matrix Multiplication Primitive

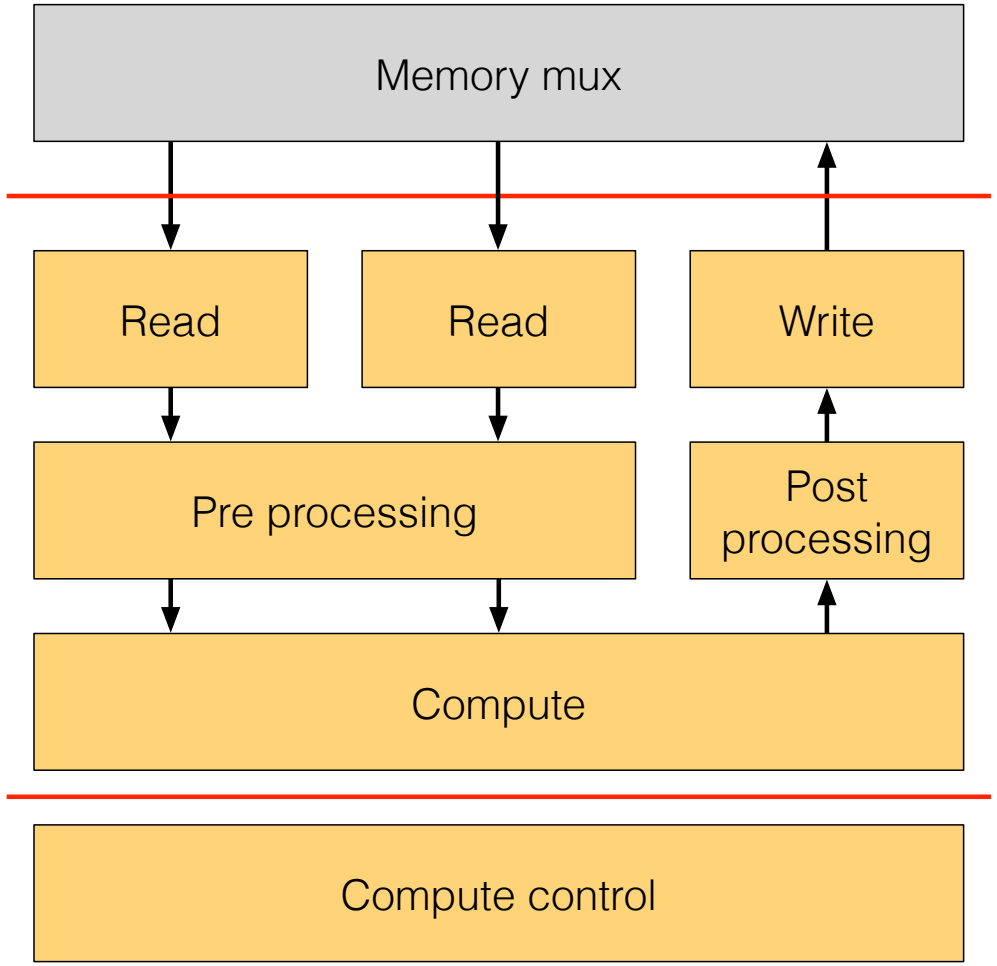
- Basics
  - Read two N x N matrices in N cycles
    - Ex: $H_{tile}(m, k)$ and $X_{tile}(k, n)$
  - Use pre processing to do formatting for the input
    - Ex: $H_{tile}(m, k) \rightarrow A$ and $X_{tile}(k, n) \rightarrow X_{filter}(k, n) \rightarrow B$
  - Compute N x N matrix multiplication in N cycles
    - C += A*B
  - Use post processing to do formatting for the output
    - $C \rightarrow Y_{tile}(m, n)$
  - Write a N x N matrix in N cycles

- Compute to data movement ratio
  - N^3 MACs in N cycles or N^2 MACs / cycle
  - A maximum of 3N pieces of data read or written to local memory per cycle



Matrix Multiplication Primitive

- Strategy
  - Reads and writes are address aligned to maximize bandwidth efficiency
  - Pre and post processing formats data to transform a compatible problem into matrix multiplication and adapt the problem size (e.g., using block matrix multiplication)
  - Compute uses ping pong registers to hold matrices as necessary based on the selected matrix multiplication method to allow continual compute in parallel with data movement
  - Different precisions are supported via scaling the matrix size keeping bandwidth constant
  - For fixed point compute
    - Accumulation is typically at 4x the number of bits of the input operands, scale round clip to bring the output back to the number of bits of the input
    - Supporting 8, 16 and 32 bit precision can be accomplished with the same bandwidth, memory and compute via appropriate multiplier design and using primitive sizes of 1x, 1/2x and 1/4x, respectively

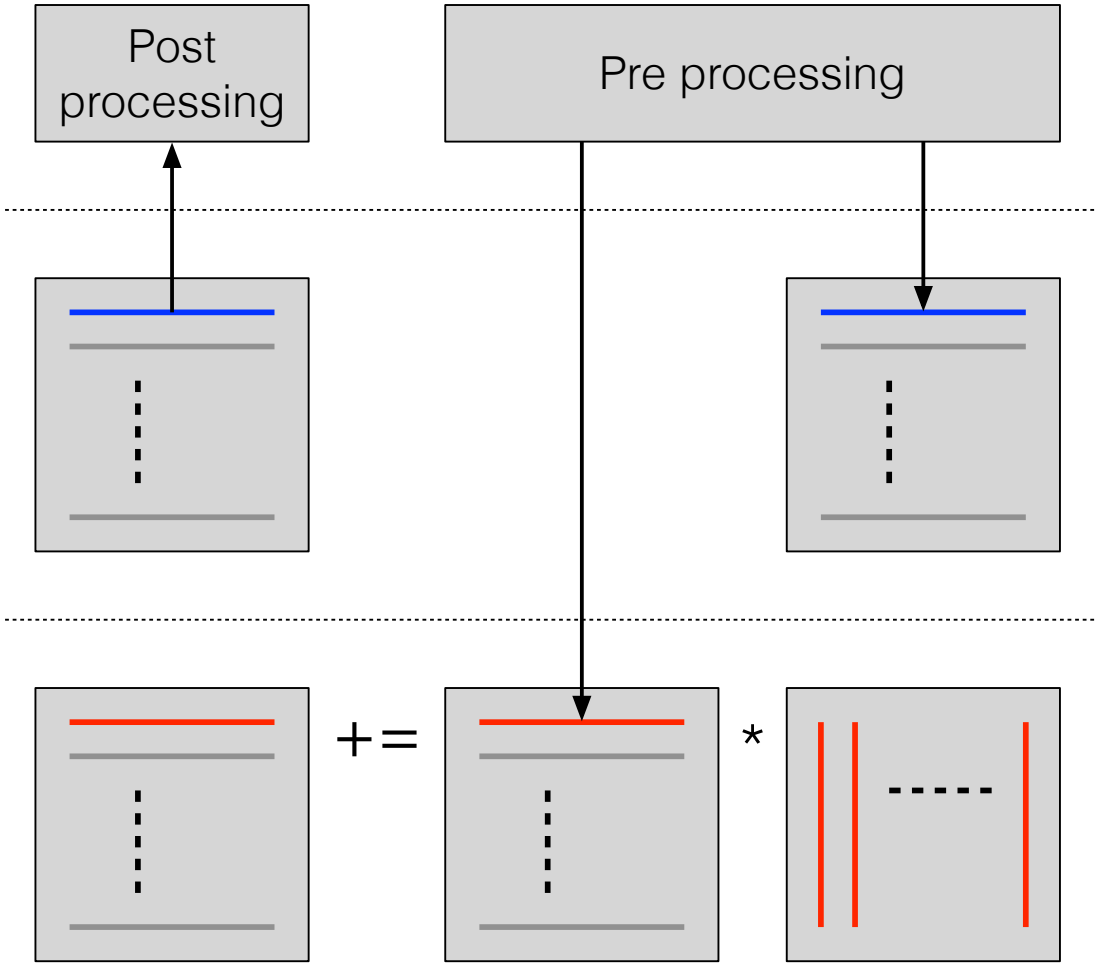


Inner Product Based Matrix Multiplication

- Mathematically it's 3 loops
  - A is in linear order
  - B is needed in transpose order
    - Transpose = bad for typical memory accesses
    - So handle this via background load
- Per cycle transfer data
  - Read A(m, :) and B_{back}(k, :), write C_{back}(m, :)
- Per cycle compute 1 output row C(m, :)

```
C = C0                    // e.g., bias matrix
For m = 0 to M - 1        // m = 0
   For n = 0 to N - 1
      For k = 0 to K - 1
         C(m, n) += A(m, k) B(k, n)
      End
   End
End
```

Parallel

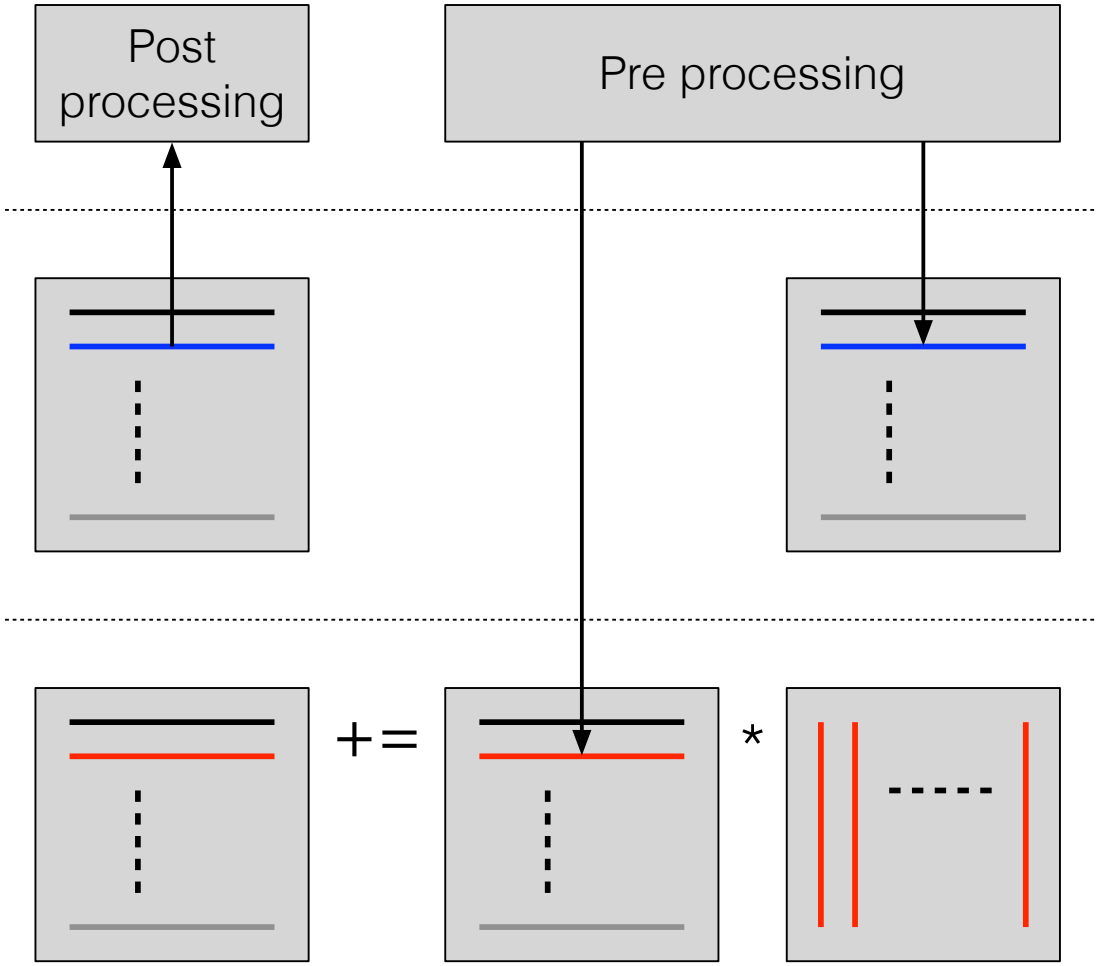


Inner Product Based Matrix Multiplication

- Mathematically it's 3 loops
  - A is in linear order
  - B is needed in transpose order
    - Transpose = bad for typical memory accesses
    - So handle this via background load
- Per cycle transfer data
  - Read A(m, :) and B_{back}(k, :), write C_{back}(m, :)
- Per cycle compute 1 output row C(m, :)

```
C = C0                    // e.g., bias matrix
For m = 0 to M - 1        // m = 1
   For n = 0 to N - 1
      For k = 0 to K - 1
         C(m, n) += A(m, k) B(k, n)
      End
   End
End
```

Parallel

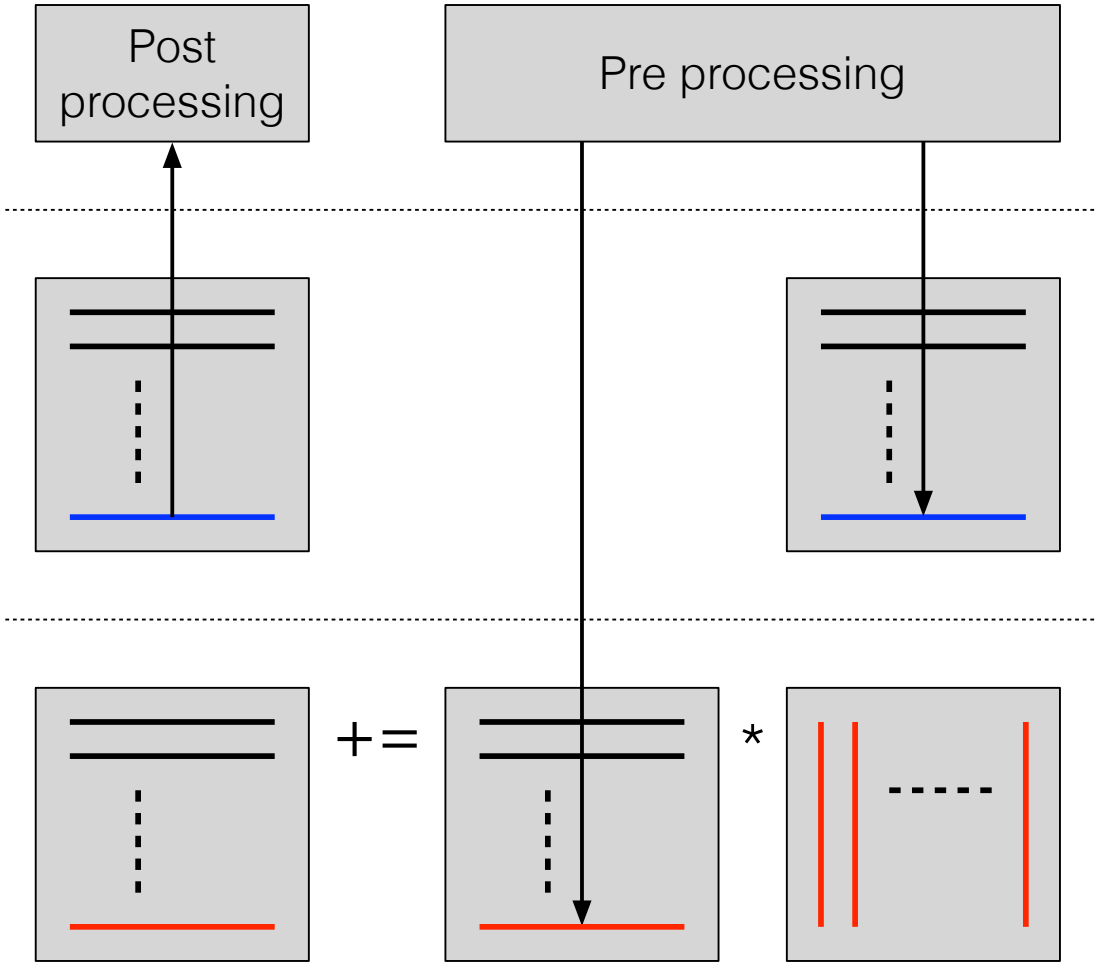


Inner Product Based Matrix Multiplication

- Mathematically it's 3 loops
  - A is in linear order
  - B is needed in transpose order
    - Transpose = bad for typical memory accesses
    - So handle this via background load
- Per cycle transfer data
  - Read A(m, :) and B_{back}(k, :), write C_{back}(m, :)
- Per cycle compute 1 output row C(m, :)

```
C = C0                    // e.g., bias matrix
For m = 0 to M - 1        // m = M - 1
   For n = 0 to N - 1
      For k = 0 to K - 1
         C(m, n) += A(m, k) B(k, n)
      End
   End
End
```

Parallel



Outer Product Based Matrix Multiplication

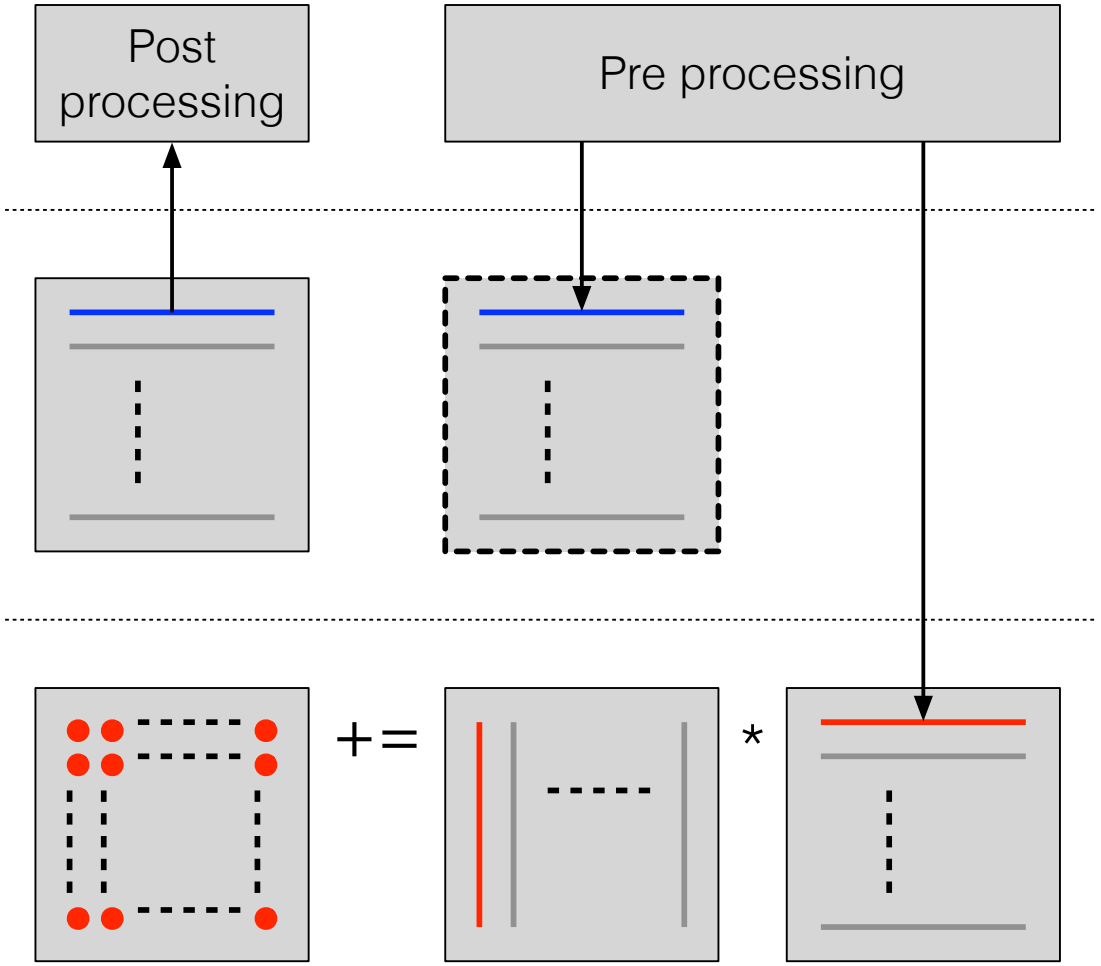
- Mathematically it's 3 loops
  - A is needed in transpose order
    - Transpose = bad for typical memory accesses
    - So handle this via store ordering of A or back
  - B is in linear order

- Per cycle transfer data
  - Read A(:, k) and B_{back}(k, :), write C_{back}(m, :)

- Per cycle compute all partial outputs C(:, :)

```
C = C0                    // e.g., bias matrix
For k = 0 to K - 1        // k = 0
   For m = 0 to M - 1
      For n = 0 to N - 1
         C(m, n) += A(m, k) B(k, n)
      End
   End
End
```

Parallel

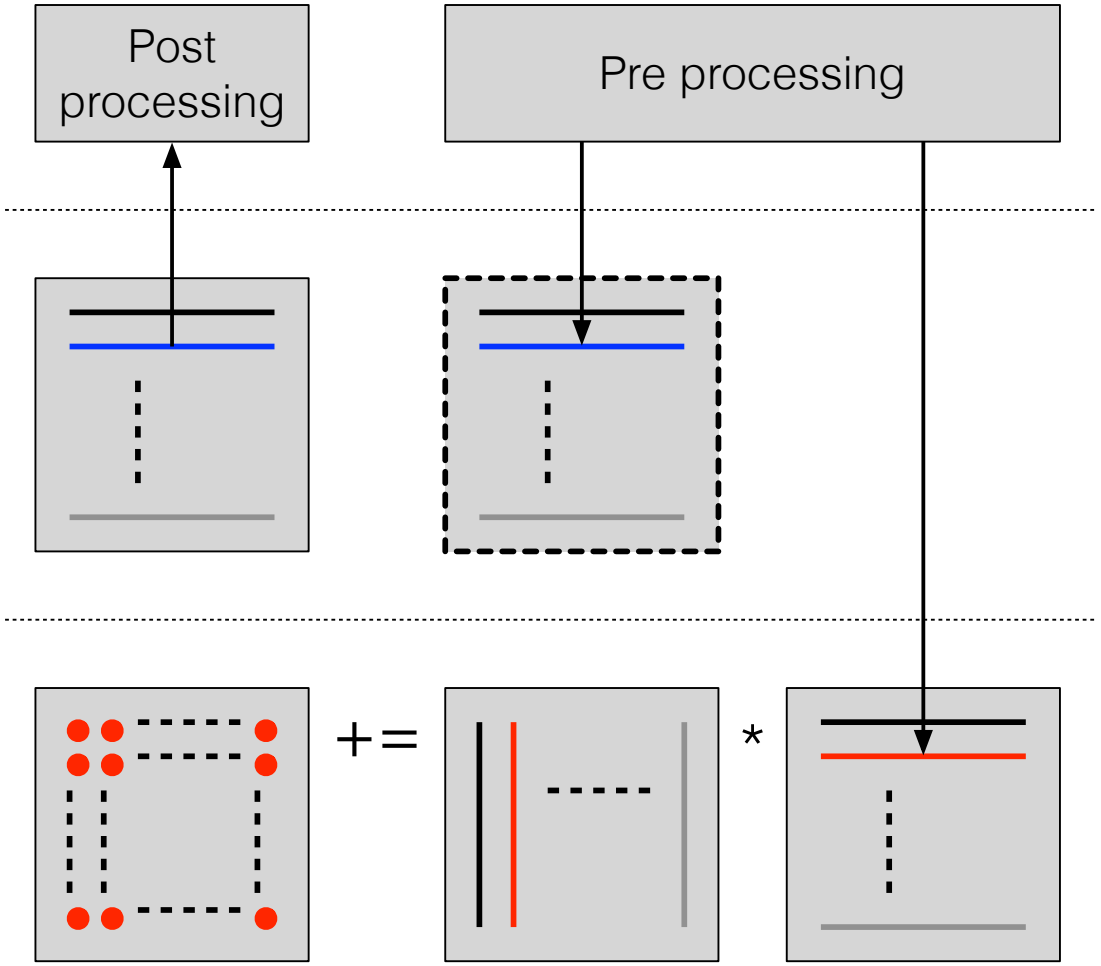


Outer Product Based Matrix Multiplication

- Mathematically it's 3 loops
  - A is needed in transpose order
    - Transpose = bad for typical memory accesses
    - So handle this via store ordering of A or back
  - B is in linear order
- Per cycle transfer data
  - Read A(:, k) and B_{back}(k, :), write C_{back}(m, :)
- Per cycle compute all partial outputs C(:, :)

```
C = C0                    // e.g., bias matrix
For k = 0 to K - 1        // k = 1
   For m = 0 to M - 1
      For n = 0 to N - 1
         C(m, n) += A(m, k) B(k, n)
      End
   End
End
```

Parallel

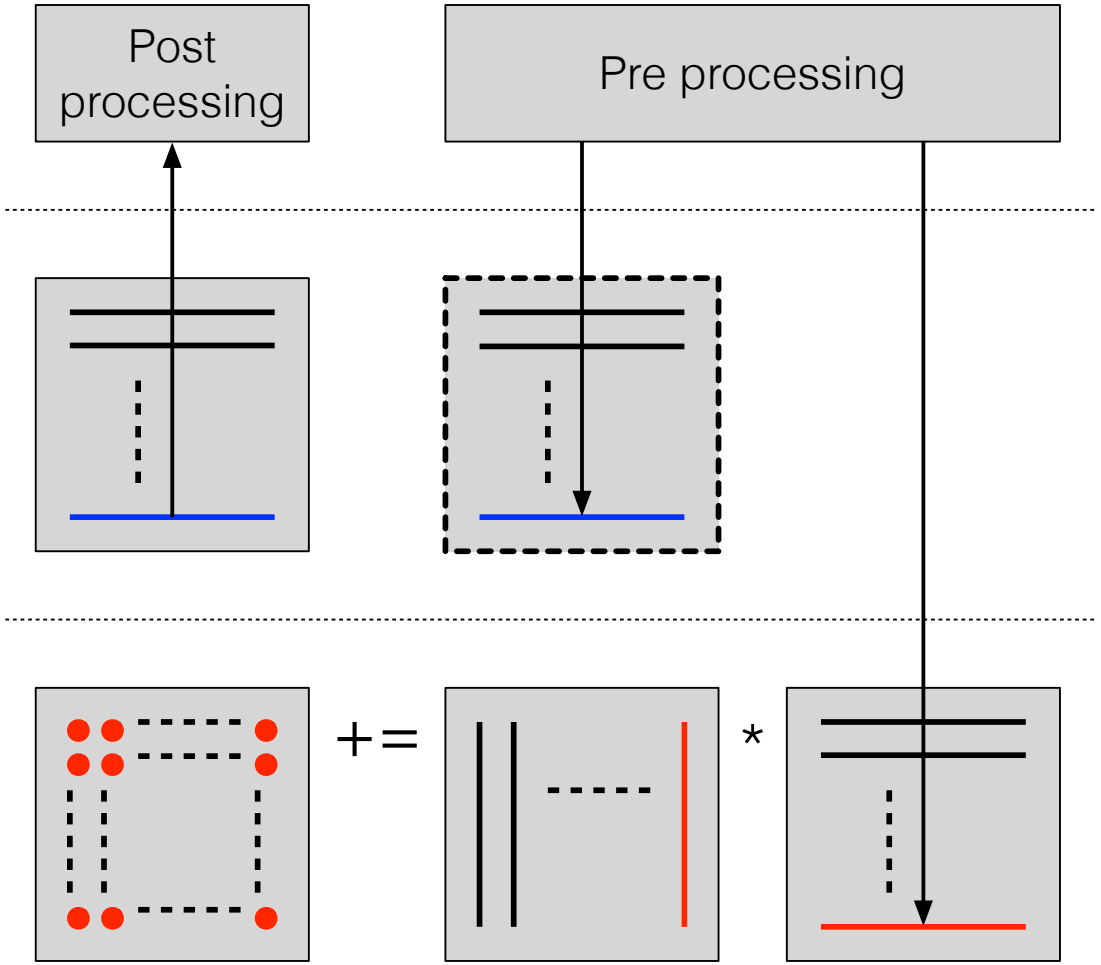


Outer Product Based Matrix Multiplication

- Mathematically it's 3 loops
  - A is needed in transpose order
    - Transpose = bad for typical memory accesses
    - So handle this via store ordering of A or back
  - B is in linear order
- Per cycle transfer data
  - Read A(:, k) and B_{back}(k, :), write C_{back}(m, :)
- Per cycle compute all partial outputs C(:, :)

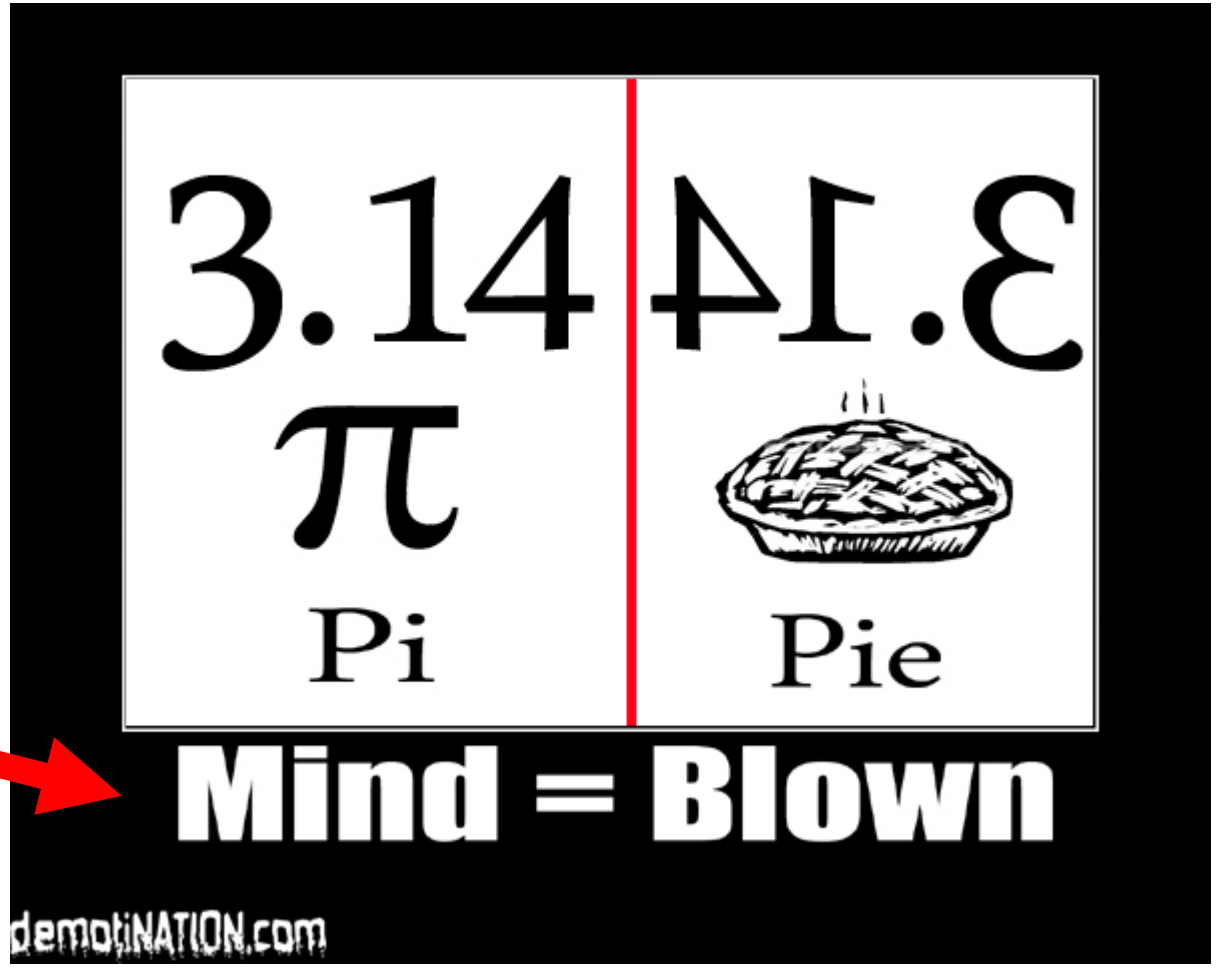
```
C = C0                    // e.g., bias matrix
For k = 0 to K - 1        // k = K - 1
   For m = 0 to M - 1
      For n = 0 to N - 1
         C(m, n) += A(m, k) B(k, n)
      End
   End
End
```

Parallel



Multiplying Matrices With < N^3 MACs

- The above described methods for matrix matrix multiplication require N^3 MACs for M = N = K square matrices
- N^3 is a large number, it would be nice to have a smaller exponent
- It's possible to multiply 2 matrices with less than N^3 MACs
- But there are tradeoff of additions, sequential operations, memory movement, ...
  - In practice this makes it somewhat questionable for many cases when applied to optimal architectures



Generally How To Reduce The Cost Of A Calculation

- Need the operations that make up the calculation to have different costs so you can tradeoff 1 for the other
  - Example: multiplies cost more than adds
- Generally need to create intermediate terms that will be re used
  - This is used in many places, either implicitly or explicitly
- Sometimes the intermediate term strategy is recursively applied

Figure from https://www.giantbomb.com/articles/hey-its-a-premium-membership-sale/1100-5747/

Example Applications Of This Strategy By Gauss

- Computing FFTs via a power of 2 matrix decomposition

- Multiplying 2 complex numbers
  - Standard = 4 real multiplies and 2 real adds

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc)$$

  - Gauss trick = 3 real multiplies and 5 real adds
    - Partial terms (note sequential dependency)

$$u = ac,\ v = bd,\ x = a + b,\ y = c + d,\ z = xy$$

    - Final result

$$ac - bd = u - v$$
$$ad + bc = z - u - v$$

Strassen's Algorithm For Matrix Multiplication

- Strassen's algorithm for reducing the number of multiplications in matrix matrix multiplication
  - Multiplies two block 2 x 2 matrices using 7 block multiplies vs the standard of 8
  - Recursively apply
  - Reduces multiplications to ~ $O(N^{log\_2(7)}) = O(N^{2.81})$

- Strassen mechanics for the multiplication of N x N matrices C = A B

$$\begin{bmatrix} C(0,0) & C(0,1) \\ C(1,0) & C(1,1) \end{bmatrix} = \begin{bmatrix} A(0,0) & A(0,1) \\ A(1,0) & A(1,1) \end{bmatrix} \begin{bmatrix} B(0,0) & B(0,1) \\ B(1,0) & B(1,1) \end{bmatrix}$$

Strassen's Algorithm For Matrix Multiplication

- Define the following 7 partial terms

	New "A"s size N/2	New "B"s size N/2	
S_1	(A(0,0) + A(1,1))	(B(0,0) + B(1,1))	// 1 mult, 2 add
S_2	(A(1,0) + A(1,1))	(B(0,0))	// 1 mult, 1 add
S_3	(A(0,0))	(B(0,1) – B(1,1))	// 1 mult, 1 add
S_4	(A(1,1))	(B(1,0) – B(0,0))	// 1 mult, 1 add
S_5	(A(0,0) + A(0,1))	(B(1,1))	// 1 mult, 1 add
S_6	(A(1,0) – A(0,0))	(B(0,0) + B(0,1)))	// 1 mult, 2 add
S_7	(A(0,1) – A(1,1))	(B(1,0) + B(1,1))	// 1 mult, 2 add

Strassen's Algorithm For Matrix Multiplication

- Note that

$C(0,0) = S_1 + S_4 - S_5 + S_7$	// 0 mult, 3 add
$C(0,1) = S_3 + S_5$	// 0 mult, 1 add
$C(1,0) = S_2 + S_4$	// 0 mult, 1 add
$C(1,1) = S_1 - S_2 + S_3 + S_6$	// 0 mult, 3 add
Strassen total	// 7 mult, 18 add
Traditional total	// 8 mult, 4 add

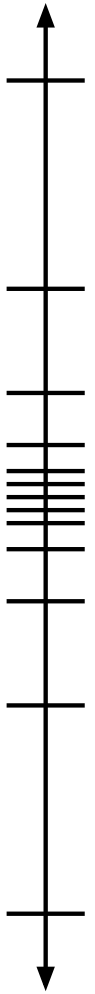
- Now recursively apply the same decomposition to each of the 7 new "A" and "B" matrix pairs

Winograd Style Convolution Algorithm

- Related side note
- Similar to fast algorithms for multiplying complex numbers, FFTs, matrix multiplication, ... Winograd figured out a fast algorithm for convolution
- A modified variant has been applied to CNNs, is used in some libraries and is appropriate for some architectures
- Fast algorithms for convolutional neural networks
  - https://arxiv.org/abs/1509.09308

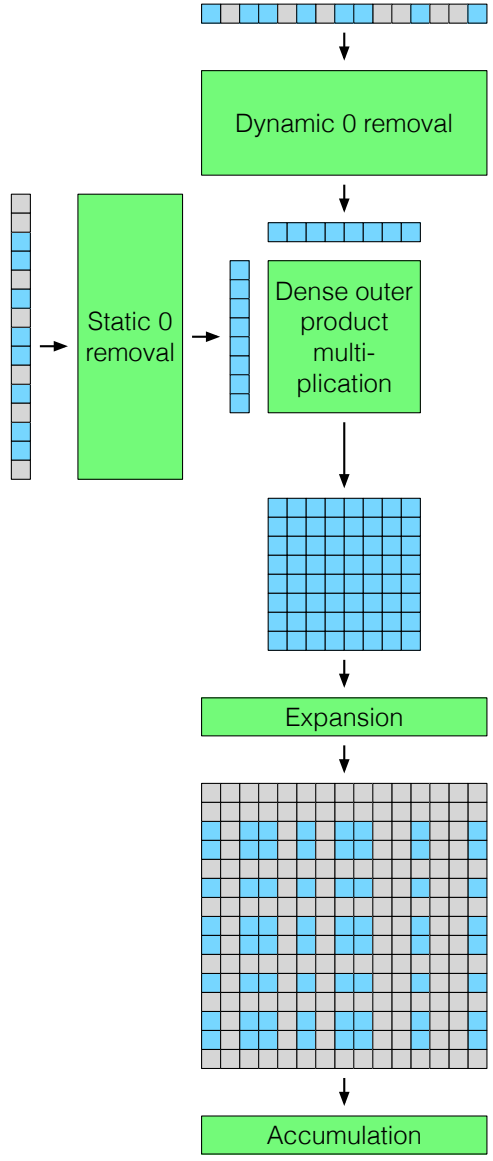
Input Power Of 2 Matrix Multiplication

- Fixed point methods described on earlier slides used quantization schemes with a uniform spacing between levels
- Possibility
  - Non uniform quantization of multiplicative filter coefficients (leave biases as arbitrary 32 bit fixed point values)
  - Choose non uniform levels to be powers of 2, include 0 too
  - Why: multiplication of the filter coefficient with the feature map becomes a simple shift and add
  - Much less complexity than normal multiplication
- A challenge of this is the distance between the larger values
  - Definitely requires some additional re training
- ShiftCNN: generalized low-precision architecture for inference of convolutional neural networks
  - https://arxiv.org/abs/1706.02393



Sparse Matrix Multiplication

- Matrix multiplication methods on previous slides were designed for dense matrix multiplication
- However
  - Around 50% of feature map values are 0 (dynamic)
  - It's also possible to force sparsity in filter coefficients (static)
    - Possibly with some implications to accuracy
    - Not fully clear if it's better than having a smaller dense set of filter coefficients
    - But regardless, it's a thing people can do
- It's possible to take advantage of this to improve matrix multiplication
  - Similar to Sparse BLAS to BLAS, the higher the level of sparsity the higher the potential advantage
  - Traditionally in high performance compute things are dense or very sparse (e.g., 1/100); in xNNs sparsity can be between these points which leads to different methods for index tracking
- Can implement via static / dynamic compression around an outer product based dense matrix multiplication primitive

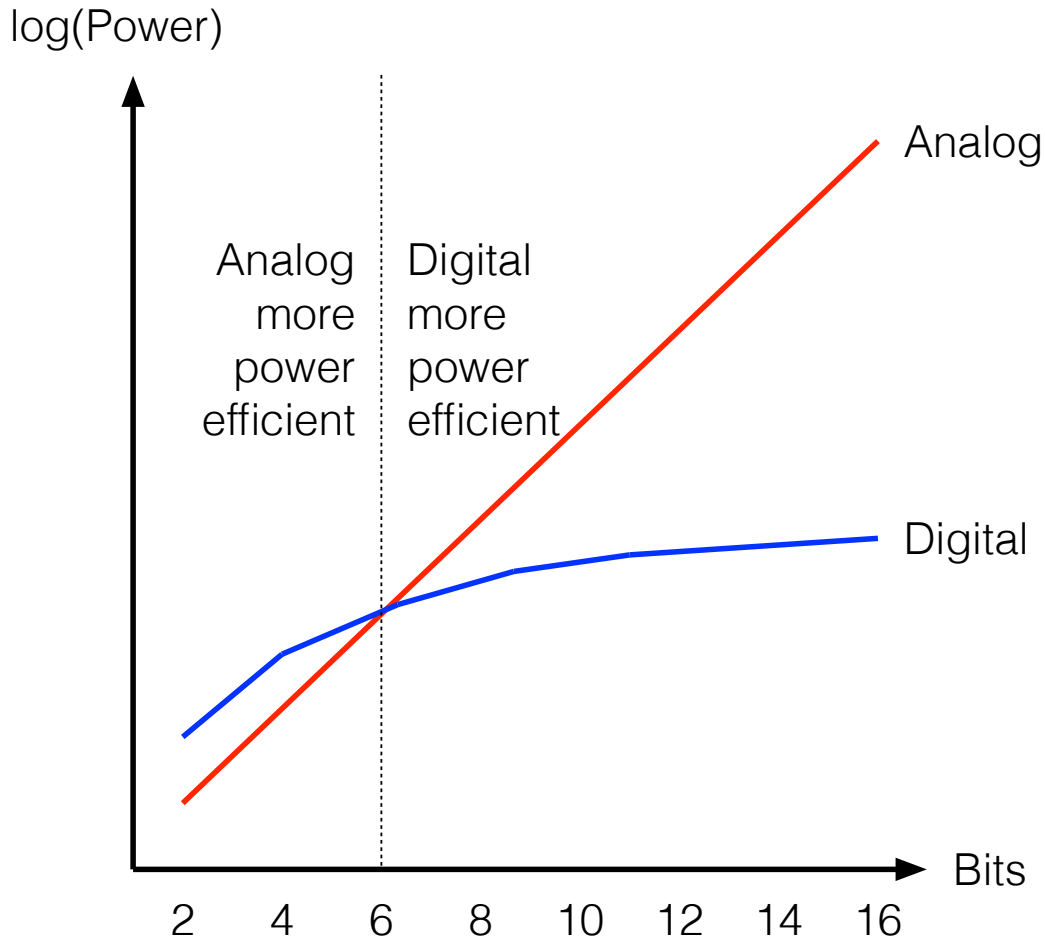


Analog Matrix Multiplication

The key is using either technology to build a matrix multiplier with appropriate data transformation; an Eric Vittoz style thought experiment implies that if power efficiency is the top priority, ~ 1, 2 and 4 bit precision ops go in analog and ~ 8, 16 and 32 bit ops go in digital

- Digital scaling
  - Addition, comparisons, memory and data movement are linear in the number of bits
  - Multiplication is square in the number of bits
  - So digital scales somewhere in between linear and quadratic in the number of bits

- Analog scaling
  - For architectures where bits are in amplitude
  - Adding an extra bit at the same slicer separation requires doubling the power rail range
  - Doubling the power rail range leads to ~ 4x the power
  - Maybe for architecture with 2 levels that increase frequency the answer is more linear (should verify)
  - But frequency increase hits exponential wall of power and eventually need to go back to adding more levels



Bias Addition

- Data type
  - Same data type as filter coefficients for floating point
  - ~ 4x bits as filter coefficients for fixed point

- Operation
  - Bias matrix has rank 1 outer product structure
  - Can take advantage of this for all multiplication methods for loading the matrix with a vector and replication
  - Can also implement as part of matrix multiplication via the affine to linear conversion via matrix augmentation

Post Processing

- The most convenient candidates for post processing are memory localized and don't have significant memory dependencies across tiles

- Elementwise nonlinearities
  - ReLU (fully localized) is very cheap / simple; definite include option for this
  - Others are likely included depending on the connection to / performance of the small general processor

- Range tracking for fixed point
  - The compute scale is either determined statically or dynamically
  - To simplify dynamically setting the compute scale, the max / min range of the accumulators can be tracked

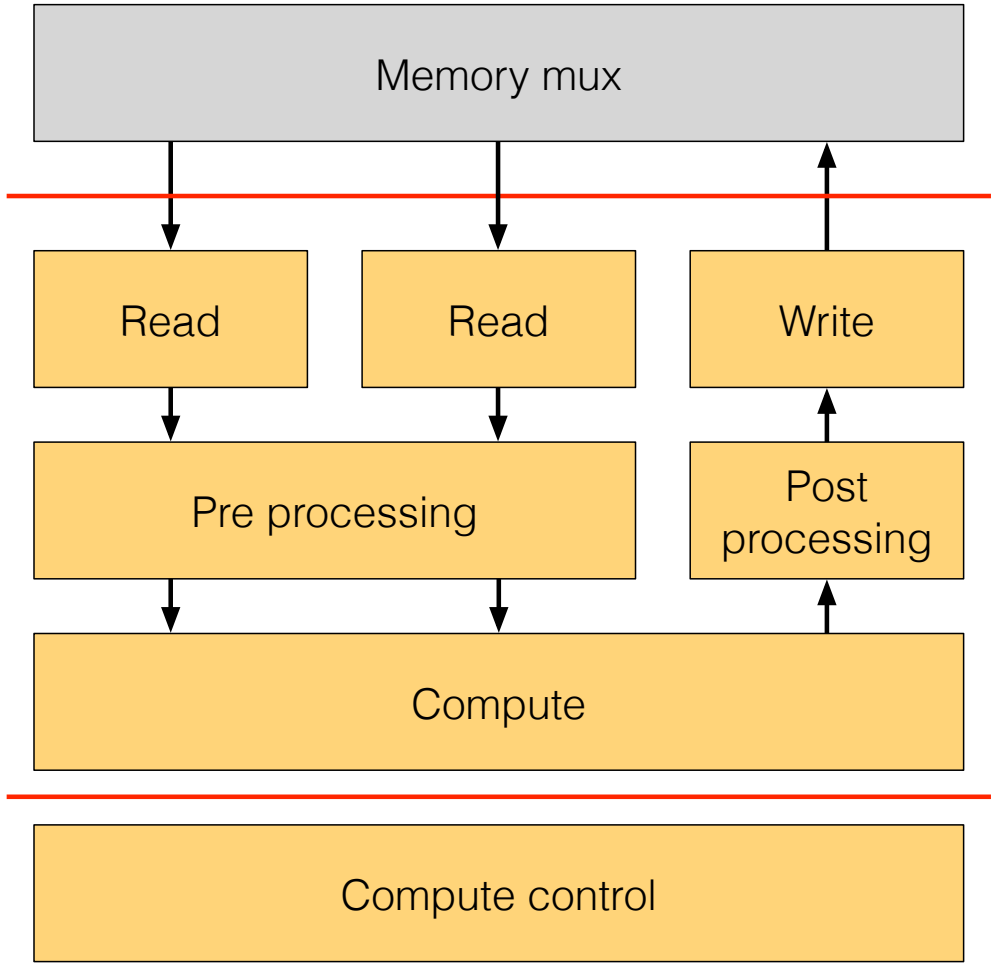
Matrix Multiplication Primitive Uses

Enabled via appropriate pre and post processing

- Dense linear algebra
  - Matrix multiplication and addition
  - Matrix pointwise multiplication and addition
- Convolution
  - CNN style 2D convolution + ReLU
  - Standard 1D and 2D convolution
- Transforms
  - DFT, FFT and DCT
- Some things you don’t expect
  - Clamp
  - Transcendental functions (via series approximations)

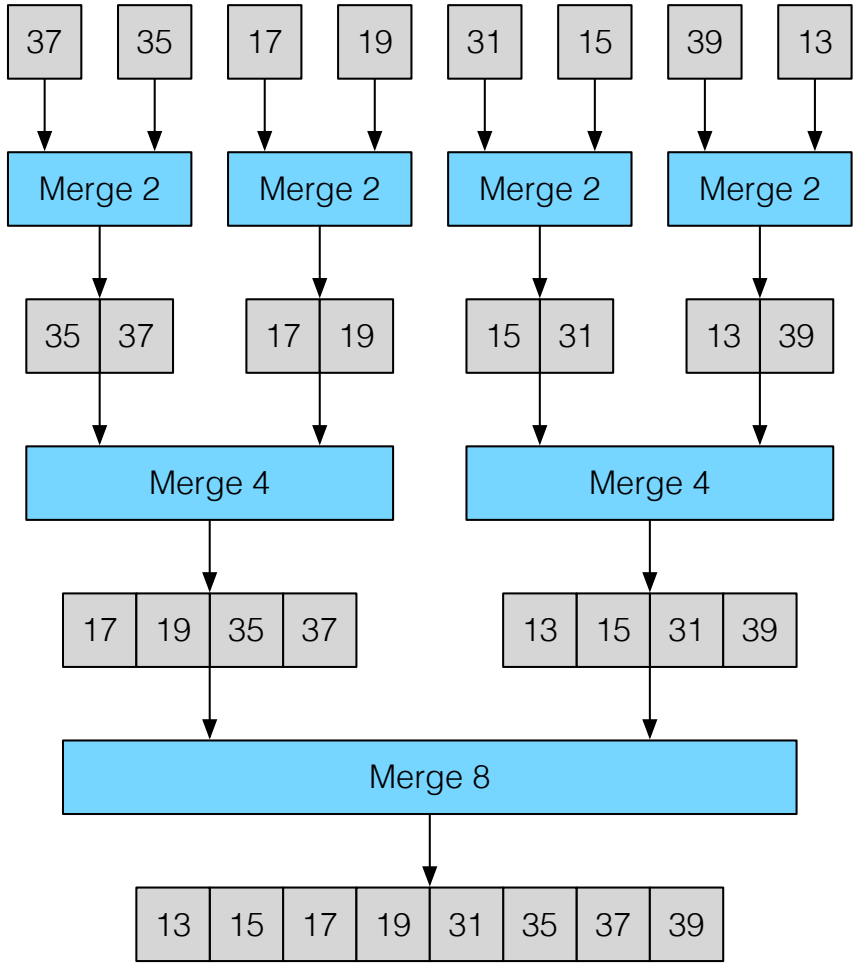
Sort Primitive

- Basics
  - Read two N x 1 vectors each cycle
    - Ex: two new rows for 3 x 3 / 2 max pooling
  - Use pre processing to do formatting for the input
    - Ex: align 3 to 4 via repetition for common sorting
  - Merge sort for a specified number number of stage
    - Ex: two to sort 4 items
  - Use post processing to do formatting for the output
    - Ex: accumulator comparison to sort across rows
    - Ex: keeping max out of each 4 columns
  - Write a N x 1 vector each cycle
    - Ex: maxes in 1 cycle

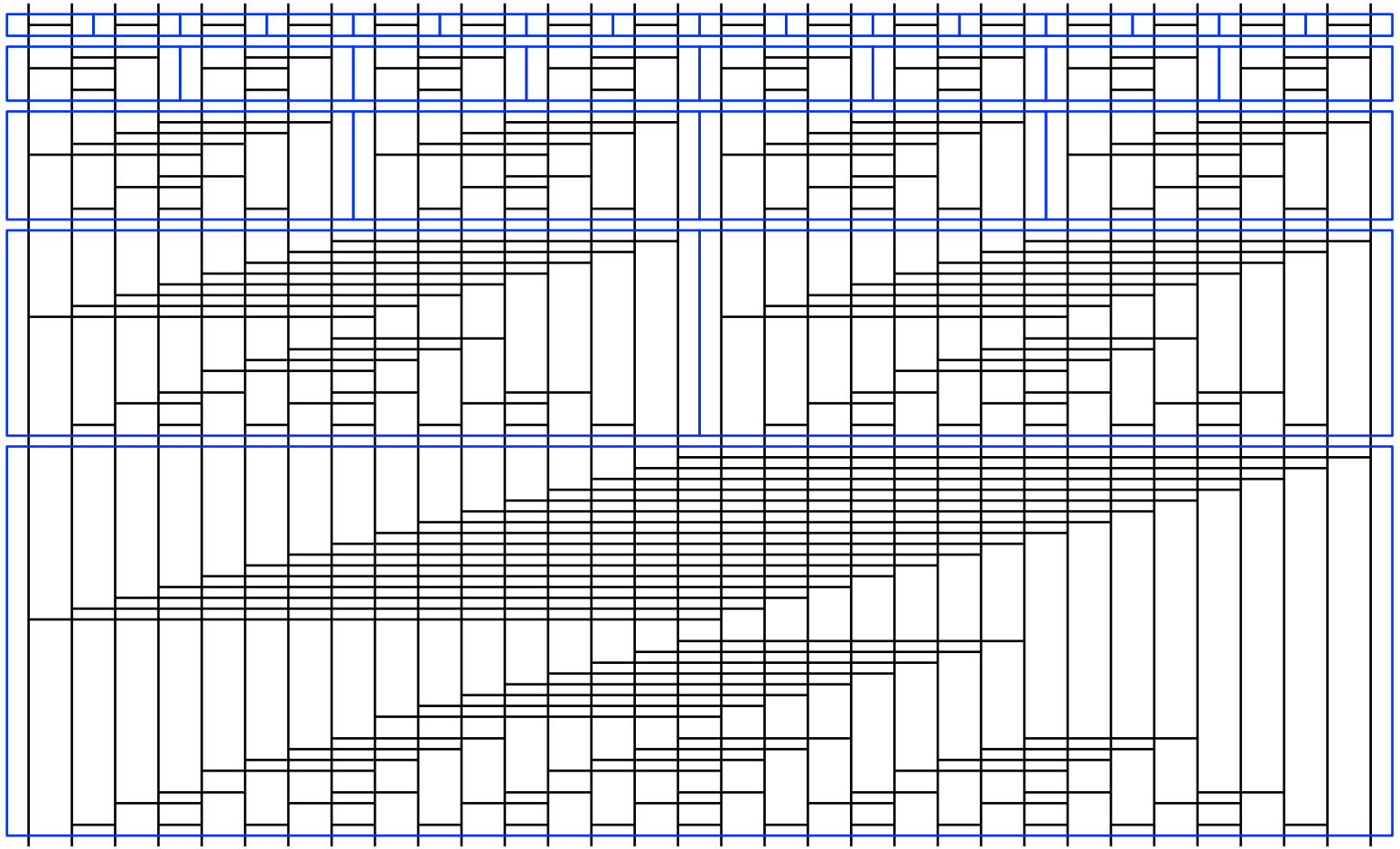


Sort Primitive

- Notes
  - Performance of proposed sorting algorithm is matched to data movement limit
  - Supporting 8, 16 and 32 bit precision can be accomplished with the same bandwidth, memory and compute via appropriate comparator design and using primitive sizes of 1x, 1/2x and 1/4x, respectively



Parallel Merge Sort Style Sorting Network



16x merge 2
(16 comparisons, 1 substage)

8x merge 4
(24 comparisons, 2 substages)

4x merge 8
(36 comparisons, 3 substages)

2x merge 16
(50 comparisons, 4 substages)

1x merge 32
(65 comparisons, 5 substages)

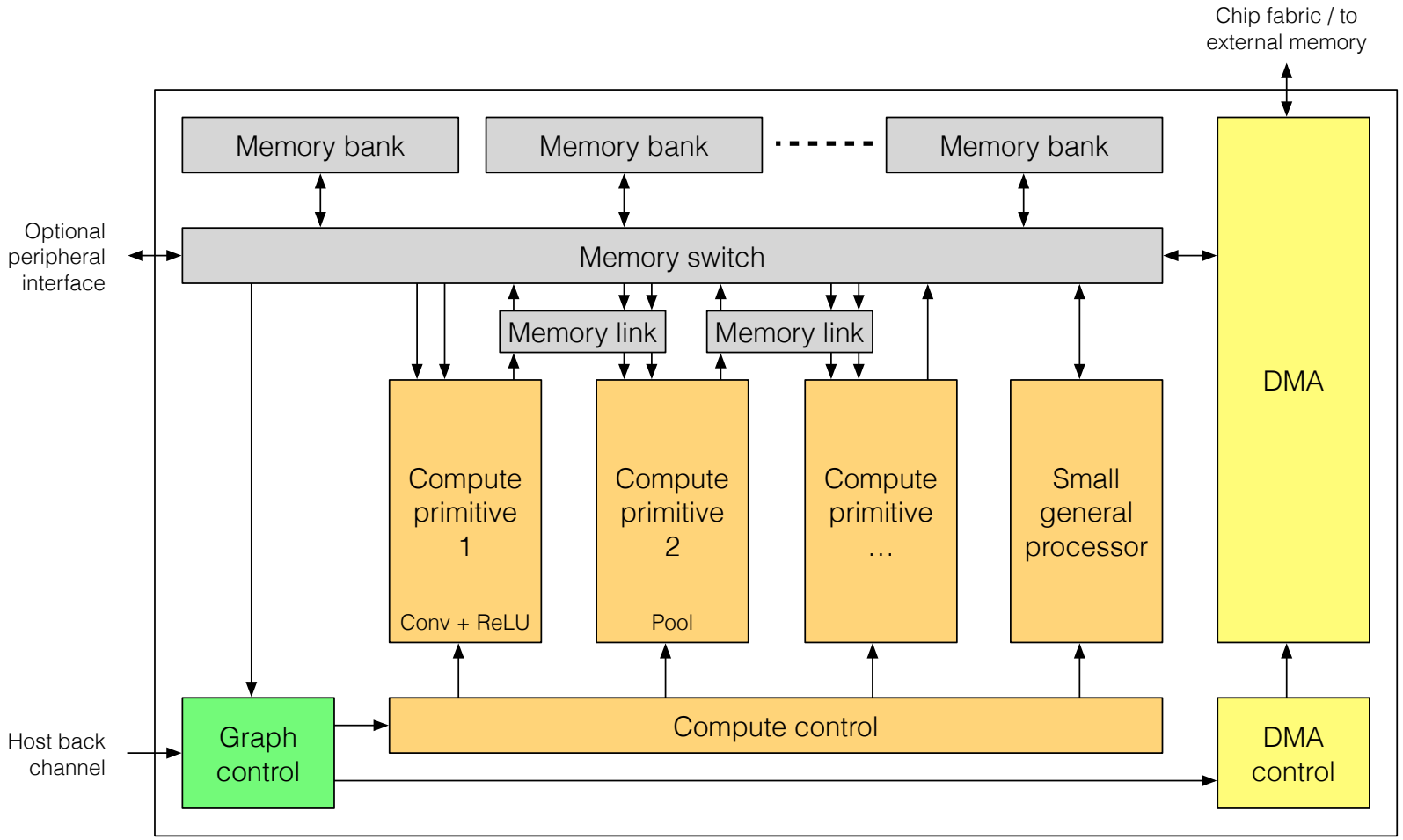
Sort Primitive Uses

Enabled via appropriate pre and post processing

- Sorting
  - Full, partial
  - 1 vector with another
  - 1D and 2D
- Min and max
- Rank order filter
  - Median and arbitrary
- Pooling
  - Max

Small General Processor

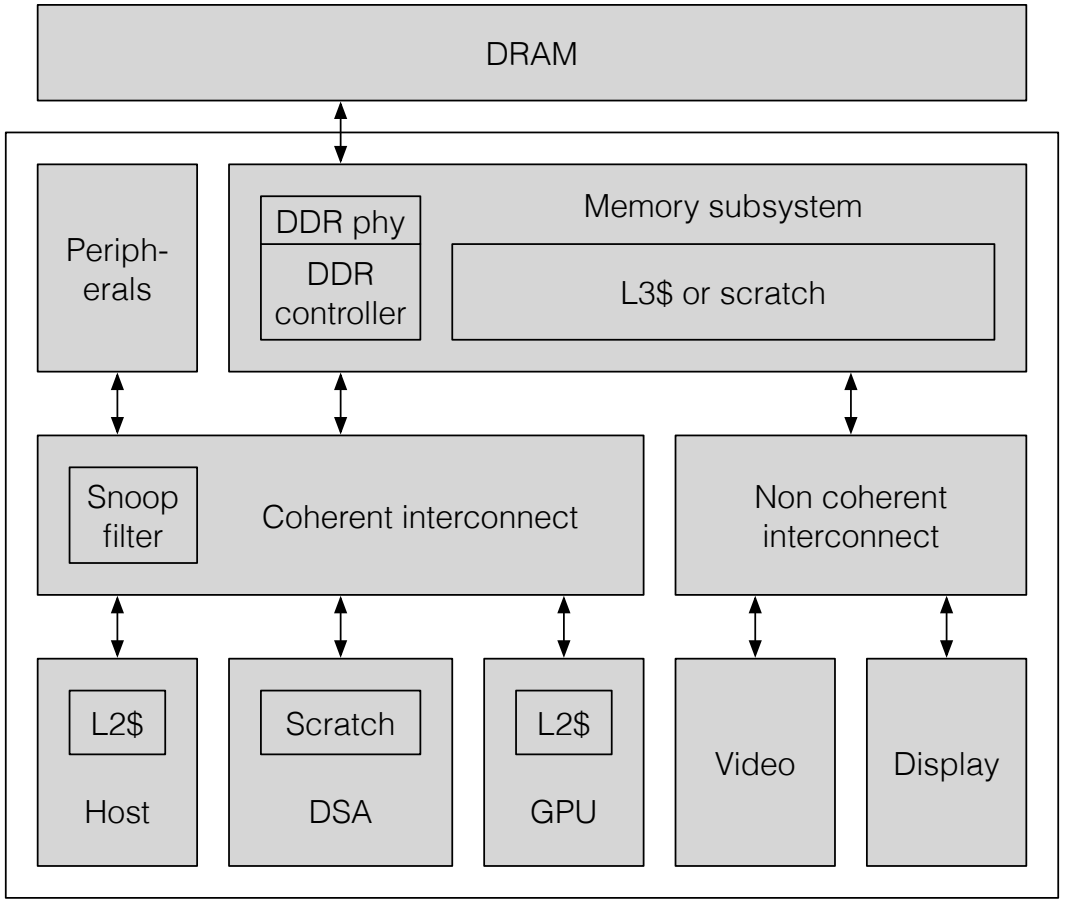
- Cleanup for anything that can't be mapped to 1 of the current computational primitives
- Use for generality
- Use for future proofing



Hardware – Network Architecture

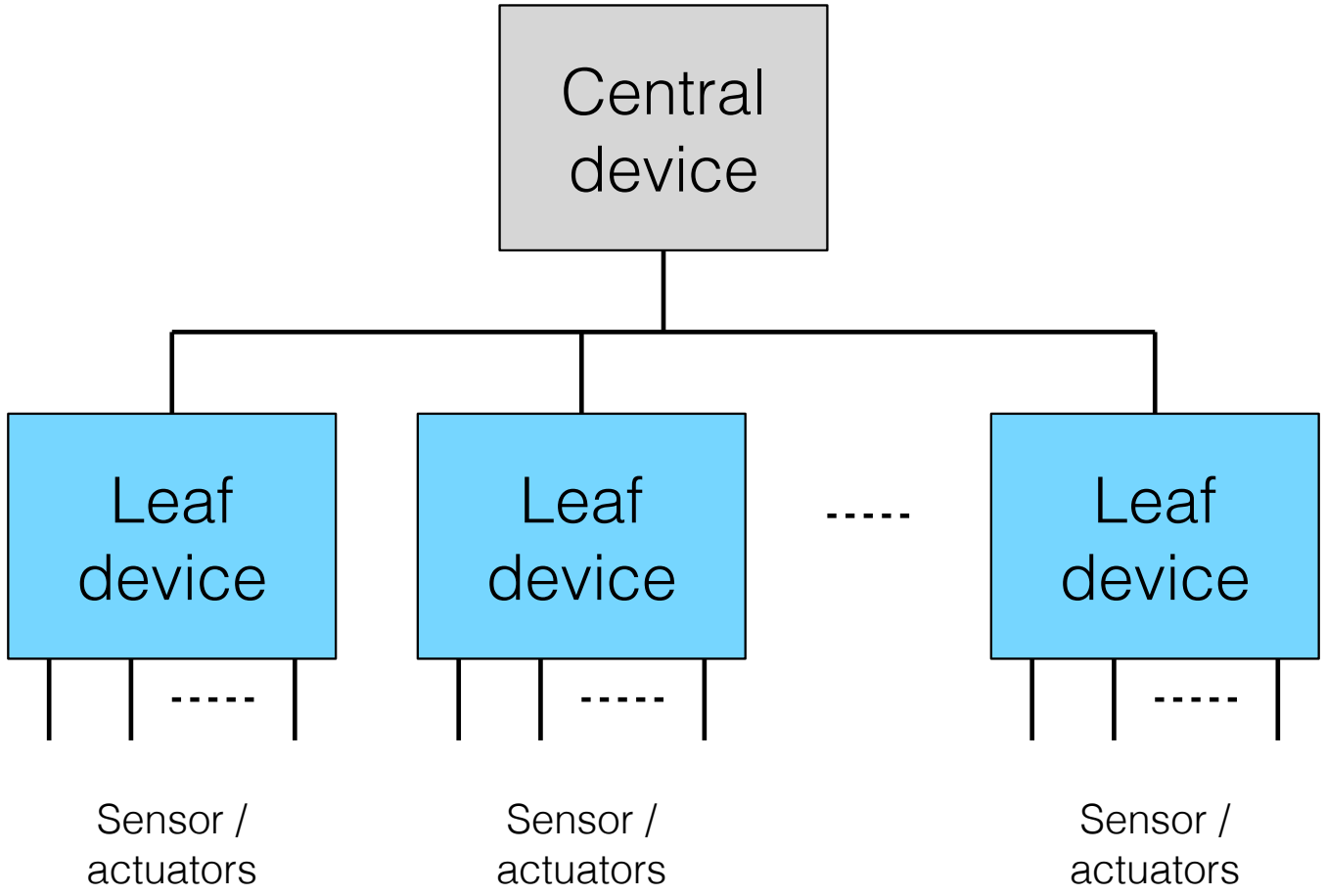
Individual Device

- This is what we've been discussing so far
  - Consider scaling up or down based on the particular application space



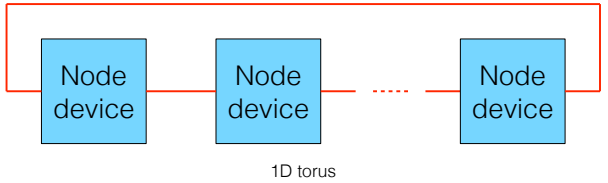
Multiple Devices In A Tree Configuration

- Example sensor / actuator use
  - Disjoint sensor processing via xNNs in edge devices
  - Higher level fusion in a central device leading to decisions; possibly xNN based, possibly shallow combination
  - Disjoint actuation in edge devices based on central device decisions
- Example training use
  - Synchronous training

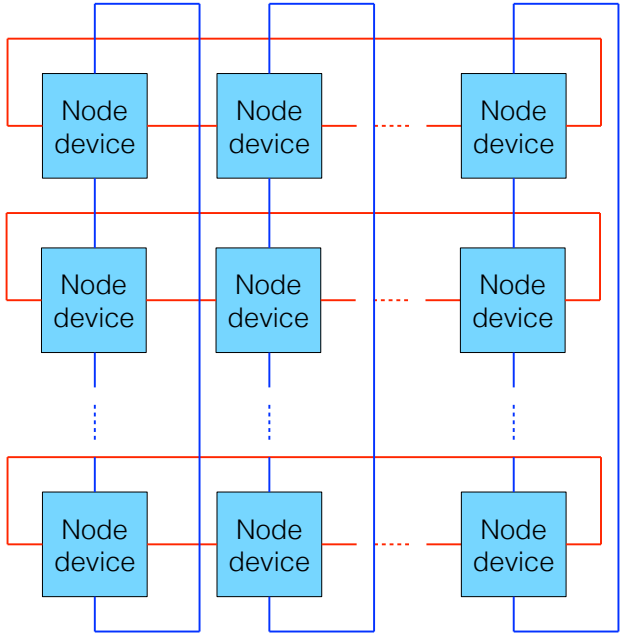


Multiple Devices In A Torus Configuration

- Devices with 2N network connections in a ND torus
- Example use
  - Larger compute problems
- Note
  - Beyond tree and torus configurations, many other configurations are possible



1D torus



2D torus

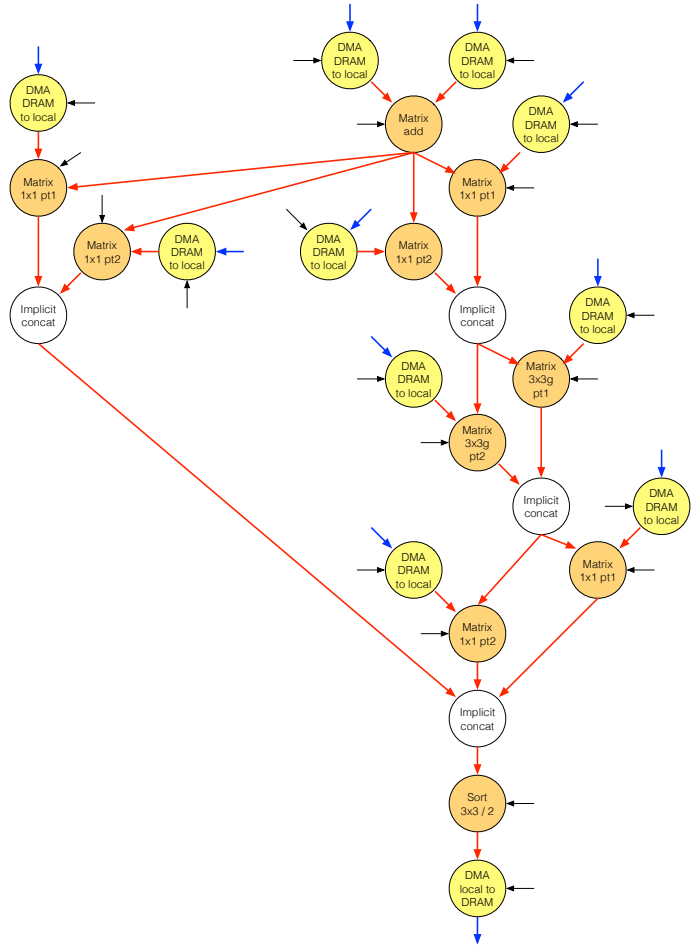
Performance

Predicting And Benchmarking

- Performance prediction == **estimating** network / software implementation / hardware implementation performance
- Benchmarking == **measuring** network / software implementation / hardware implementation performance
- Architecture decisions affect performance
  - Not an especially profound statement
  - But perhaps slightly more subtle, memory, data movement and compute choices mean certain systems can perform better / more efficiently in some cases and others in other cases

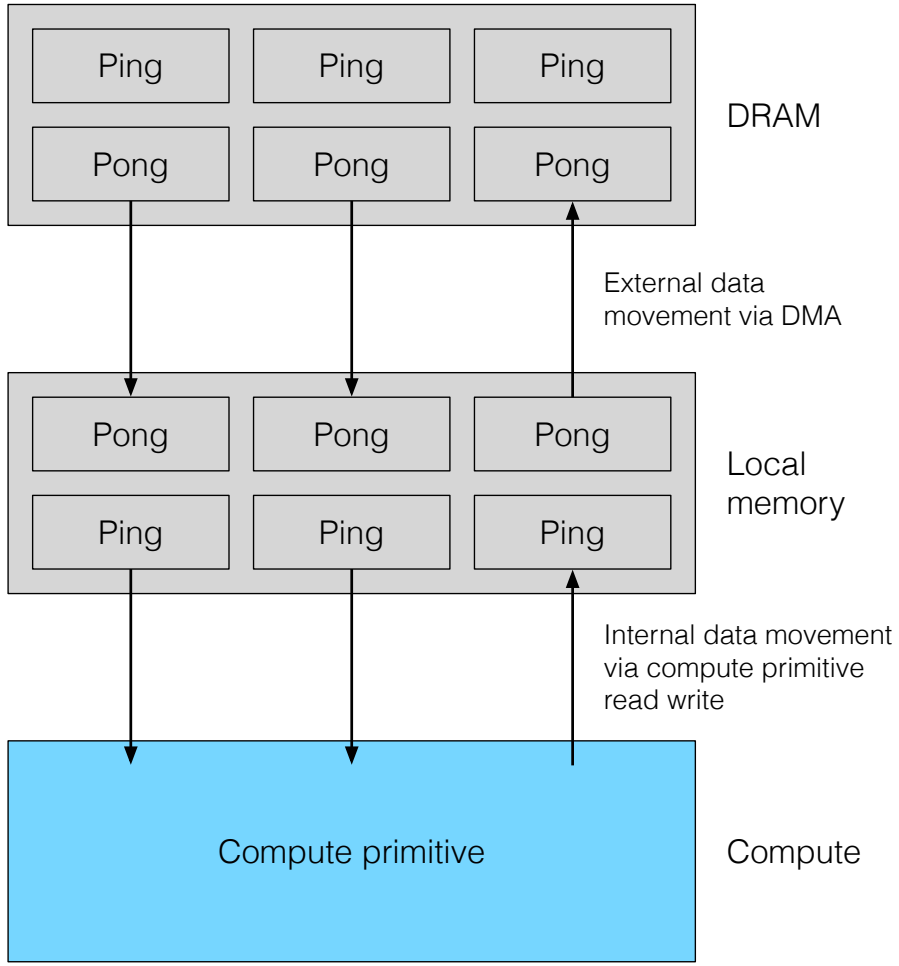
Predicting Performance

- Known from the low level graph
  - Exact order, parallelism and time for each node



An Approximation To Give Intuition

- Choose memory locations
  - Feature maps in local memory if they fit, external memory otherwise
  - Filter coefficients in external memory
- Calculate data movement time
  - Input feature maps, filter coefficients, output feature maps
- Calculate compute time
  - Matrix and vector operations
- Bound total time per layer
  - Serial bound: data movement + compute
  - Parallel bound: max(data movement, compute)
- Bound total time for the network
  - Serial bound: sum of serial time for each layer
  - Parallel bound: sum of parallel time for each layer



Benchmarking Example

- MLPerf (machine learning performance benchmarking suite)
  - Links
    - https://mlperf.org
    - https://mlperf.org/assets/static/media/MLPerf-User-Guide.pdf
    - https://github.com/mlperf/reference
    - https://github.com/mlperf/submissions
  - Tests
    - Image_classification - Resnet-50 v1 applied to Imagenet
    - Object_detection - Mask R-CNN applied to COCO
    - Speech_recognition - DeepSpeech2 applied to Librispeech
    - Translation - Transformer applied to WMT English-German
    - Recommendation - Neural Collaborative Filtering applied to MovieLens 20 Million (ml-20m)
    - Sentiment_analysis - Seq-CNN applied to IMDB dataset
    - Reinforcement - Mini-go applied to predicting pro game moves

Benchmarking Example

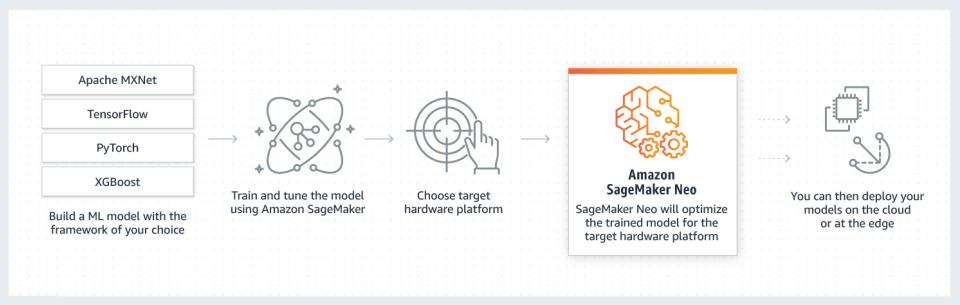
- Stanford data analytics for what's next (DAWN) project (includes a deep learning benchmark)
  - Links
    - https://dawn.cs.stanford.edu/benchmark/
    - http://dawn.cs.stanford.edu/2018/04/30/dawnbench-v1-results/

- Efficient processing of deep neural networks: a tutorial and survey
  - https://arxiv.org/abs/1703.09039

Backup – Example Software

Amazon SageMaker ML Workflow

- Build
  - Collect and prepare training data using SageMaker Ground Truth
    - https://aws.amazon.com/sagemaker/groundtruth/
  - Create ML models
    - Purchase ML models in Amazon Marketplace
    - Develop ML in MXNet, TensorFlow, PyTorch or XGBoost using SageMaker Notebooks (running on Amazon Compute instances)
- Train
  - Train models SageMaker Training
  - Optimize and compile for a target platform using SageMaker Neo compiler and runtime
    - https://aws.amazon.com/blogs/aws/amazon-sagemaker-neo-train-your-machine-learning-models-once-run-them-anywhere/
    - https://aws.amazon.com/sagemaker/neo/
    - https://github.com/neo-ai/neo-ai-dlr
    - https://neo-ai-dlr.readthedocs.io/en/latest/
- Deploy
  - In the cloud with SageMaker Hosting or EC2 instances
  - On the the edge on Amazon IoT Greengrass devices
    - https://aws.amazon.com/greengrass/ml/



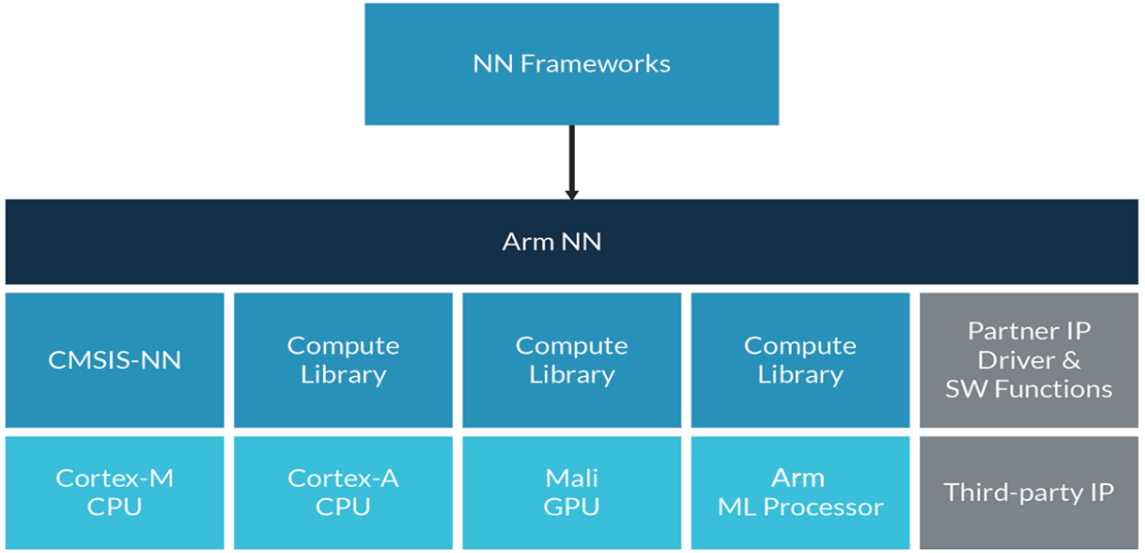
- Links
  - https://aws.amazon.com/sagemaker/
  - https://docs.aws.amazon.com/sagemaker/index.html
  - https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf

Figure from https://aws.amazon.com/sagemaker/neo/

ARM NN Graph Optimizer And Runtime

- ARM NN
  - Open source Linux software and tools to bridge between xNN frameworks and ARM cores
  - Uses the compute library to target A cores, GPUs and the ML processor
  - Does not currently provide support for M cores (uses CMSIS-NN instead)
  - Translates xNN frameworks to an internal representation and distributes to present cores (effectively a graph optimizer and runtime)
  - Open source and donated to the Linaro Machine Intelligence Initiative

- Supported xNN frameworks
  - TensorFlow, TensorFlow Lite, PyTorch, ONNX, MXNet, Caffe, Caffe2, Android NN API
  - https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/how-to-guides/configuring-the-arm-nn-sdk-build-environment-for-tensorflow
  - https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/how-to-guides/configuring-the-arm-nn-sdk-build-environment-for-tensorflow-lite
  - https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/how-to-guides/configuring-the-arm-nn-sdk-build-environment-for-onnx
  - https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/how-to-guides/configuring-the-arm-nn-sdk-build-environment-for-caffe



- Links
  - https://mlplatform.org
  - https://github.com/ARM-software/armnn
  - https://developer.arm.com/ip-products/processors/machine-learning/arm-nn
  - https://developer.arm.com/ip-products/processors/machine-learning/compute-library

Figure from https://mlplatform.org

ARM M Core Library

- ARM CMSIS-NN
  - Neural network software library for ARM M cores
  - 8 and 16 bit integer implementations

- Functions
  - Neural network convolution
  - Neural network activation
  - Fully-connected layer
  - Neural network pooling
  - Softmax
  - Neural network support

- Links
  - https://arm-software.github.io/CMSIS_5/NN/html/index.html
  - https://arxiv.org/abs/1801.06601

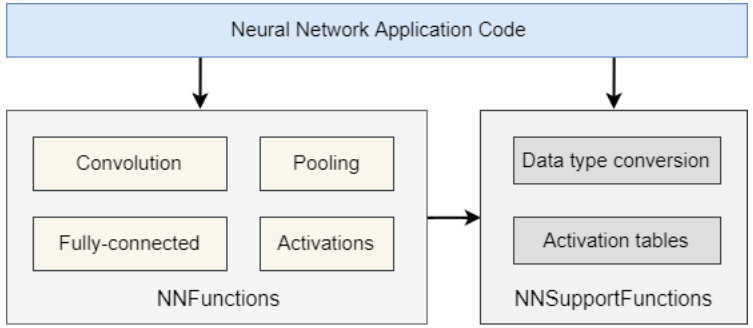


Figure from https://arm-software.github.io/CMSIS_5/NN/html/index.html

ARM A Core, Mali GPU And ML Proc Library

- Compute library
  - Basic arithmetic, mathematical, and binary operator functions
  - Color manipulation
  - Convolution filters
  - Canny Edge, Harris corners, optical flow
  - Pyramids
  - HOG
  - SVM
  - H/SGEMM
  - Convolutional Neural Networks building blocks
- Adding a new backend
  - https://github.com/ARM-software/armnn/tree/branches/armnn_19_02/src/backends

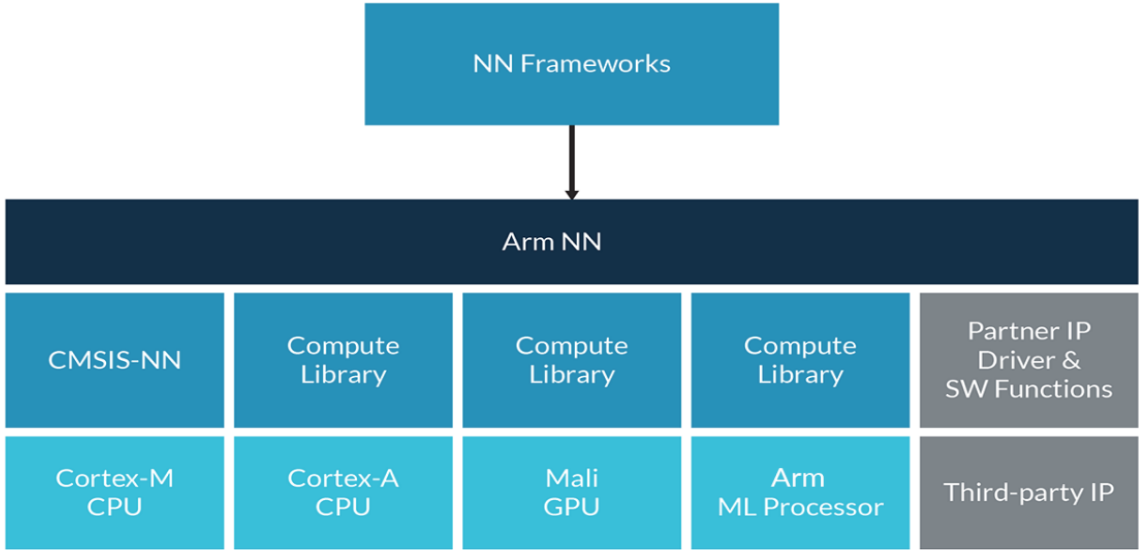
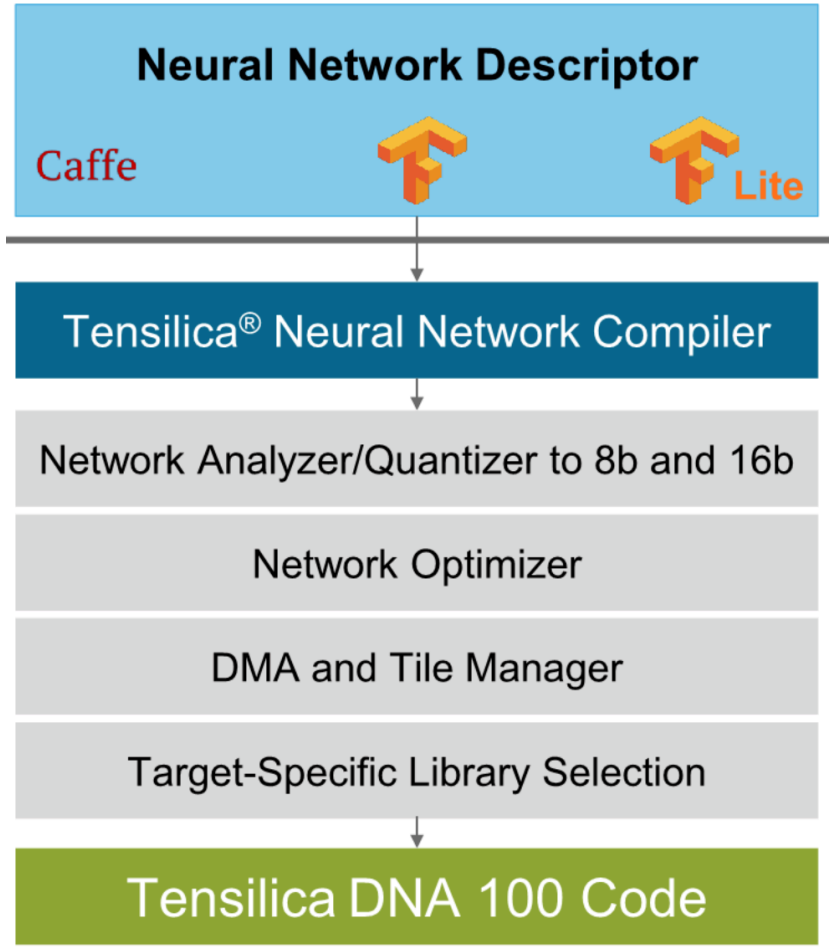


Figure from https://mlplatform.org

Cadence Tensilica DNA 100



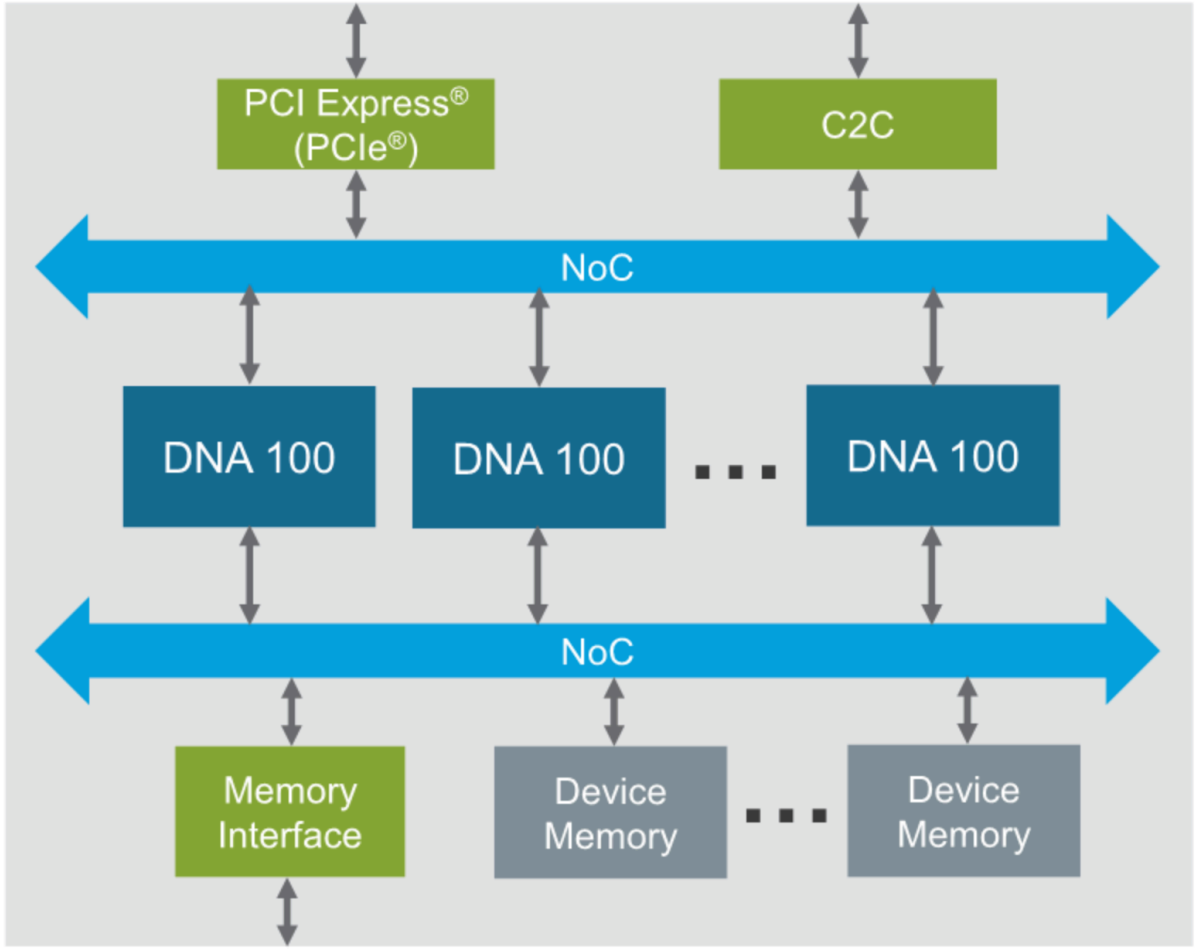


Figure from https://www.anandtech.com/show/13377/cadence-announces-tensilica-dna-100-a-bigger-nn-ip

CEVA CDNN And NeuPro

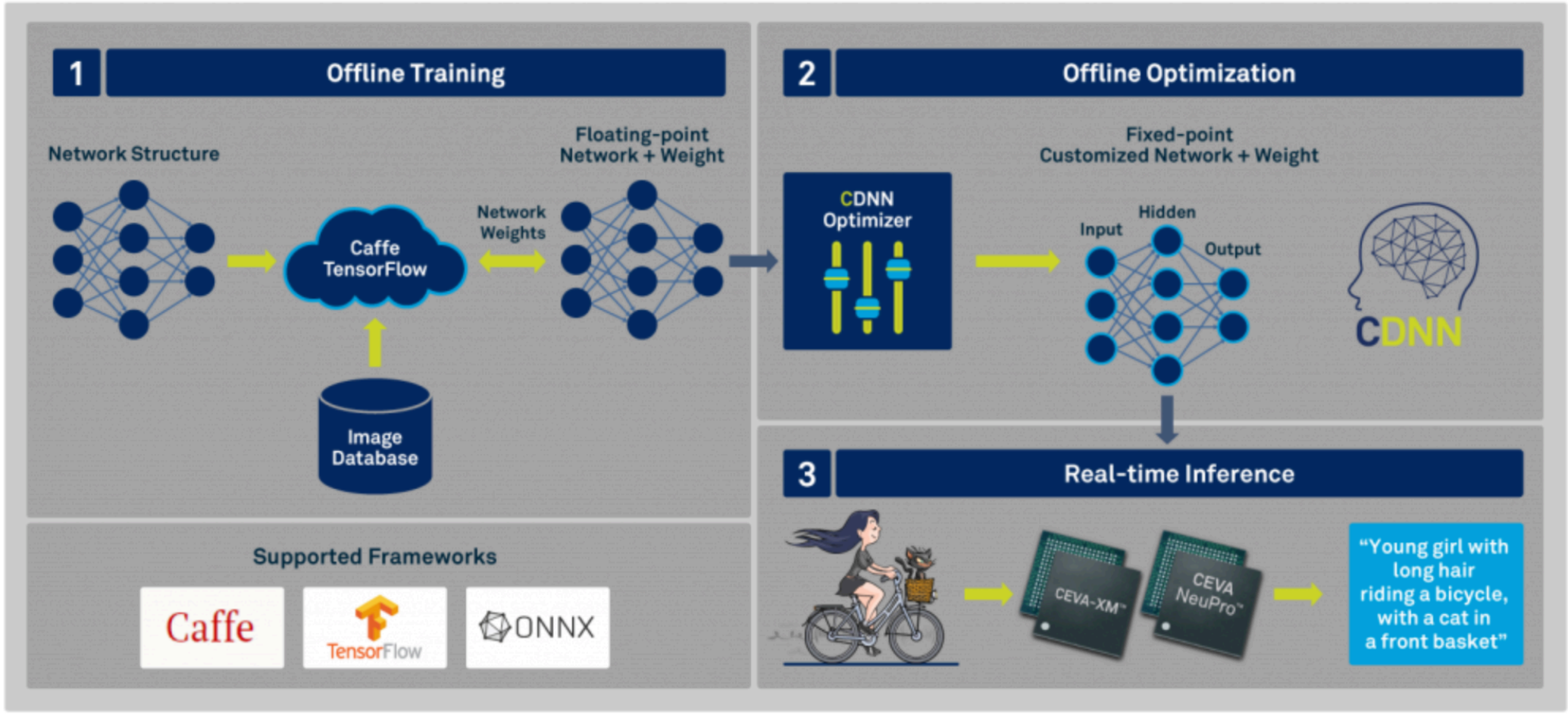
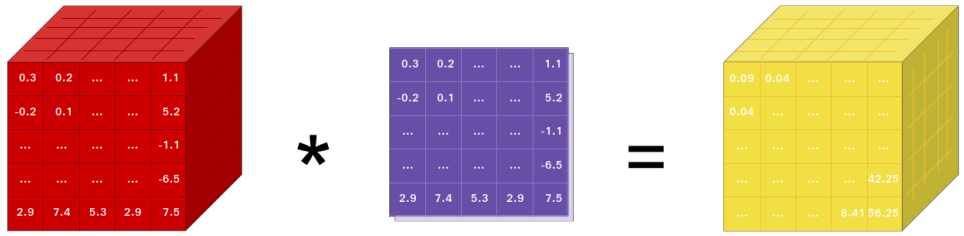


Figure from https://www.ceva-dsp.com/product/ceva-deep-neural-network-cdnn/

Facebook PyTorch

- PyTorch is Facebook's open source library for creating and training machine learning models (https://pytorch.org)



- Key features
  - Optimized tensor library for deep learning using CPUs and GPUs
  - Tape based autograd system
  - Front end support for eager and graph mode

- Components
  - Torch: NumPy like tensor library with GPU acceleration
  - Torch.nn: neural network library
  - Torch.hub: pre trained model repository
  - Torch.distributions: probability library
  - Torch.optim: optimizer library
  - Torch.autograd: tape based auto grad library
  - Torch.jit: compiler to create serializable and optimizable models from PyTorch code
  - Torch.utils: data loaders and other common utilities
  - Torch.Tensor: tensor type definition
  - Torch.distributed: distributed communication
  - Torch.multiprocessing: Python multiprocessing
  - Torch.onnx: ONNX exporter

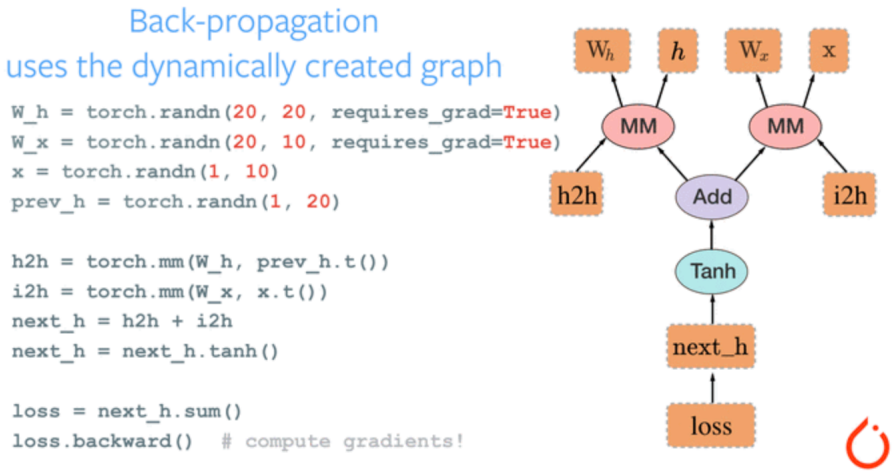
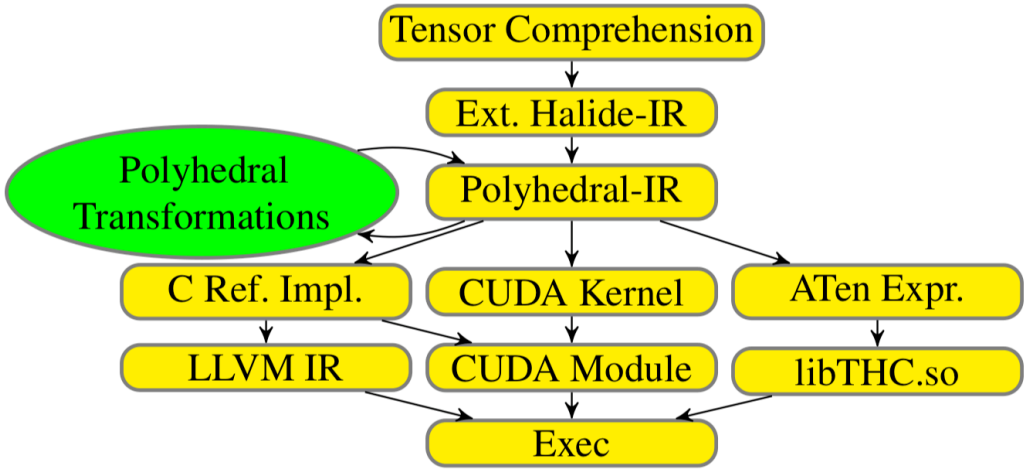


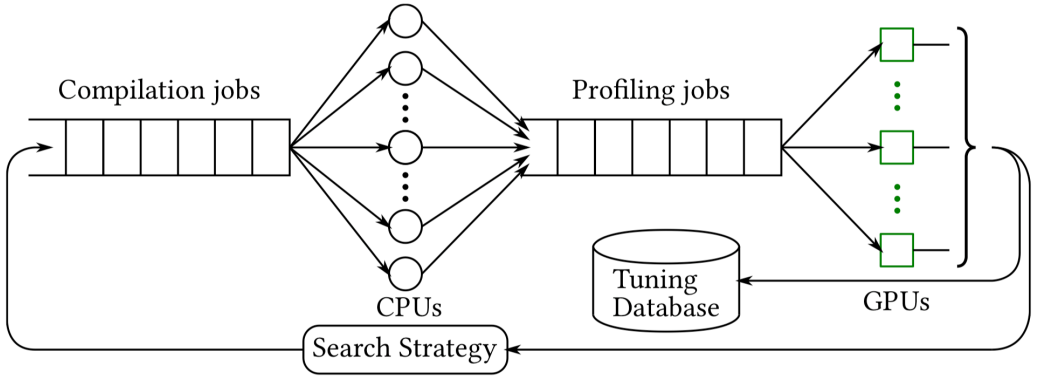
Figure from https://github.com/pytorch/pytorch

Facebook Custom Layer Optimizer

- Tensor Comprehensions
  - A notation for concisely writing layers
  - Integrates with other frameworks
  - 1 function == 1 kernel
  - No allocation of memory
  - Focuses on the loop structure implied by tensors



- Links
  - Tensor comprehensions
    - https://github.com/facebookresearch/TensorComprehensions
  - Tensor comprehensions documentation
    - https://facebookresearch.github.io/TensorComprehensions/index.html
  - Tensor comprehensions: framework-agnostic high-performance machine learning abstractions
    - https://arxiv.org/abs/1802.04730



Figures from https://arxiv.org/abs/1802.04730

Facebook Model Format

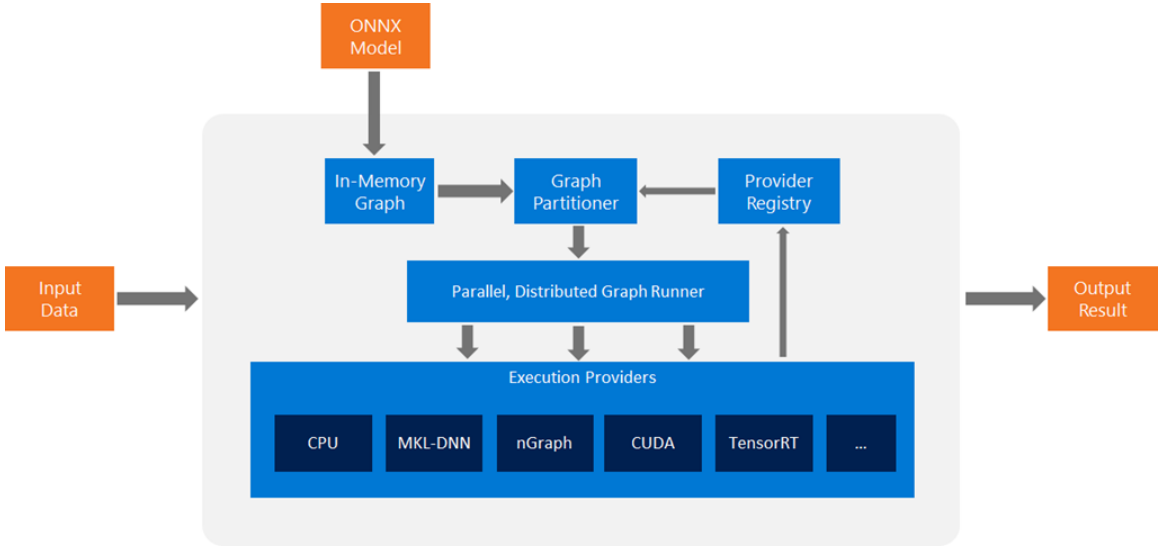
- Open neural network exchange format (ONNX)
  - An open source format for xNN models that provides a common IR that's runtime agnostic (so it needs a runtime)
  - Allows models to be trained in 1 framework and transferred to another for inference
  - Includes deep learning (standard) and classical machine learning (-ml) variants
  - See https://onnx.ai and https://github.com/onnx

- Includes
  - Definitions of an extensible computational graph model
    - https://github.com/onnx/onnx/blob/master/onnx/onnx.proto3
    - https://github.com/onnx/onnx/blob/master/onnx/onnx-ml.proto3
  - Definitions of standard data types
  - Definitions of built in operators
    - https://github.com/onnx/onnx/blob/master/docs/Operators.md
    - https://github.com/onnx/onnx/blob/master/docs/Operators-ml.md

- Tools
  - Frameworks: PyTorch, Caffe2, Cognitive Toolkit, MXNet, Chainer, PaddlePaddle, Matlab, SAS, Neural Network Libraries
  - Converters: TensorFlow, Keras, CoreML, sciit-learn, XGBoost, LIBSVM, ncnn
  - Visualizers: Netron, VisualDL
  - Compilers: Intel AI, Skymizer, TVM
  - Runtimes: Nvidia, Qualcomm, BitMain, Tencent, Vespa, Windows, Synopsys, **ONNX Runtime**, Ceva, MACE, Habana

Facebook Model Runtime (From MSFT)

- ONNX Runtime
  - An inference engine for ONNX models created by Microsoft
  - Currently supports CUDA, TensorRT, MLAS, MKL-DNN, MKL-ML and nGraph execution providers (execution provider == custom accelerator / runtime abstraction)
  - https://github.com/microsoft/onnxruntime
  - https://github.com/microsoft/onnxruntime/blob/master/docs/HighLevelDesign.md



- Flow
  - Convert ONNX model to in memory graph representation
  - Perform execution provider independent optimizations
  - Partition the graph into sub graphs based on the available execution providers
  - Assign sub graphs to execution providers

- Extensibility
  - Adding custom operators / kernels
  - Adding execution providers
  - Adding graph transformations

Figure from https://github.com/microsoft/onnxruntime/blob/master/docs/HighLevelDesign.md

Facebook Model Compiler And Runtime

- Glow is a machine learning compiler and runtime for hardware accelerators
  - Not all input operators need to be supported on all hardware back ends
- High level graph IR
  - Strongly types node based graph representation
  - Domain specific target independent optimizations
- Low level graph IR
  - Instruction based address only intermediate representation allows copy elimination, static memory allocation and instruction scheduling
- Machine code
  - Hardware specific code generation

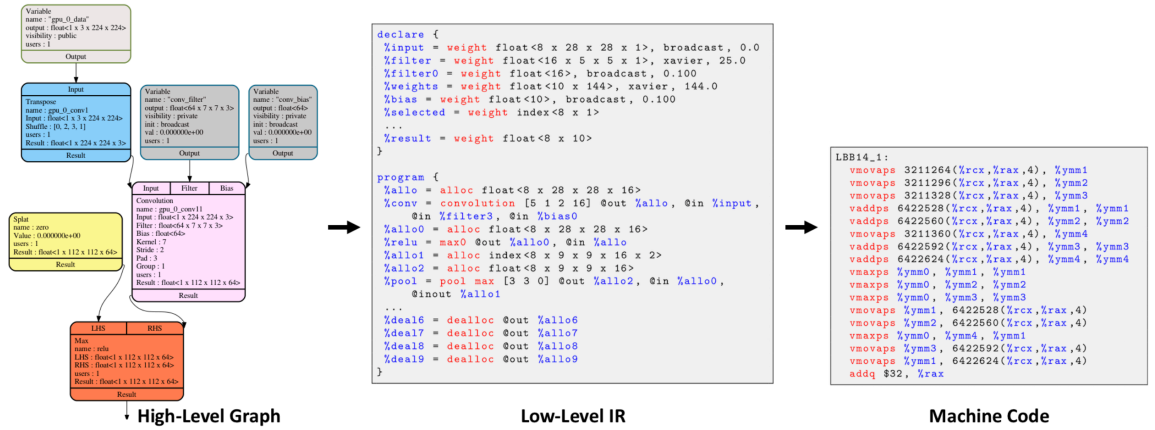


Figure from https://github.com/pytorch/glow

Facebook Model Compiler And Runtime

- Compiler flow (from https://arxiv.org/abs/1805.00907)
  1. The graph is either loaded via the graph loader (from ONNX or Caffe2 format) or constructed via the C++ interface
  2. The graph is differentiated if needed
  3. The graph is optimized
  4. Linear algebra node lowering takes place
  5. Additional rounds of optimizations occur, both target independent and target specific
  6. The graph is scheduled into a linear sequence of nodes that minimizes memory usage
  7. IRGen converts the low level graph into instructions
  8. Low level IR optimizations are performed
  9. Backend specific optimizations and code generation are performed

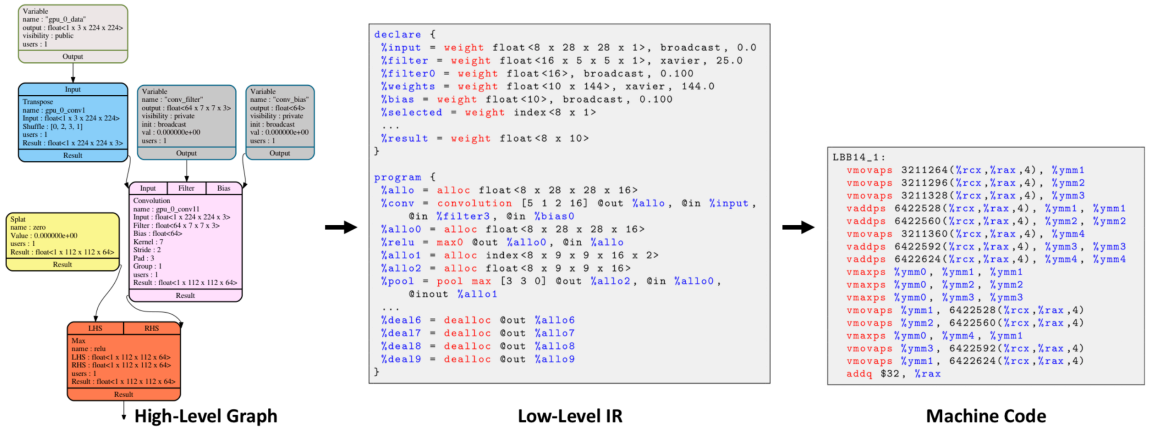


Figure from https://github.com/pytorch/glow

Facebook Model Compiler And Runtime

- Runtime flow for adding a network (from https://arxiv.org/abs/1805.00907)
  1. The Partitioner splits the network into one or more sub networks
  2. The Provisioner compiles each sub network and assigns them to one or more devices
  3. One or more DeviceManagers load the sub networks and their weights onto its associated device

- Runtime flow for handling an inference request (from https://arxiv.org/abs/1805.00907)
  1. The HostManager creates a new execution graph with intermediate storage
  2. The Executor kicks off the first sub network execution
  3. The DeviceManager loads inputs onto the card and begins execution; when done it reads outputs and signals completion
  4. The Executor triggers any sub networks with satisfied dependencies
  5. When complete the HostManager returns outputs

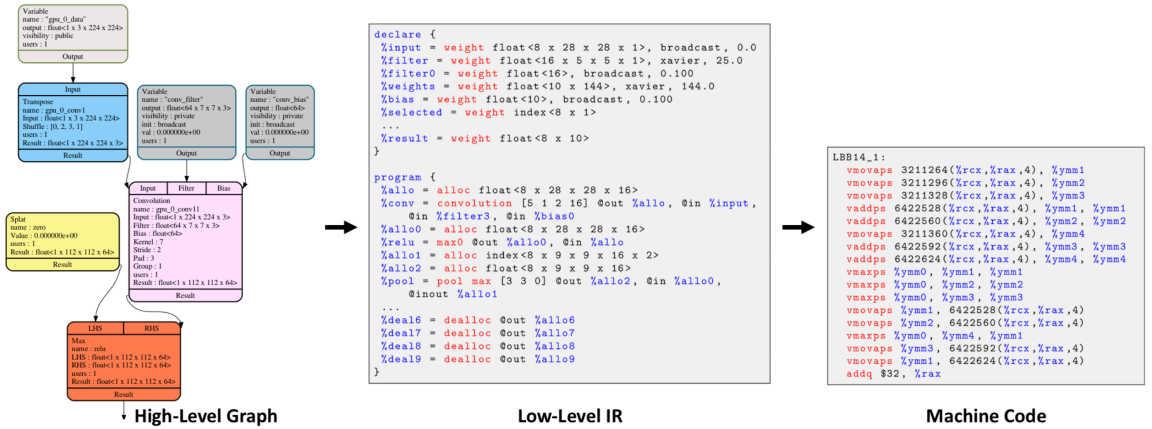
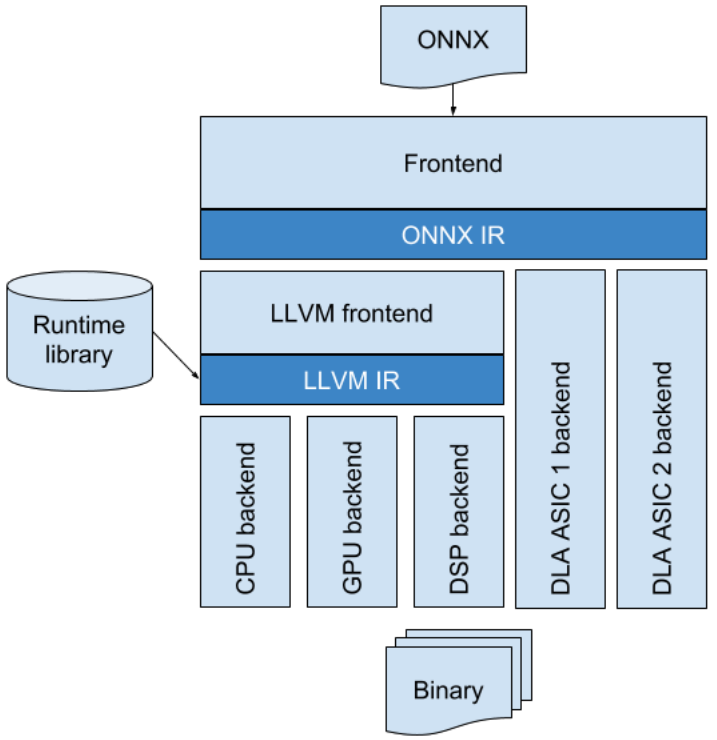
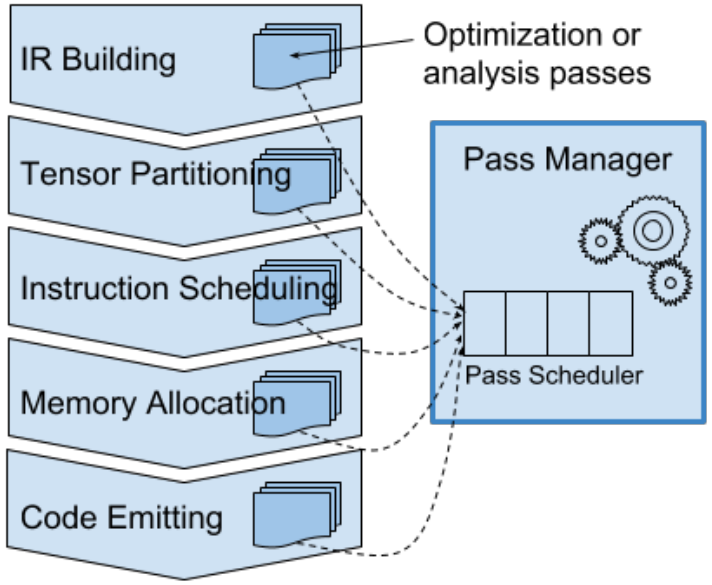


Figure from https://github.com/pytorch/glow

Facebook Model Compiler And Runtime

- From ONNC (not FB)
- ONNX API and IR with LLVM and generic interfaces
  - Allows targeting of CPUs, DSPs and GPUs via LLVM code
  - Allows targeting of DSAs via generic interface
- For more info
  - https://onnc.ai
  - https://github.com/ONNC/onnc
  - https://www.youtube.com/watch?time_continue=1&v=-FuKZFfWIXo





Figures from https://onnc.ai

Google TensorFlow

- TensorFlow is Google's open source library for creating and training machine learning models
  - 2.x beta: https://www.tensorflow.org/beta

- Versions
  - 1.x: default is declarative execution
    - Specify graph, execute graph
  - 2.x: default is eager execution
    - Imperative environment, operations execute immediately
    - The standard method for using the Keras API in 2.x is ~ declarative like
    - And the @tf.function decorator can be used to create declarative code
    - Declarative code (specify graph, execute graph) is good for performance

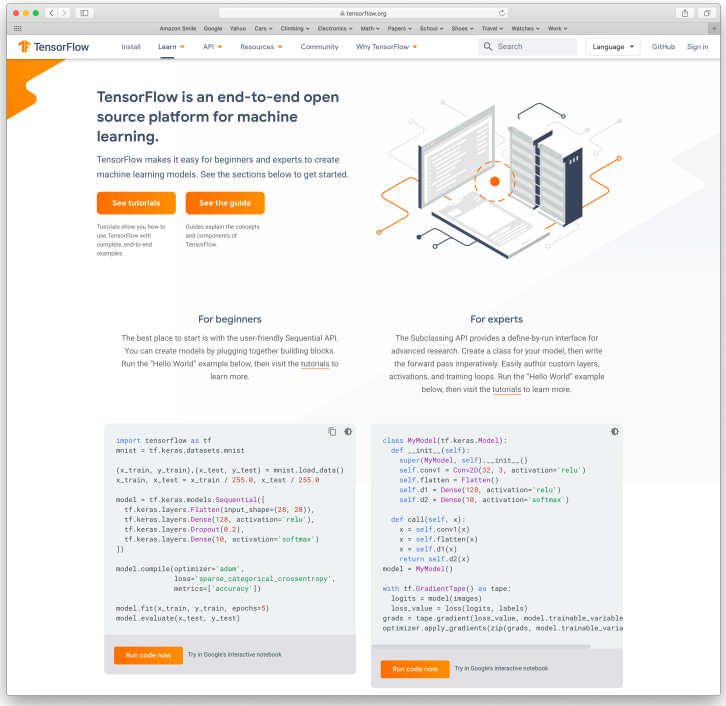


Figure from https://www.tensorflow.org/overview

Google TensorFlow

Examples below indicate the tf.data or high level Keras API

- High level graph specification
  - Data (tf.data)
  - Network (tf.keras.Model)
  - Loss (tf.keras.Model.compile)
  - Gradient back prop (tf.keras.Model.compile)
  - Weight update (tf.keras.Model.compile)
- High level graph execution
  - Training (tf.keras.Model.fit)
  - Evaluation (tf.keras.Model.evaluate)
  - Prediction (tf.keras.Model.predict)

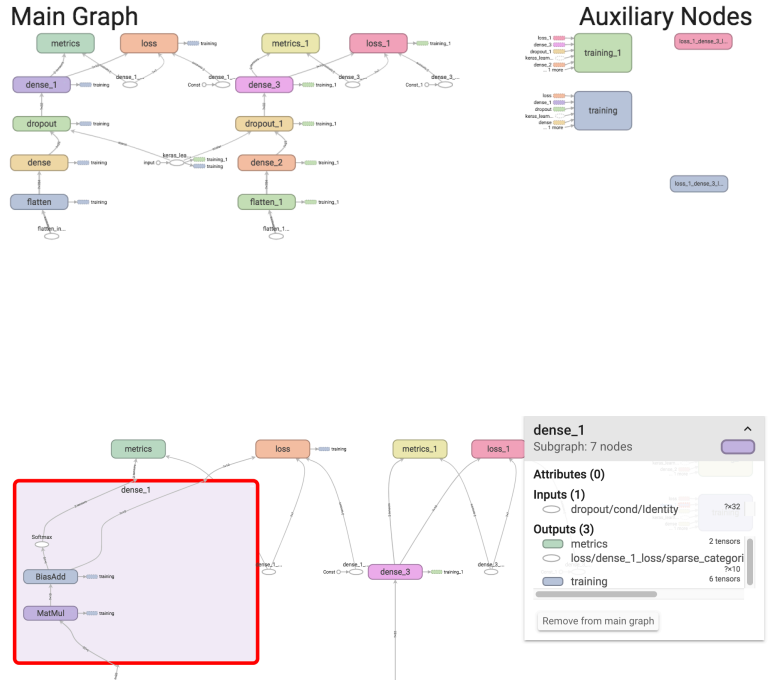
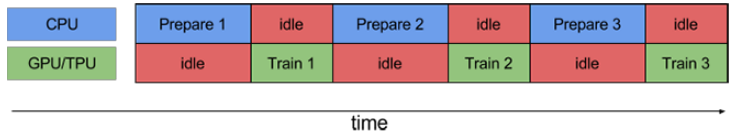


Figure from https://www.tensorflow.org/tensorboard/r2/graphs

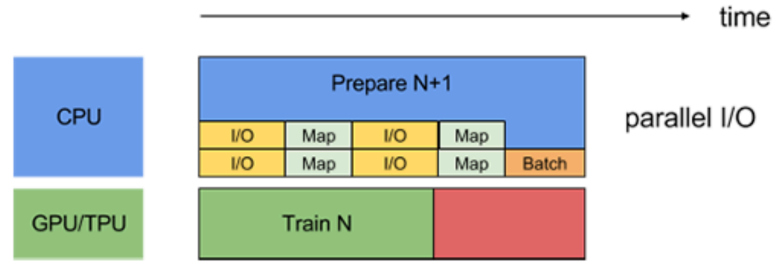
Google TensorFlow Graph Specification

Data

- The tf.data API is used for building data pipelines
  - Inputs and true outputs
  - Training, validation and prediction
  - The data pipeline is part of the graph



- The tf.data.Dataset abstraction is used to represent a sequence of elements where each element consists of 1 or more tensors
  - Basic strategy is to create a datasets from data in memory or files
  - Then transform the dataset via map, shuffle, batch and repeat operations
  - To improve performance consider pre fetching to overlap data I/O for the next batch on the CPU with processing the current batch on the GPU, parallel data mapping and parallel data I/O



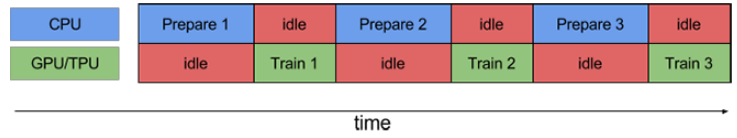
- References
  - https://www.tensorflow.org/beta/guide/data
  - https://www.tensorflow.org/beta/guide/data_performance

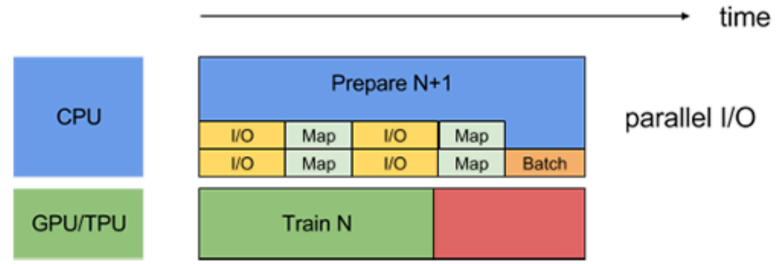
Figures from https://www.tensorflow.org/beta/guide/data_performance

Google TensorFlow Graph Specification

Data

- For examples of specific types of data
  - Load CSV with tf.data
    - https://www.tensorflow.org/beta/tutorials/load_data/csv
  - Load NumPy Data with tf.data
    - https://www.tensorflow.org/beta/tutorials/load_data/numpy
  - Load images with tf.data
    - https://www.tensorflow.org/beta/tutorials/load_data/images
  - Load text with tf.data
    - https://www.tensorflow.org/beta/tutorials/load_data/text
  - Using TFRecords and tf.Example
    - https://www.tensorflow.org/beta/tutorials/load_data/tf_records
  - Unicode strings
    - https://www.tensorflow.org/beta/tutorials/text/unicode
  - TF.Text
    - https://www.tensorflow.org/beta/tutorials/tensorflow_text/intro





Figures from https://www.tensorflow.org/beta/guide/data_performance

Google TensorFlow Graph Specification

Data

- The Keras API simplifies downloading and loading data from public research datasets

- References
  - https://keras.io/datasets/

```
# import
from keras.datasets import mnist

# download and load MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Google TensorFlow Graph Specification

Data

- TensorFlow Datasets simplifies downloading, loading and creating tf.data.Datasets from public research datasets

- References
  - https://medium.com/tensorflow/introducing-tensorflow-datasets-c7f01f7e19f3
  - https://www.tensorflow.org/datasets
  - https://www.tensorflow.org/datasets/datasets
  - https://www.tensorflow.org/datasets/overview
  - https://www.tensorflow.org/datasets/api_docs/python/tfds

```
# see all registered datasets
tfds.list_builders()

# load a given dataset by name, along with the dataset info
data, info = tfds.load("mnist", with_info=True)
train_data, test_data = data['train'], data['test']
assert isinstance(train_data, tf.data.Dataset)
assert info.features['label'].num_classes == 10
assert info.splits['train'].num_examples == 60000

# you can also access a builder directly
builder = tfds.builder("mnist")
assert builder.info.splits['train'].num_examples == 60000
builder.download_and_prepare()
datasets = builder.as_dataset()

# if you need NumPy arrays
np_datasets = tfds.as_numpy(datasets)
```

Code from https://www.tensorflow.org/datasets/datasets

Google TensorFlow Graph Specification

Network

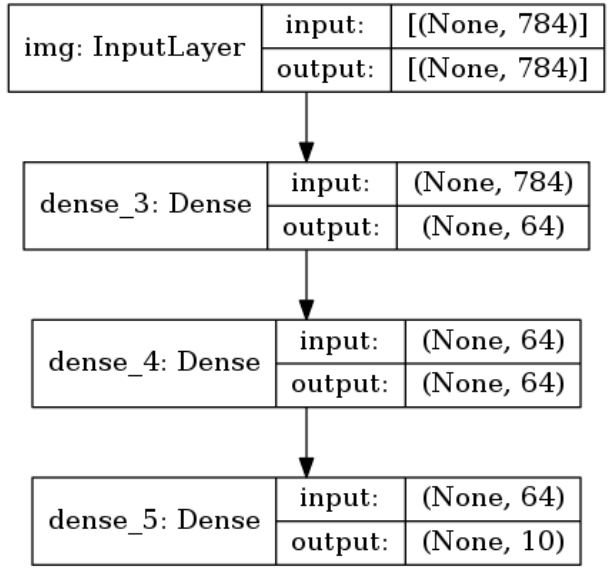
- Networks
  - Are specified as graphs
    - Edges are memory or dependencies
    - Nodes are operators (layers)
  - Map data inputs to outputs
    - Encoder output = embeddings (typically)
    - Encoder + decoder output = predictions (typically)

- High and low level APIs are available for network specification
  - **High level**
    - TensorFlow 2.x has standardized on Keras
    - There's a general trend towards this providing an appropriate level of control and being appropriate for most work
    - https://keras.io
    - https://www.tensorflow.org/beta/guide/keras/overview
  - Low level

Google TensorFlow Graph Specification

Network

- Keras provides multiple high level API options for specifying network models
  - Overview
    - https://keras.io/models/about-keras-models/
  - Sequential API
    - Ok for getting started with simple examples, but not flexible with respect to topology
    - https://keras.io/getting-started/sequential-model-guide/
    - https://keras.io/models/sequential/
  - Functional API
    - A nice level of abstraction for most networks
    - https://keras.io/getting-started/functional-api-guide/
    - https://keras.io/models/model/
    - https://www.tensorflow.org/beta/guide/keras/functional
  - Model sub classing
    - Create layers in __init__ and define the forward pass in call; this allows the forward pass to be run imperatively, but it is preferred to use the functional API when possible
    - https://keras.io/models/about-keras-models/#model-subclassing
    - https://www.tensorflow.org/beta/guide/keras/overview#model_subclassing
    - https://www.tensorflow.org/beta/guide/keras/custom_layers_and_models#building_models



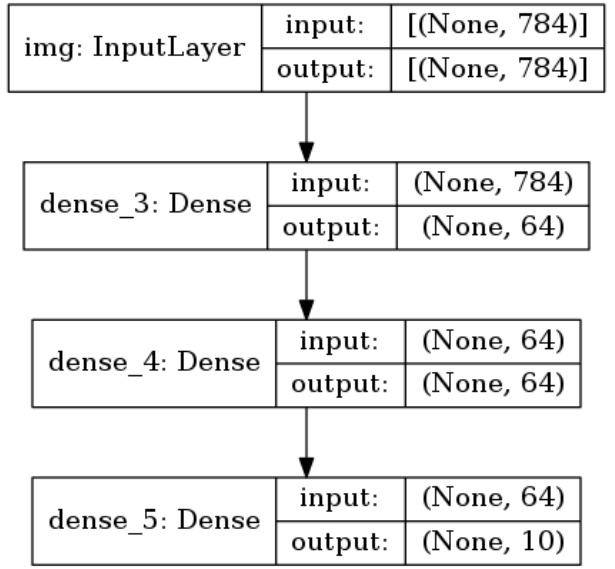
- model.summary() creates a text description of the model including layers, output shapes and numbers of parameters
- keras.utils.plot_model(model, 'file_name.png') creates a figure showing connections
- keras.utils.plot_model(model, 'file_name.png', show_shapes=True) creates a figure showing connections and input / output feature map sizes

Figure from https://www.tensorflow.org/beta/guide/keras/functional

Google TensorFlow Graph Specification

Network

- Things to consider when specifying models
  - Having separate access to the encoder (and possibly multiple points in the encoder) and decoder outputs to simplify transfer learning
  - Enabling arbitrary numbers of repeats for building blocks to simplify the creation of different accuracy vs performance models
- Models are built from layers
  - Built in layers
    - https://keras.io/layers/about-keras-layers/
  - Custom layers
    - https://keras.io/layers/writing-your-own-keras-layers/
    - https://www.tensorflow.org/beta/guide/keras/custom_layers_and_models#the_layer_class
- Operator placement
  - https://www.tensorflow.org/beta/guide/using_gpu
  - Default operator placement uses GPUs when implementations are possible, falls back to CPU when not



- model.summary() creates a text description of the model including layers, output shapes and numbers of parameters
- keras.utils.plot_model(model, 'file_name.png') creates a figure showing connections
- keras.utils.plot_model(model, 'file_name.png', show_shapes=True) creates a figure showing connections and input / output feature map sizes

Figure from https://www.tensorflow.org/beta/guide/keras/functional

Google TensorFlow Graph Specification

Loss, gradient back prop and weight update

- There are different options available for training
  - **Built in training loop**
    - Keras losses https://keras.io/losses/ , weight update https://keras.io/optimizers/ and metrics https://keras.io/metrics/
    - https://www.tensorflow.org/beta/guide/keras/training_and_evaluation#part_i_using_build-in_training_evaluation_loops
    - 1: compile (define loss, implicitly define back prop, specify weight update)
    - 2: fit
    - 3: evaluate and predict
  - Custom training loop
    - https://www.tensorflow.org/beta/guide/keras/training_and_evaluation#part_ii_writing_your_own_training_evaluation_loops_from_scratch
    - 1: define loss, define weight update, explicitly use gradient tape to record forward pass to implicitly define back prop
    - 2: explicitly retrieve gradients, use gradients with weight update
    - 3: explicitly evaluate and predict

- Why mention the above training items during graph specification?
  - Because the loss, back prop and weight update can be thought of as part of the graph needed for training
  - And when using the high level API they're specified in the compile function

Google TensorFlow Graph Specification

Loss, gradient back prop and weight update

- Loss and optimizer (tf.keras.Model.compile)
  - Adds a loss
  - Backwards path is automatically created from forwards path
  - Adds an optimizer to the model
  - Specifies metrics to track

Google TensorFlow Graph Specification

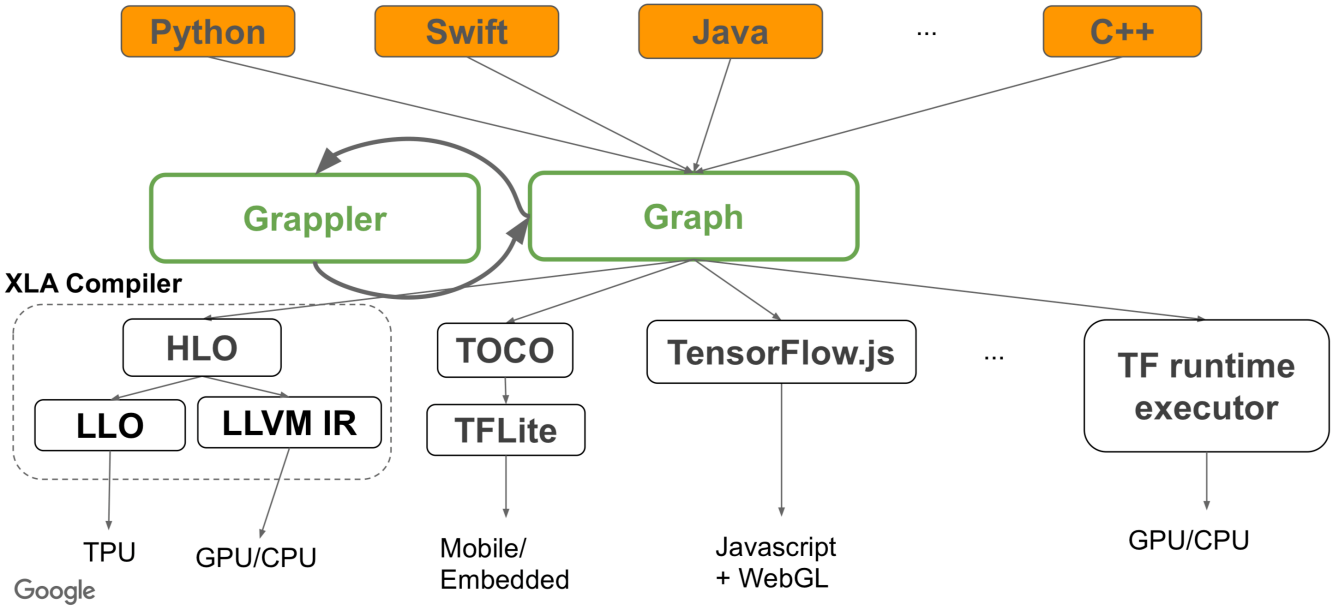
Pre trained models

- TensorFlow Hub provides a library of pre trained graphs / graph fragments referred to as modules that can be used stand alone or as part of other graphs
  - Ex: using a pre trained image to feature encoder used as the encoder for Faster R-CNN

- References
  - https://www.tensorflow.org/hub
  - https://tfhub.dev

Google TensorFlow Graph Optimization

- Grappler is the default high level graph optimizing system in the TensorFlow runtime
  - Re writes TensorFlow high level graphs to improve out of the box TensorFlow performance
  - Includes plug in infrastructure to register custom optimizers and high level graph re writers
- TensorFlow graph optimizations
  - http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf



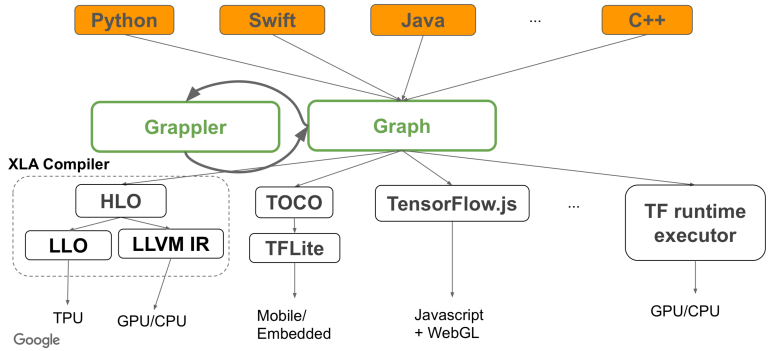
- You don't explicitly call Grappler for graph optimization
- It's called for you automatically during the compile process
- But it's worthwhile to know that it's there, the functionality that it provides and how it can be extended / modified

Figure from http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf

Google TensorFlow Graph Optimization

- Optimizer loop (from below reference)

```
i= 0
while i < num iterations (default=2):
   Pruning()          # remove nodes not in fanin of outputs, unused functions
   Function()         # function specialization and inlining, symbolic gradient inlining
   DebugStripper()    # remove assert, print, check_numerics
   ConstFold()        # constant folding and materialization
   Shape()            # symbolic shape arithmetic
   Remapper()         # op fusion
   Arithmetic()       # node deduping (CSE) and arithmetic simplification
   if i==0: Layout()  # layout optimization for GPU
   if i==0: Memory()  # swap out / swap in, recompute, split large nodes
   Loop()             # loop invariant node motion, stack push and dead node elimination
   Dependency()       # prune / optimize control edges, noop / identity node pruning
   Custom()           # run registered custom optimizers (e.g. TensorRT)
   i += 1
```

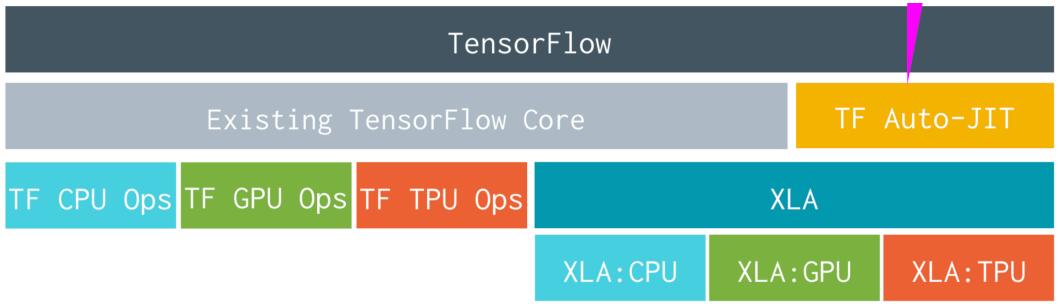


- TensorFlow graph optimizations
  - http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf

Figure from http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf

Google Sub Graph Optimization

- XLA is a domain specific compiler for TensorFlow computation analysis, optimization and code generation
  - Example use: fuse together a number of primitive operations (a sub graph) into a single larger operation (node) optimized for a specific target and generate associated code



- Overview
  - TensorFlow w/XLA: TensorFlow, compiled!
  - https://autodiff-workshop.github.io/slides/JeffDean.pdf

Figure from https://autodiff-workshop.github.io/slides/JeffDean.pdf

Google Sub Graph Optimization

- Improve execution speed
  - Compile sub graphs to eliminate the overhead of the TensorFlow runtime
  - Fuse pipelined operations to reduce memory overhead
  - When an individual op is large all is well (assuming that there's not a large on device memory for linking big ops) but when individual ops are small this helps
  - Specialize to known tensor shapes to allow for more aggressive constant propagation
- Improve memory usage
  - Analyze and schedule memory usage
  - Eliminate intermediate storage buffers
- Reduce reliance on custom ops
  - Fuse low level ops to new high level ops while maintaining performance
- Reduce mobile footprint
  - Remove TensorFlow runtime by ahead of time compiling the sub graph to an object / header file that can then be linked into another program
- Improve portability
  - Make it easy to write a new backend for novel hardware

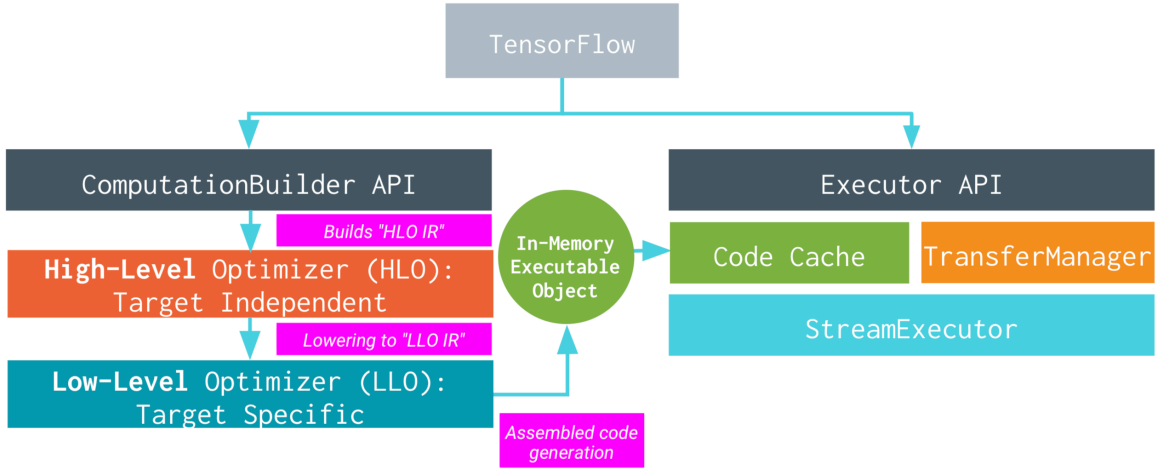


Figure from https://autodiff-workshop.github.io/slides/JeffDean.pdf

Google Sub Graph Optimization

- The input to XLA is high level optimizer (HLO) IR
  - These are graphs with graph operations are defined in operation semantics (https://www.tensorflow.org/xla/operation_semantics)
  - Think of these as all of the supported operations (there are a lot)
  - A while loop is also included
- Target independent front end does analysis and optimization
  - Common sub expression elimination
  - Target independent operation fusion
  - Buffer analysis of allocating runtime memory (though real optimization of this should be hardware specific)
  - The output of this is still HLO IR
- Target dependent back end does analysis and optimization and code generation for CPUs, GPUs and TPUs
  - Hardware target specific optimization
  - The output is LLVM for the CPU and GPU

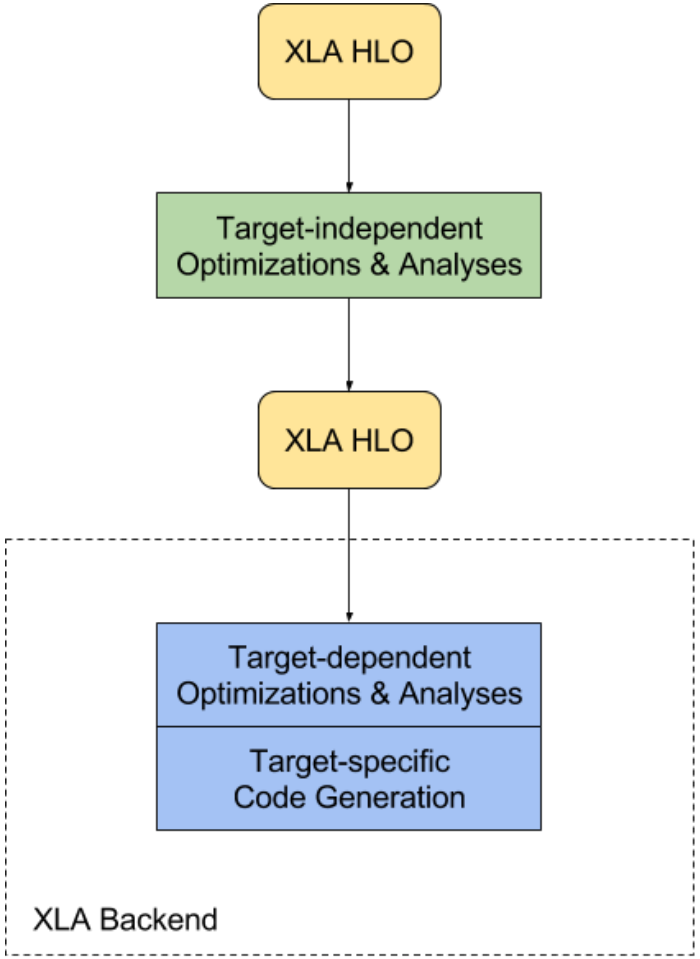


Figure from https://www.tensorflow.org/xla/overview

Google Sub Graph Optimization

- Supported operations and operand broadcasting semantics
  - https://www.tensorflow.org/xla/operation_semantics
  - https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/client/xla_builder.h
  - https://www.tensorflow.org/xla/broadcasting
- Operand shape and memory layout specification
  - Shapes
    - https://www.tensorflow.org/xla/shapes
    - Allows specification of row or col major order (and generalizations)
    - Allows specification of 0 padding dimensions to a larger value (ut not the same as symmetric 0 padding)
  - Tiled layouts
    - https://www.tensorflow.org/xla/tiled_layout
    - Indexing for tiling a larger matrix
    - Can be applied recursively
    - Likely used to tile problems for the TPU

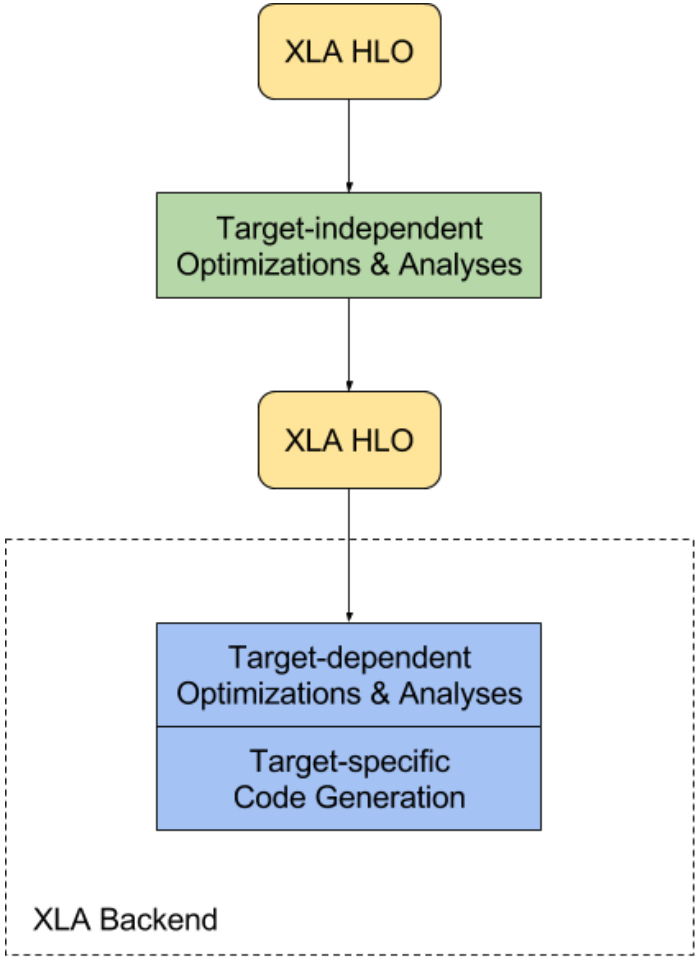


Figure from https://www.tensorflow.org/xla/overview

Google Sub Graph Optimization

- Methods for developing a new backend (to add a new processor)
  - https://www.tensorflow.org/xla/developing_new_backend
  - 3 scenarios
    - 1: CPU architecture with a LLVM backend
    - 2: non CPU architecture with a LLVM backend
    - 3: non CPU architecture without a LLVM backend
  - For scenario 3 need to implement the following functions
    - StreamExecutor to load and launch kernels and invoke pre canned library routines
    - xla::Compiler to encapsulate the compilation of a HLO computation to an xla::Executable
    - xla::Executable to launch a compiled computation to the platform
    - xla::TransferManager to transfer of data between device and host

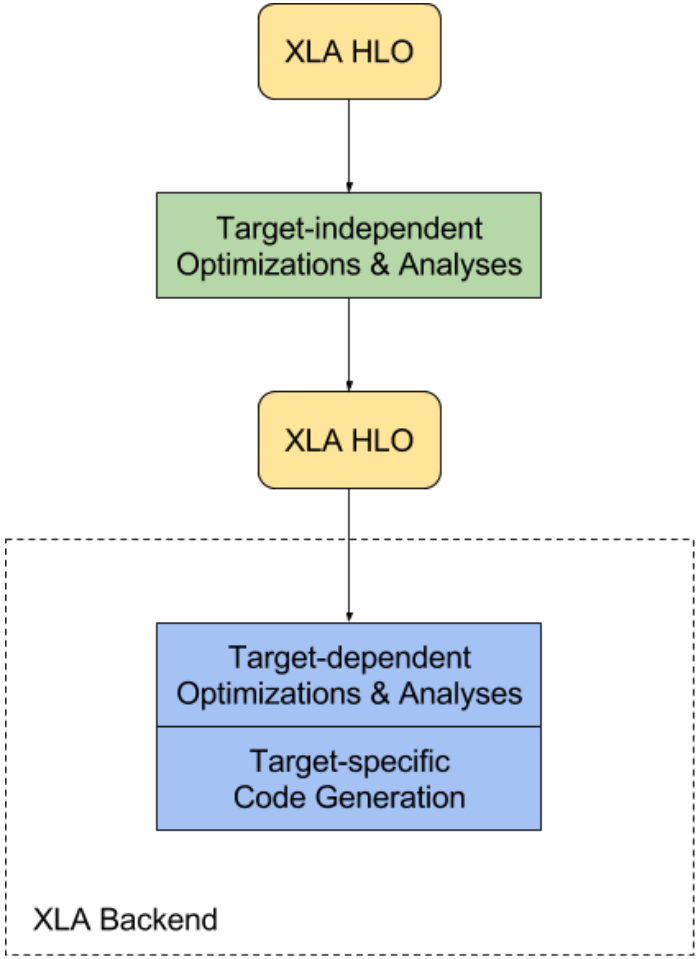


Figure from https://www.tensorflow.org/xla/overview

Google Sub Graph Optimization

- Ahead of time compiler
  - Appropriate for inference
  - tfcompile
    - https://www.tensorflow.org/xla/tfcompile
    - Compiles TensorFlow sub graphs into executable code
    - A sub graph is defined by feeds (input arguments) and fetches (output arguments)
    - Returns a compiled function that implements the sub graph
    - Does not use the TensorFlow runtime
    - Only has dependencies on the kernels in the sub graph
  - Colab example
    - https://www.tensorflow.org/xla/tutorials/xla_compile
    - Shows how to compile a model with specified inputs and output(s)
    - Currently doesn't support with tf.keras.Model.fit or eager mode
  - GitHub example
    - https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/g3doc/tfcompile.md
    - 1: Configure the sub graph to compile
    - 2: Use tf_library build macro to compile the sub graph
    - 3: Write code to invoke the sub graph
    - 4: Create the final binary

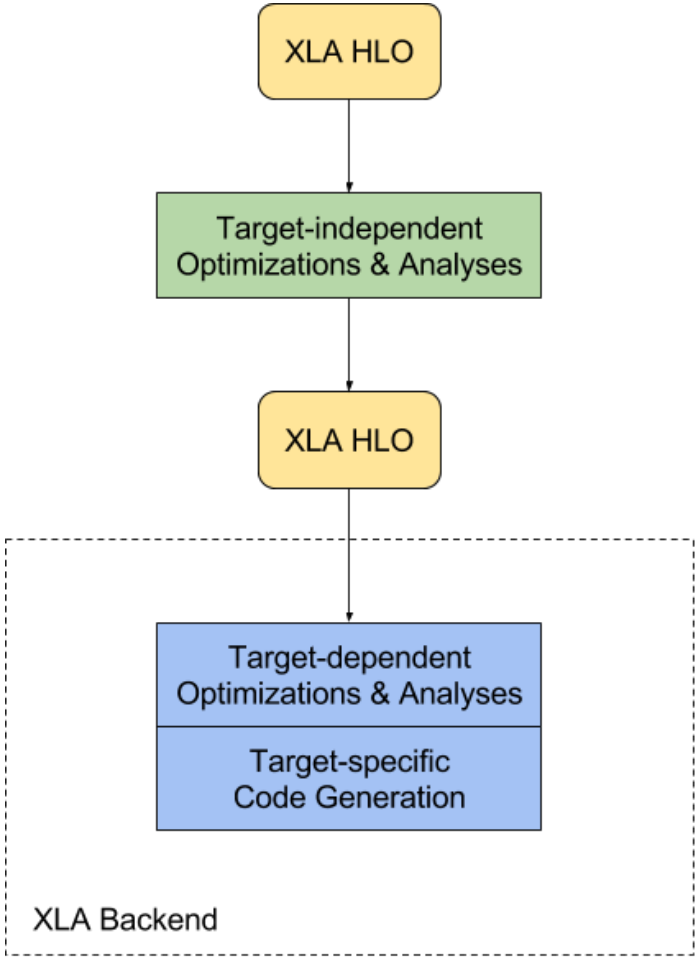


Figure from https://www.tensorflow.org/xla/overview

Google Sub Graph Optimization

- Just in time compiler
  - Appropriate for training
  - compile
    - https://www.tensorflow.org/xla/jit
    - Operations are marked on the graph for XLA
    - These operations are compiled into 1 or more kernels for the device
    - Operator fusion happens for consecutive graph nodes
  - GitHub example
    - https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/g3doc/jit.md

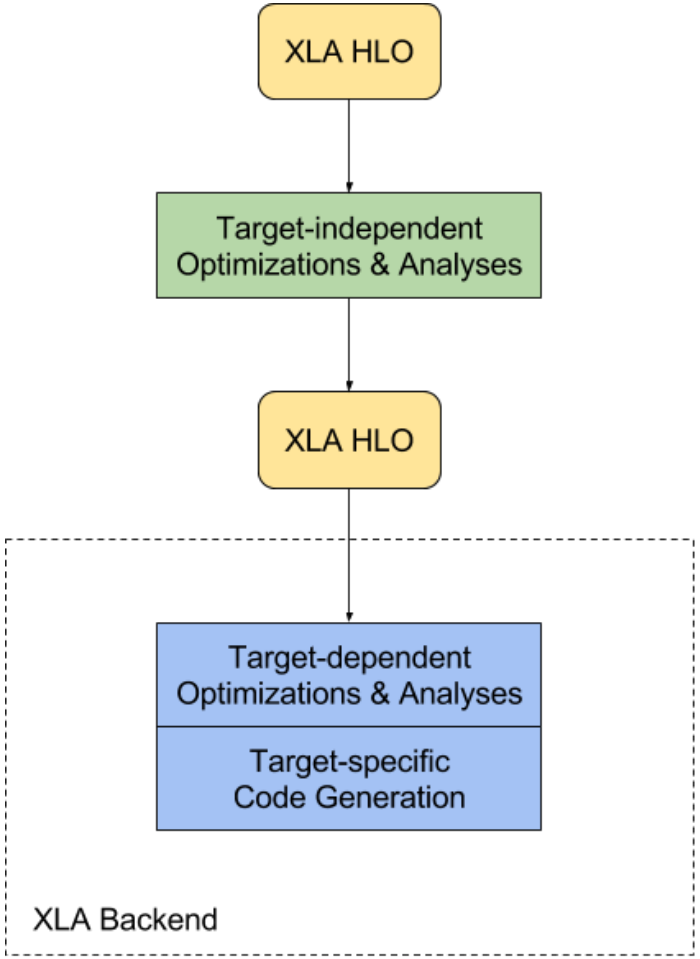


Figure from https://www.tensorflow.org/xla/overview

Google TensorFlow Graph Execution

Training

- Training (tf.keras.Model.fit)
  - https://www.tensorflow.org/beta/guide/keras/training_and_evaluation
  - Applies the model to the training inputs and computes the loss between the true and predicted outputs
  - Uses the optimizer to update the model parameters to minimize the loss
  - Specifies training parameters like epochs, batch size and validation set
  - Tracks loss and metrics

- Distributed training
  - https://www.tensorflow.org/beta/guide/distribute_strategy
  - https://www.tensorflow.org/beta/tutorials/distribute/keras
  - https://www.tensorflow.org/beta/tutorials/distribute/training_loops
  - https://www.tensorflow.org/beta/tutorials/distribute/multi_worker_with_keras
  - MirroredStrategy: synchronous distributed training on multiple GPUs on one machine with each variable mirrored across all GPUs
  - CentralStorageStrategy: synchronous distributed training on multiple GPUs on one machine with variables on the CPU
  - MultiWorkerMirroredStrategy: MirroredStrategy with multiple machines
  - TPUStrategy: MirroredStrategy for TPUs
  - ParameterServerStrategy: some machines are parameter servers and some machines are workers

Google TensorFlow Graph Execution

Training utilities

- Saving and restoring
  - https://www.tensorflow.org/beta/guide/keras/saving_and_serializing
  - Strategy for saving and restoring models created using the serial and functional APIs
    - https://www.tensorflow.org/beta/guide/keras/saving_and_serializing#part_i_saving_sequential_models_or_functional_models
  - Strategy for saving and restoring models created via model sub classing
    - https://www.tensorflow.org/beta/guide/keras/saving_and_serializing#saving_subclassed_models

- Options
  - Whole model saving
    - Architecture, weights, training configuration and optimizer state
    - Enables restarting training (useful, things will crash)
    - Can export to Keras or native TensorFlow formats
  - Architecture only saving
  - Weights only saving

Google TensorFlow Graph Execution

Training utilities

- Callbacks
  - Callbacks are functions applied during specific points in training that can be applied to fit, evaluate and predict
  - Built in callbacks are provided and custom callbacks can be defined by extending the base class keras.callbacks.Callback
  - https://keras.io/callbacks/
  - https://www.tensorflow.org/beta/guide/keras/custom_callback
  - https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/callbacks

- Built in callbacks
  - History: records events into a History object
  - RemoteMonitor: stream events to a server
  - BaseLogger: accumulates epoch averages of metrics
  - ProgbarLogger: prints metrics to stdout
  - CSVLogger: streams epoch results to a csv file
  - TensorBoard: writes a log for TensorBoard
  - ModelCheckpoint: save the model after every epoch
  - LearningRateScheduler: learning rate scheduler
  - ReduceLROnPlateau: reduce learning rate when a metric has stopped improving
  - TerminateOnNaN: terminates training when a NaN loss is encountered
  - EarlyStopping: stop training when a monitored quantity has stopped improving
  - LambdaCallback: creates simple, custom callbacks on-the-fly

- In TensorFlow 2.x TensorBoard now works in Colab notebooks
  - https://www.tensorflow.org/tensorboard/r2/get_started

Google TensorFlow Graph Execution

Evaluation and prediction

- Evaluate (tf.keras.Model.evaluate)
  - https://www.tensorflow.org/beta/guide/keras/training_and_evaluation
  - Applies the model to the validation inputs and computes the loss between the true and predicted outputs
  - Tracks loss and metrics (so it doesn't do gradient back prop and weight update)

- Predict (tf.keras.Model.predict)
  - https://www.tensorflow.org/beta/guide/keras/training_and_evaluation
  - Generate predicted outputs from test inputs (so it doesn't do loss eval, gradient back prop and weight update)

Google TensorFlow Graph Execution

Runtime (large, training optimized)

- Calls to fit, evaluate and predict execute graphs
  - https://www.tensorflow.org/guide/extend/architecture
  - http://public.kevinrobinsonblog.com/docs/A%20tour%20through%20the%20TensorFlow%20codebase%20-%20v4.pdf
- Client
  - Defines the computation as a dataflow graph
  - Initiates graph execution using a session (or Keras API equivalent)
- Distributed master
  - Prunes a specific subgraph from the graph based on the arguments to Session.run()
  - Partitions the subgraph into multiple pieces that run in different processes and devices
  - Distributes the graph pieces to worker services
  - Initiates graph piece execution by worker services
- Worker services (1 for each task)
  - Schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, ...)
  - Send and receive operation results to and from other worker services
- Kernel Implementations
  - Perform the computation for individual graph operations

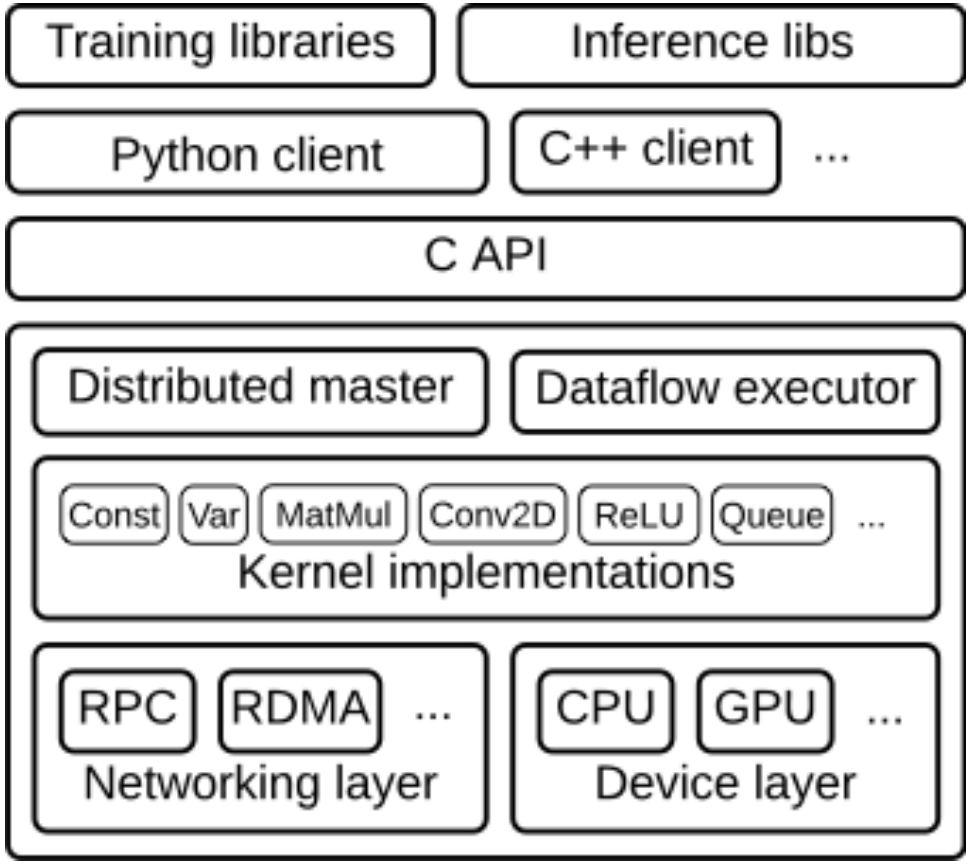


Figure from https://www.tensorflow.org/guide/extend/architecture

Google TensorFlow Graph Execution

Runtime (large, training optimized)

- Calls to fit, evaluate and predict execute graphs
  - https://www.tensorflow.org/guide/extend/architecture
  - http://public.kevinrobinsonblog.com/docs/A%20tour%20through%20the%20TensorFlow%20codebase%20-%20v4.pdf
- Client
  - Defines the computation as a dataflow graph
  - Initiates graph execution using a session (or Keras API equivalent)
- Distributed master
  - Prunes a specific subgraph from the graph based on the arguments to Session.run()
  - Partitions the subgraph into multiple pieces that run in different processes and devices
  - Distributes the graph pieces to worker services
  - Initiates graph piece execution by worker services
- Worker services (1 for each task)
  - Schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, ...)
  - Send and receive operation results to and from other worker services
- Kernel Implementations
  - Perform the computation for individual graph operations

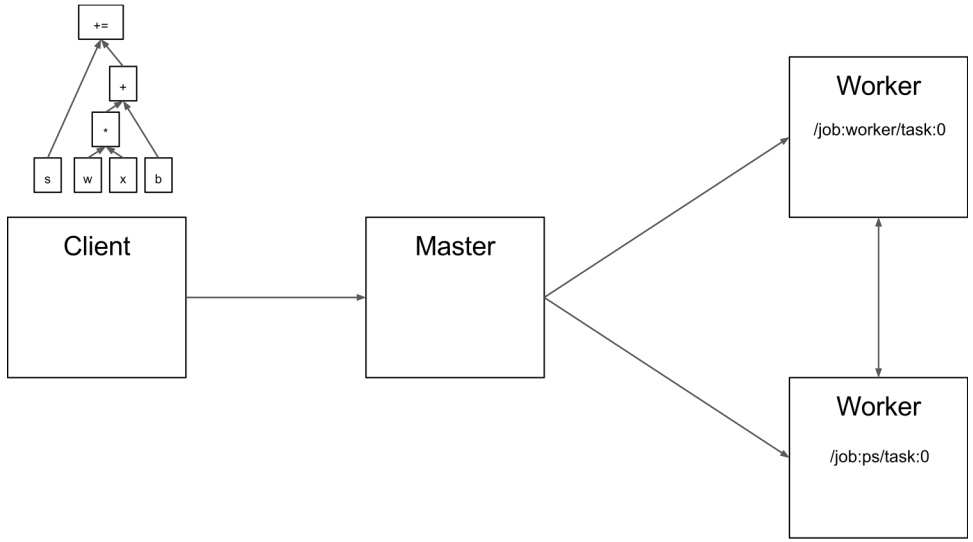


Figure from https://www.tensorflow.org/guide/extend/architecture

Google TensorFlow Graph Execution

Runtime (large, training optimized)

- Calls to fit, evaluate and predict execute graphs
  - https://www.tensorflow.org/guide/extend/architecture
  - http://public.kevinrobinsonblog.com/docs/A%20tour%20through%20the%20TensorFlow%20codebase%20-%20v4.pdf
- Client
  - Defines the computation as a dataflow graph
  - Initiates graph execution using a session (or Keras API equivalent)
- Distributed master
  - Prunes a specific subgraph from the graph based on the arguments to Session.run()
  - Partitions the subgraph into multiple pieces that run in different processes and devices
  - Distributes the graph pieces to worker services
  - Initiates graph piece execution by worker services
- Worker services (1 for each task)
  - Schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, ...)
  - Send and receive operation results to and from other worker services
- Kernel Implementations
  - Perform the computation for individual graph operations

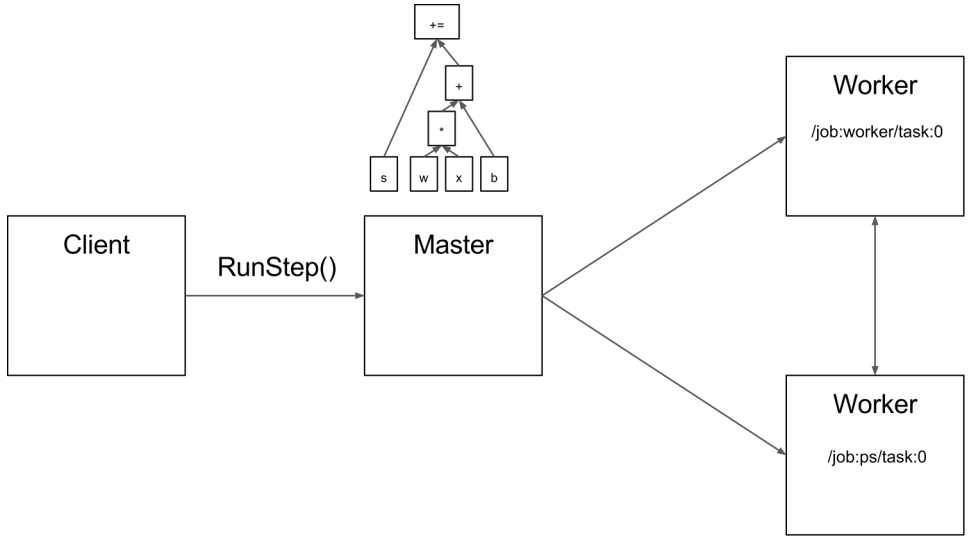


Figure from https://www.tensorflow.org/guide/extend/architecture

Google TensorFlow Graph Execution

Runtime (large, training optimized)

- Calls to fit, evaluate and predict execute graphs
  - https://www.tensorflow.org/guide/extend/architecture
  - http://public.kevinrobinsonblog.com/docs/A%20tour%20through%20the%20TensorFlow%20codebase%20-%20v4.pdf
- Client
  - Defines the computation as a dataflow graph
  - Initiates graph execution using a session (or Keras API equivalent)
- Distributed master
  - Prunes a specific subgraph from the graph based on the arguments to Session.run()
  - Partitions the subgraph into multiple pieces that run in different processes and devices
  - Distributes the graph pieces to worker services
  - Initiates graph piece execution by worker services
- Worker services (1 for each task)
  - Schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, ...)
  - Send and receive operation results to and from other worker services
- Kernel Implementations
  - Perform the computation for individual graph operations

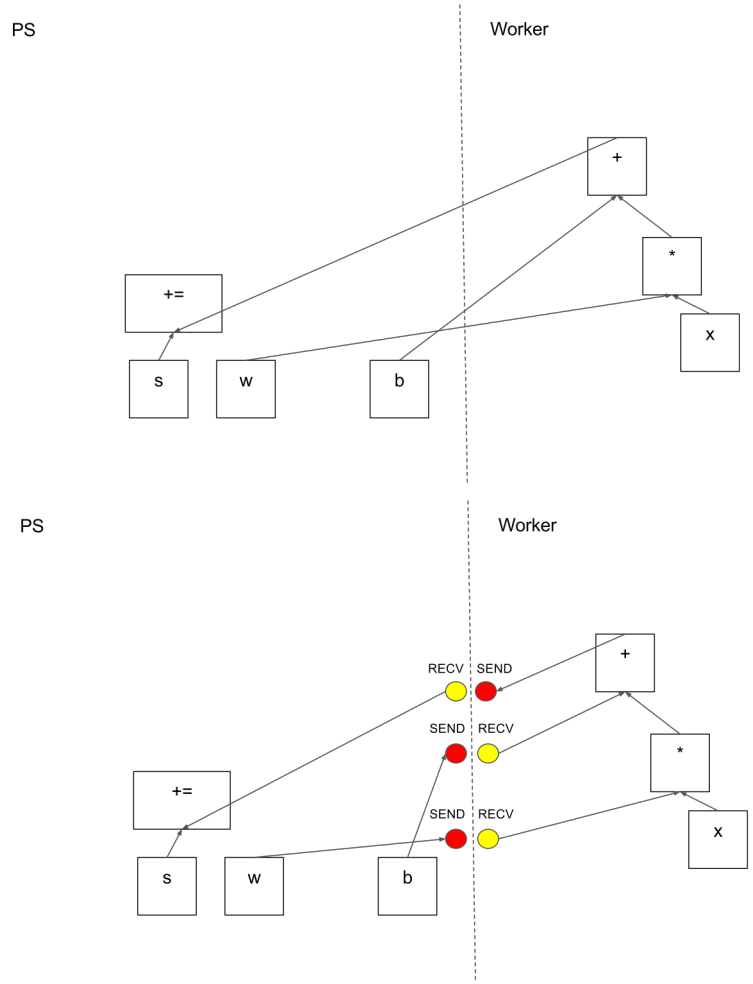


Figure from https://www.tensorflow.org/guide/extend/architecture

Google TensorFlow Graph Execution

Runtime (large, training optimized)

- Calls to fit, evaluate and predict execute graphs
  - https://www.tensorflow.org/guide/extend/architecture
  - http://public.kevinrobinsonblog.com/docs/A%20tour%20through%20the%20TensorFlow%20codebase%20-%20v4.pdf
- Client
  - Defines the computation as a dataflow graph
  - Initiates graph execution using a session (or Keras API equivalent)
- Distributed master
  - Prunes a specific subgraph from the graph based on the arguments to Session.run()
  - Partitions the subgraph into multiple pieces that run in different processes and devices
  - Distributes the graph pieces to worker services
  - Initiates graph piece execution by worker services
- Worker services (1 for each task)
  - Schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, ...)
  - Send and receive operation results to and from other worker services
- Kernel Implementations
  - Perform the computation for individual graph operations

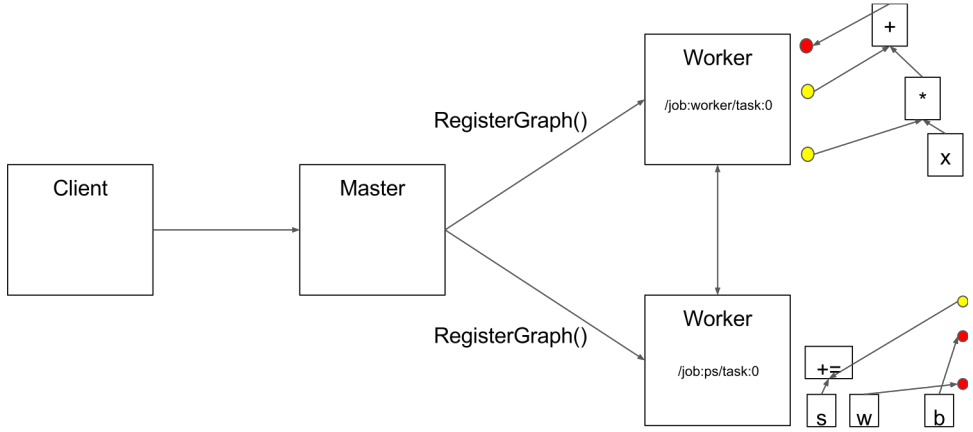


Figure from https://www.tensorflow.org/guide/extend/architecture

Google TensorFlow Graph Execution

Runtime (large, training optimized)

- Calls to fit, evaluate and predict execute graphs
  - https://www.tensorflow.org/guide/extend/architecture
  - http://public.kevinrobinsonblog.com/docs/A%20tour%20through%20the%20TensorFlow%20codebase%20-%20v4.pdf
- Client
  - Defines the computation as a dataflow graph
  - Initiates graph execution using a session (or Keras API equivalent)
- Distributed master
  - Prunes a specific subgraph from the graph based on the arguments to Session.run()
  - Partitions the subgraph into multiple pieces that run in different processes and devices
  - Distributes the graph pieces to worker services
  - Initiates graph piece execution by worker services
- Worker services (1 for each task)
  - Schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, ...)
  - Send and receive operation results to and from other worker services
- Kernel Implementations
  - Perform the computation for individual graph operations

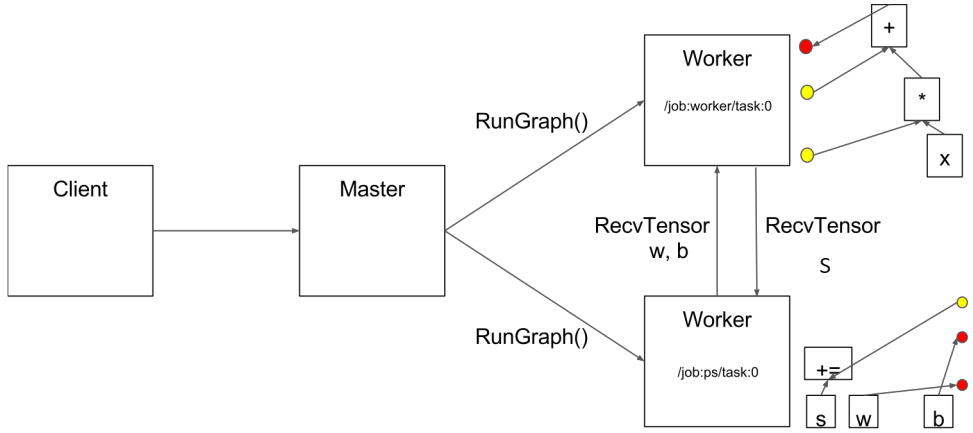


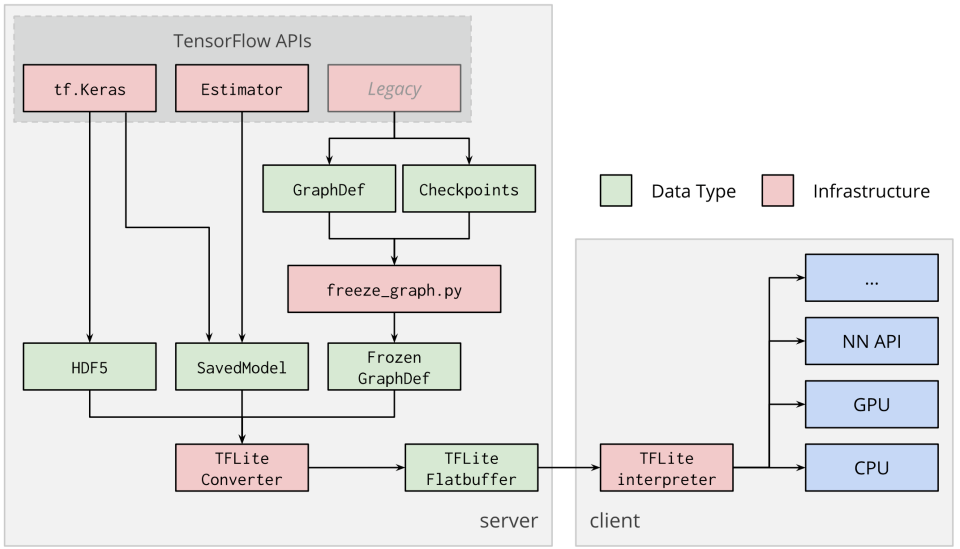
Figure from https://www.tensorflow.org/guide/extend/architecture

Google TensorFlow Lite Graph Execution

Runtime (small, inference optimized)

- TensorFlow Lite is Google's open source framework for using (inference) machine learning models on embedded devices
  - Takes around 200 kB of memory
  - https://www.tensorflow.org/lite
  - https://www.tensorflow.org/lite/guide
  - https://www.tensorflow.org/lite/guide/roadmap

- Flow
  - 1. Create and train a TensorFlow model using supported operators
  - 2. Use the TensorFlow Lite Converter to convert the TensorFlow model to a TensorFlow Lite model
    - https://www.tensorflow.org/lite/convert/index
  - 3. Run the TensorFlow Lite model using the TensorFlow Lite Interpreter
    - https://www.tensorflow.org/lite/guide/inference



TensorFlow 1.x → TensorFlow Lite

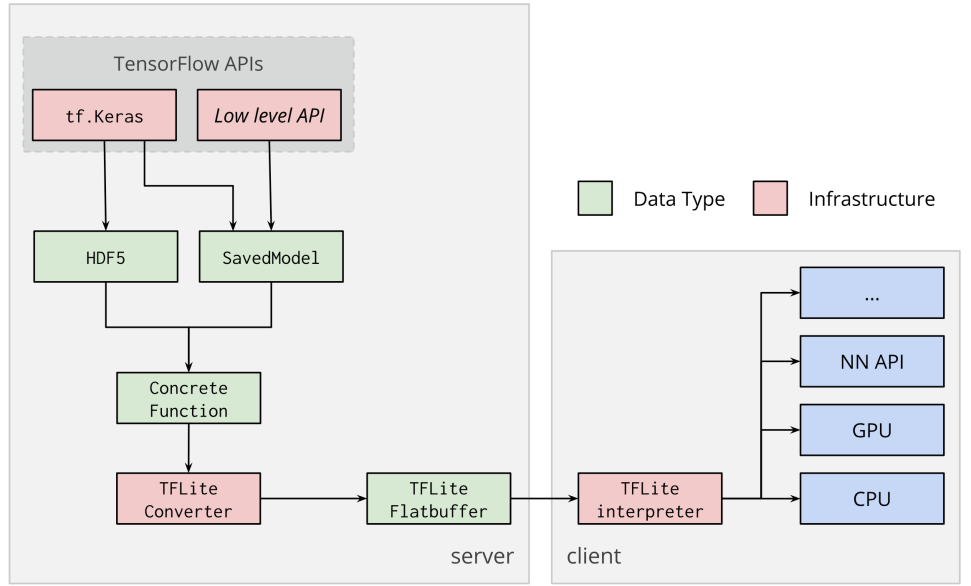
Figure from https://www.tensorflow.org/lite/convert

Google TensorFlow Lite Graph Execution

Runtime (small, inference optimized)

- TensorFlow Lite is Google's open source framework for using (inference) machine learning models on embedded devices
  - Takes around 200 kB of memory
  - https://www.tensorflow.org/lite
  - https://www.tensorflow.org/lite/guide
  - https://www.tensorflow.org/lite/guide/roadmap

- Flow
  - 1. Create and train a TensorFlow model using supported operators
  - 2. Use the TensorFlow Lite Converter to convert the TensorFlow model to a TensorFlow Lite model
    - https://www.tensorflow.org/lite/convert/index
  - 3. Run the TensorFlow Lite model using the TensorFlow Lite Interpreter
    - https://www.tensorflow.org/lite/guide/inference



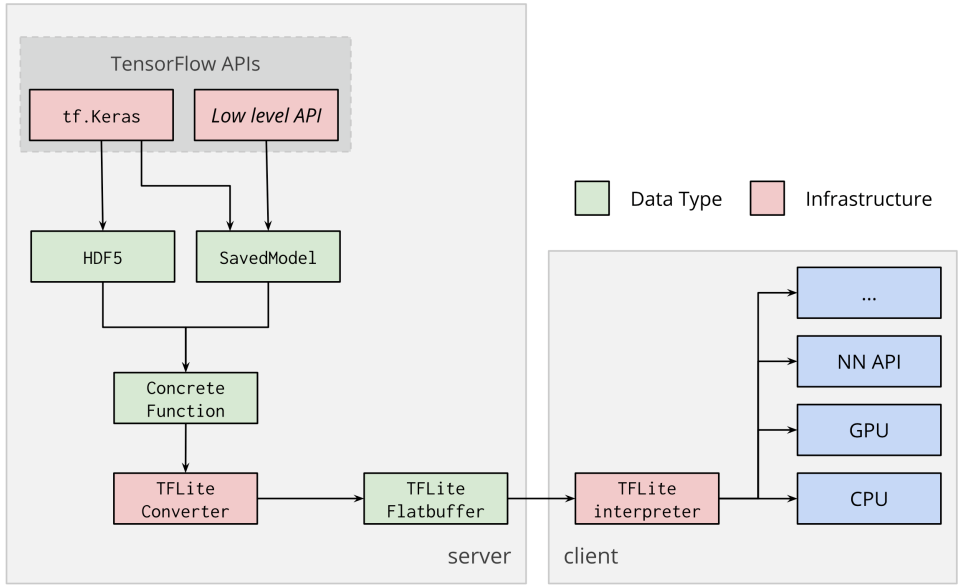
TensorFlow 2.x → TensorFlow Lite

Figure from https://www.tensorflow.org/lite/r2/convert

Google TensorFlow Lite Graph Execution

Runtime (small, inference optimized)

- Standard operators
  - Select TensorFlow ops supported out of the box
  - https://www.tensorflow.org/lite/guide/ops_compatibility
- Custom operators
  - Can be added
  - https://www.tensorflow.org/lite/guide/ops_custom
  - Require 4 functions with a C++ interface
    - Init() is called once at the beginning for every node on the graph and used to initialize variables
    - Free() is called at the end and used to free up space
    - Prepare() is called anytime input tensors are resized
    - Eval() is called at inference
  - Use global registration add the custom op to the built in op resolver
  - Example here
    - https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/kernels/conv.cc



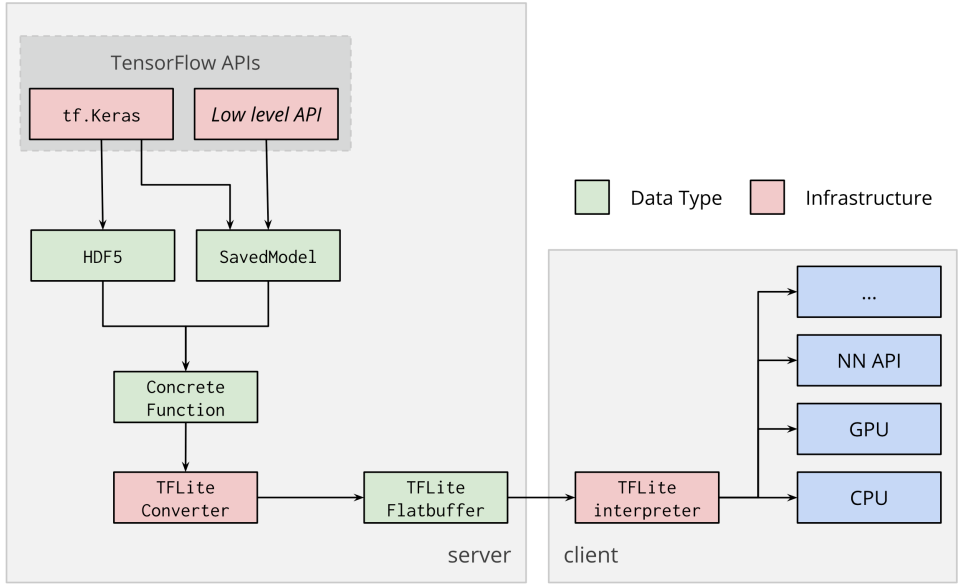
TensorFlow 2.x → TensorFlow Lite

Figure from https://www.tensorflow.org/lite/r2/convert

Google TensorFlow Lite Graph Execution

Runtime (small, inference optimized)

- Model training and conversion
  - Training
    - Quantization aware training / fake quantization (optional)
      - https://github.com/tensorflow/tensorflow/tree/r1.13/tensorflow/contrib/quantize
  - Trained TensorFlow model
    - Format can be SavedModels, Frozen GraphDef, Keras HDF5, tf.Session graph
    - In TensorFlow 2.x the default earer execution requires saving to a graph or creating a concrete function
  - Optimization (optional)
    - Model Optimization Toolkit
      - https://www.tensorflow.org/lite/performance/model_optimization
    - Quantization post training
      - https://medium.com/tensorflow/introducing-the-model-optimization-toolkit-for-tensorflow-254aca1ba0a3
      - https://www.tensorflow.org/lite/performance/post_training_quantization
    - Future support: pruning
      - https://medium.com/tensorflow/tensorflow-model-optimization-toolkit-pruning-api-42cac9157a6a?linkId=67380711
    - Future support: topology modification
  - Visualization
    - Graphviz in 1.x and visualize.py in 2.x
  - Converted TensorFlow Lite model
    - Format is TensorFlow Lite FlatBuffer



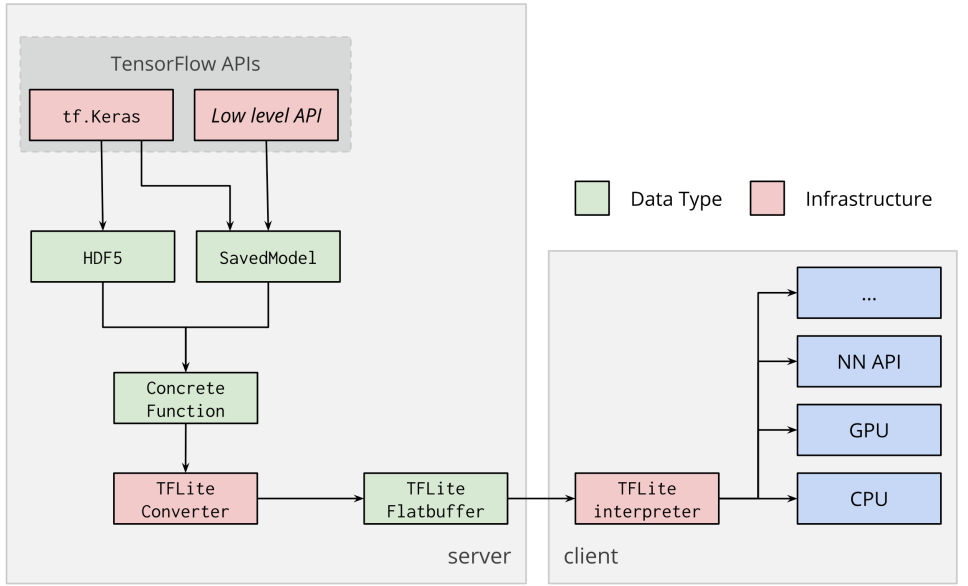
TensorFlow 2.x → TensorFlow Lite

Figure from https://www.tensorflow.org/lite/r2/convert

Google TensorFlow Lite Graph Execution

Runtime (small, inference optimized)

- Inference
  - Load the TensorFlow Lite FlatBuffer format model using the C++ or Python API
  - Build an Interpreter from the FlatBuffer format model
  - Resize inputs if necessary
  - Allocate memory
  - Set inputs
  - Perform inference
  - Read outputs



TensorFlow 2.x → TensorFlow Lite

Figure from https://www.tensorflow.org/lite/r2/convert

Google TensorFlow Lite Acceleration

Acceleration for Android

- Android NDK allows developers to implement parts of their Android apps in C/C++ or other languages
- The Android Neural Network API is a C API that can be used by runtimes such as TensorFlow Lite to offload operations on Android devices
  - Note that it can also be used directly to define models, though this is expected to be less common
- The Android Neural Network Runtime distributes these operations to various processors based on their capabilities with the CPU used as the fallback
- The Android Neural Network HAL allows processor vendors to connect specialized accelerators to the runtime

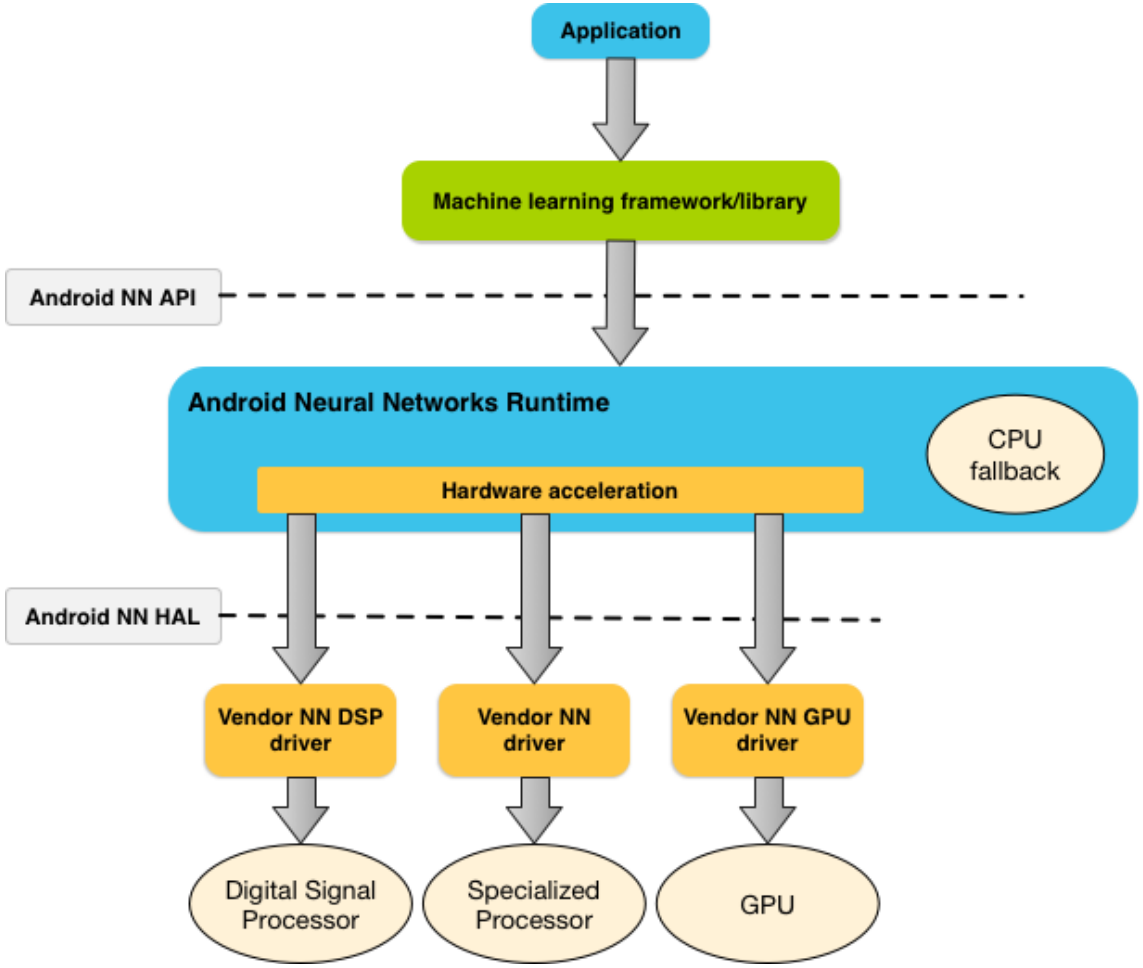


Figure from https://developer.android.com/ndk/guides/neuralnetworks/

Google TensorFlow Lite Acceleration

Acceleration for Android

- Model definition
  - Create a model instance
  - Add operands to the model specifying type, shape and quantization parameters (if applicable); for constants such as weights and biases specify their values at this point in time
  - Add operations to the model specifying type, input operands and output operands
  - Mark input and output operands for the model; these locations will be provided during execution
- Model compilation
  - Create a compilation instance
  - Set optional optimization preferences for power, single result time or throughput time
  - Compile; this determines where operations are run and allows hardware drivers to prepare for execution
- Model execution
  - Create an execution instance
  - Specify input and output operand locations
  - Start execution
  - Wait
  - Repeat for new inputs and outputs

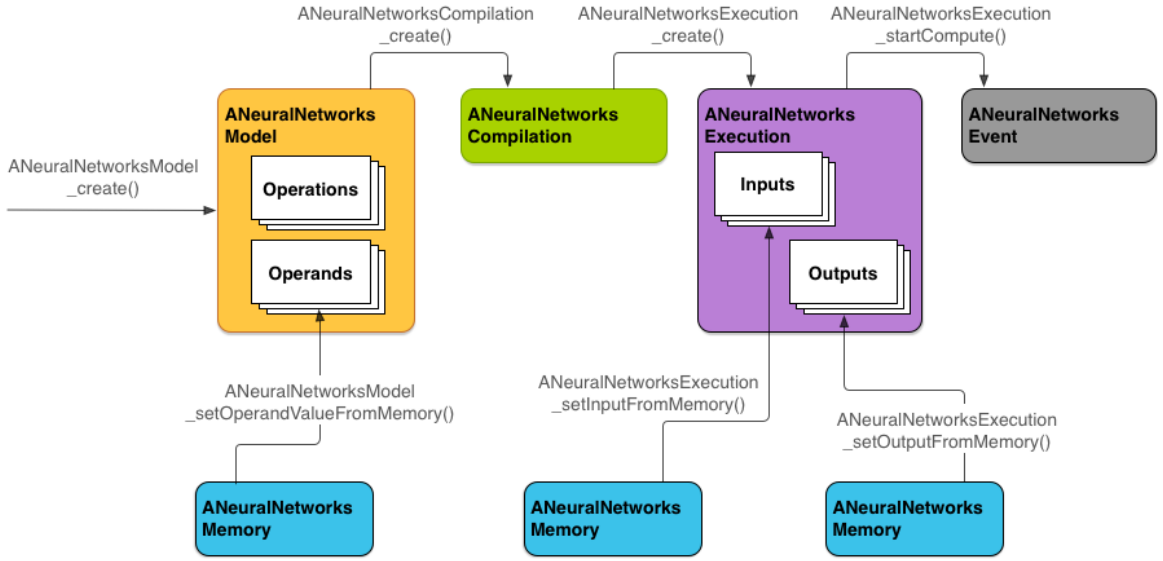


Figure from https://developer.android.com/ndk/guides/neuralnetworks/

Google TensorFlow Lite MCU Graph Execution

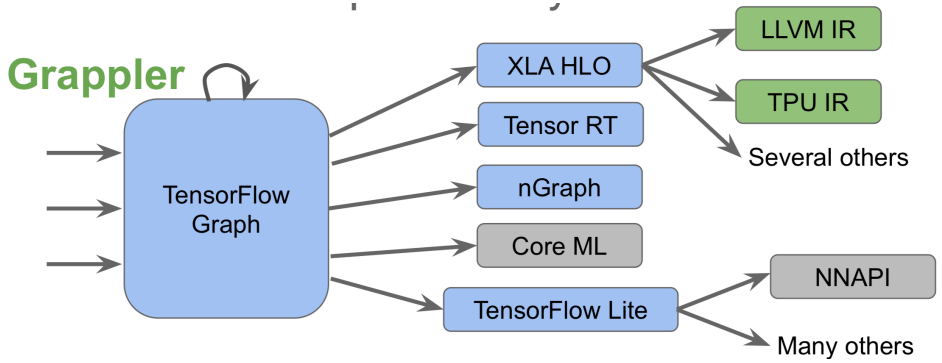
Runtime (very small, inference optimized)

- TensorFlow Lite Microcontroller is Google's open source framework for using (inference) machine learning models on very small embedded devices
  - Takes around 20 kB of memory (vs ~ 200 kB for TensorFlow Lite)
  - Does not require a high level OS, C / C++ standard libraries or dynamic memory allocation
  - https://www.tensorflow.org/lite/microcontrollers/overview
  - https://www.tensorflow.org/lite/microcontrollers/get_started
  - https://www.tensorflow.org/lite/microcontrollers/build_convert
  - https://www.tensorflow.org/lite/microcontrollers/library
  - https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/experimental/micro

- Flow
  - 1. Create or obtain a TensorFlow model (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/experimental/micro/kernels/all_ops_resolver.cc)
  - 2. Convert the model to a TensorFlow Lite FlatBuffer
  - 3. Convert the FlatBuffer to a C byte array
  - 4. Integrate the TensorFlow Lite for Microcontrollers C++ library
  - 5. Deploy to your device

Google Common Graph IR

- MLIR ~ multi level IR
  - Many graph IRs currently exist within the TensorFlow ecosystem
  - Duplication of work and complications
  - Goal of MLIR is to use SSA based design to generalize and improve graphs at all levels
  - Many similarities to LLVM



- MLIR: a new intermediate representation and compiler framework
  - https://medium.com/tensorflow/mlir-a-new-intermediate-representation-and-compiler-framework-beba999ed18d
- Multi-Level intermediate representation overview
  - https://github.com/tensorflow/mlir
- MLIR primer: a compiler infrastructure for the end of Moore's law
  - https://storage.googleapis.com/pub-tools-public-publication-data/pdf/1c082b766d8e14b54e36e37c9fc3ebbe8b4a72dd.pdf
- MLIR tutorial: building a compiler with MLIR
  - https://llvm.org/devmtg/2019-04/slides/Tutorial-AminiVasilacheZinenko-MLIR.pdf
- MLIR: multi-level intermediate representation compiler infrastructure
  - https://llvm.org/devmtg/2019-04/slides/Keynote-ShpeismanLattner-MLIR.pdf

Figure from http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf

Google Common Graph IR

- MLIR ~ multi level IR
  - Many graph IRs currently exist within the TensorFlow ecosystem
  - Duplication of work and complications
  - Goal of MLIR is to use SSA based design to generalize and improve graphs at all levels
  - Many similarities to LLVM

- MLIR: a new intermediate representation and compiler framework
  - https://medium.com/tensorflow/mlir-a-new-intermediate-representation-and-compiler-framework-beba999ed18d
- Multi-Level intermediate representation overview
  - https://github.com/tensorflow/mlir
- MLIR primer: a compiler infrastructure for the end of Moore's law
  - https://storage.googleapis.com/pub-tools-public-publication-data/pdf/1c082b766d8e14b54e36e37c9fc3ebbe8b4a72dd.pdf
- MLIR tutorial: building a compiler with MLIR
  - https://llvm.org/devmtg/2019-04/slides/Tutorial-AminiVasilacheZinenko-MLIR.pdf
- MLIR: multi-level intermediate representation compiler infrastructure
  - https://llvm.org/devmtg/2019-04/slides/Keynote-ShpeismanLattner-MLIR.pdf

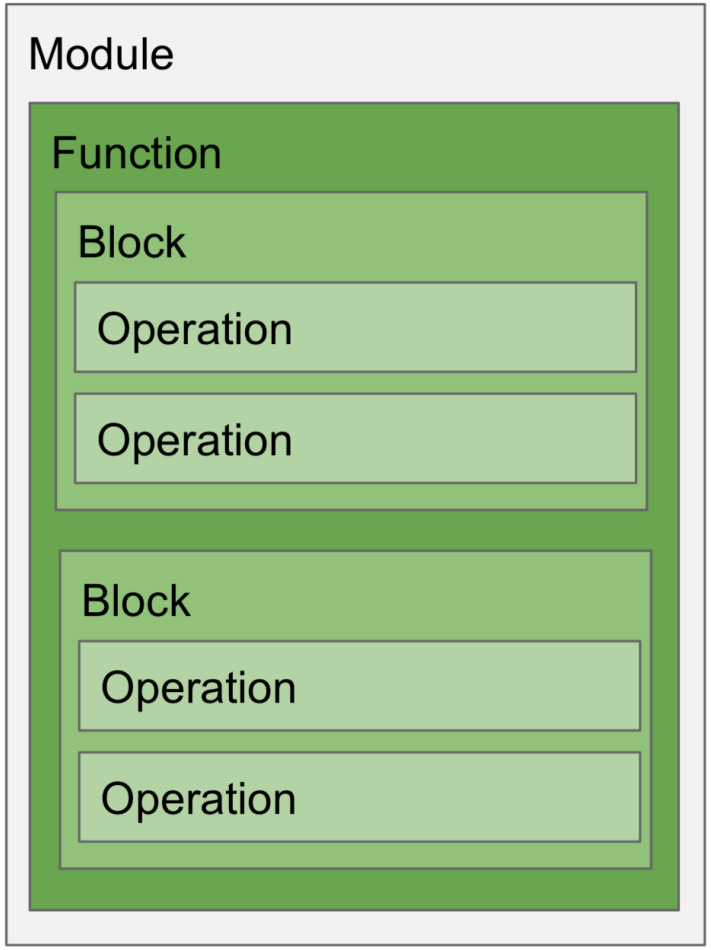


Figure from http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf

Intel nGraph

- A compiler stack designed to enable the connection of multiple front end framework options with multiple back end execution options without requiring a unique back end implementation for each front end framework

- Intel nGraph: an intermediate representation, compiler, and executor for deep learning
  - https://arxiv.org/abs/1801.08058

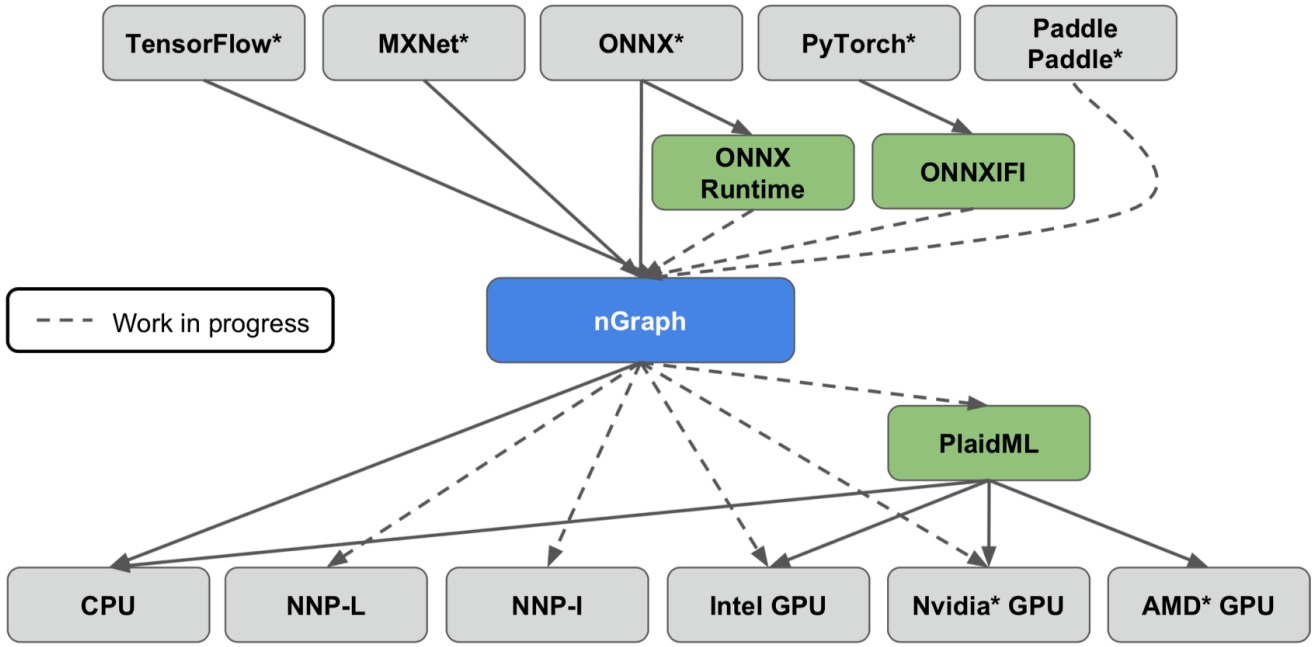


Figure from https://github.com/NervanaSystems/ngraph

Intel nGraph

- A compiler stack designed to enable the connection of multiple front end framework options with multiple back end execution options without requiring a unique back end implementation for each front end framework

- Separate APIs are defined for front end frameworks and back end execution options

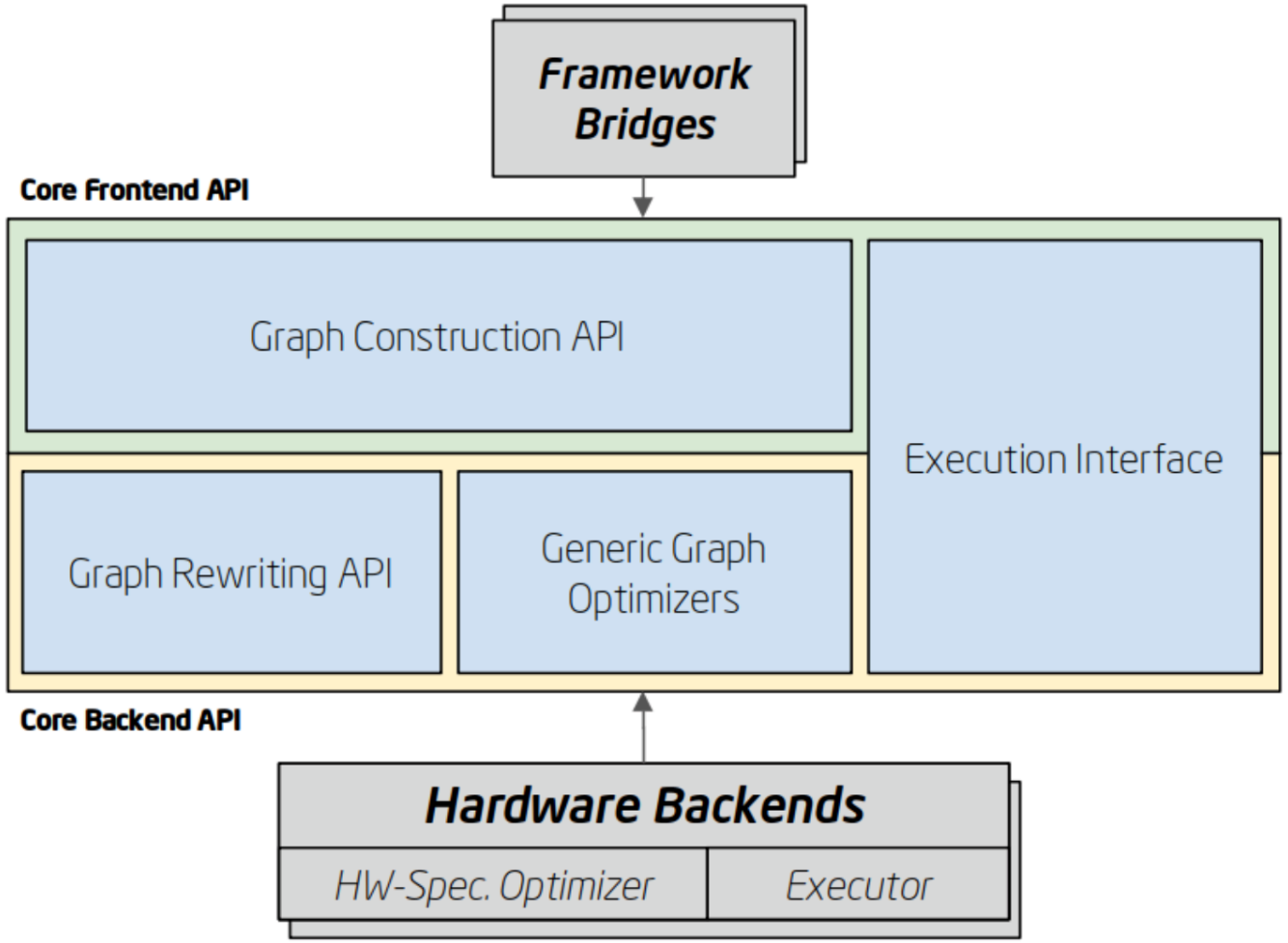


Figure from https://github.com/NervanaSystems/ngraph

Intel nGraph

- A compiler stack designed to enable the connection of multiple front end framework options with multiple back end execution options without requiring a unique back end implementation for each front end framework

- Sub graphs supported by back ends are replaced by single nodes
  - These sub graph can be optimized and implemented in any fashion by the back end compiler
  - They're replaced in the original graph by a single super node
  - Any remaining nodes fall back to the default CPU based implementation provided by the runtime

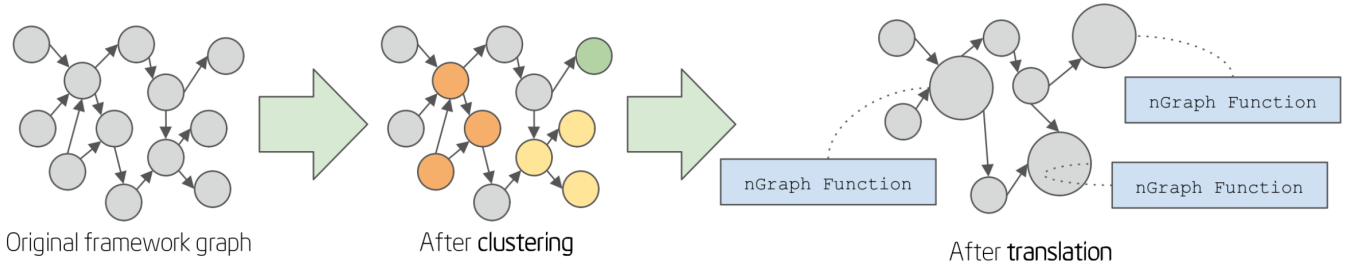


Figure from http://ngraph.nervanasys.com/index.html/core/overview.html

Intel nGraph

- A compiler stack designed to enable the connection of multiple front end framework options with multiple back end execution options without requiring a unique back end implementation for each front end framework

- Sub graphs supported by back ends are replaced by single nodes
  - These sub graph can be optimized and implemented in any fashion by the back end compiler
  - They're replaced in the original graph by a single super node
  - Any remaining nodes fall back to the default CPU based implementation provided by the runtime

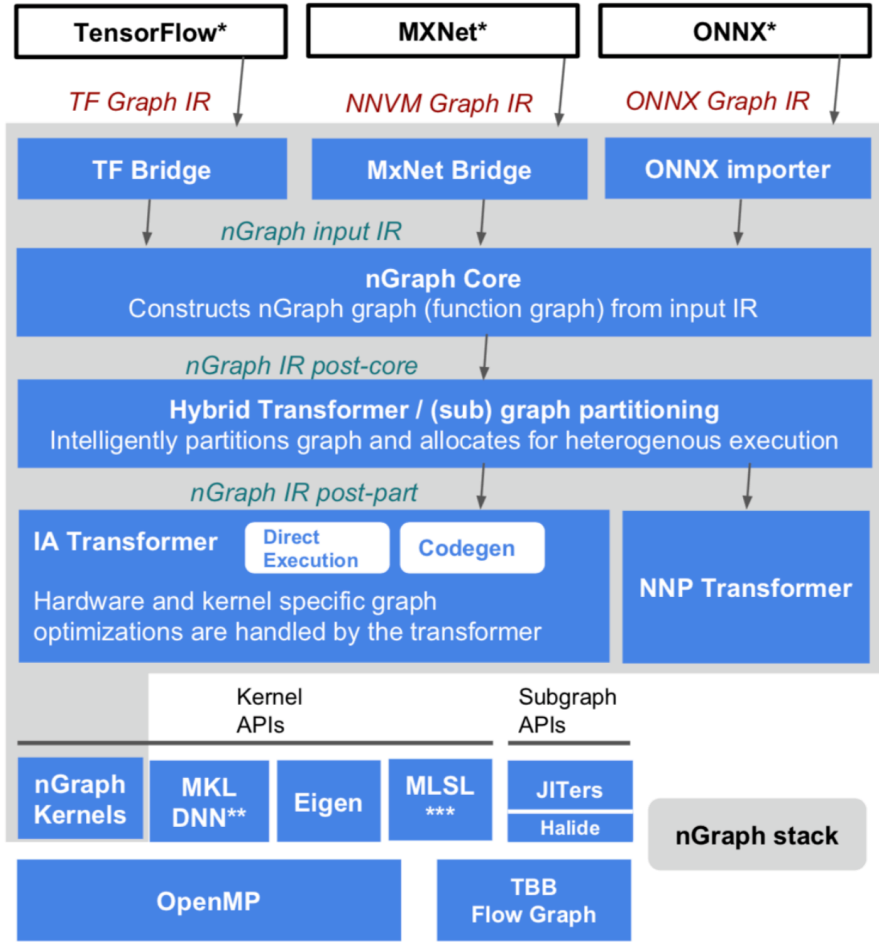


Figure from https://ngraph.nervanasys.com/docs/latest/project/about.html

Nvidia cuDNN

Operator library

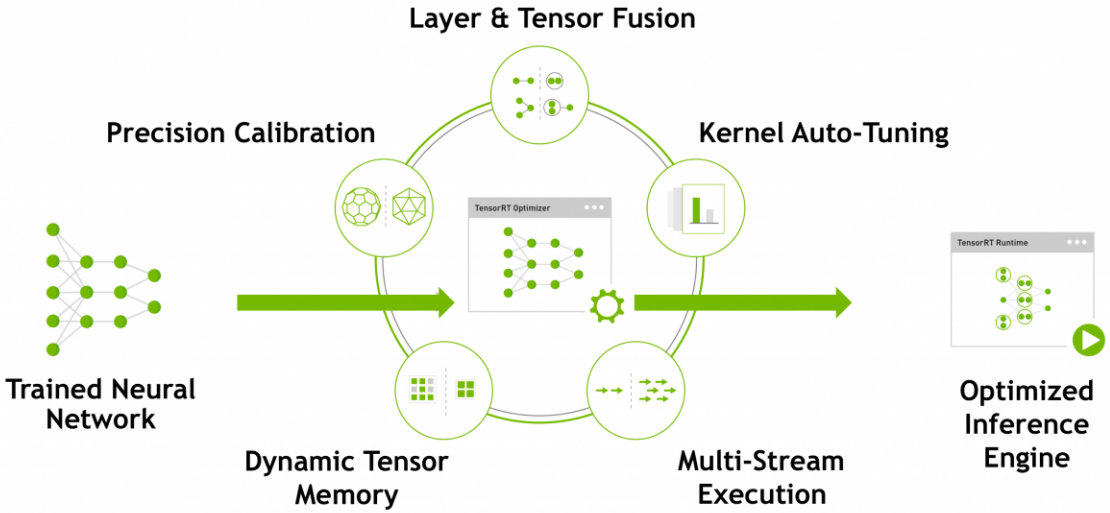
- Widely used by all xNN frameworks for operator acceleration on GPUs
  - cuDNN includes optimized versions of key primitives for xNNs
  - cuBLAS is used for GPU accelerated BLAS
  - NCCL is used for multi GPU communication

- Links
  - https://developer.nvidia.com/cudnn
  - https://docs.nvidia.com/deeplearning/sdk/index.html
  - https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html

Nvidia TensorRT

Inference runtime

- Nvidia TensorRT
  - Nvidia's C++ network optimizer and runtime engine for inference on Nvidia devices
  - Includes TensorFlow, PyTorch, theano, PaddlePaddle, mxnet, ONNX, Caffe and Caffe 2 model parsers
  - Includes C++ and Python APIs for programmatically creating models
  - Targets Nvidia Tesla, Drive, Jetson, NVDLA, ... hardware



- Links
  - https://developer.nvidia.com/tensorrt
  - https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html

Qualcomm Snapdragon NPE SDK

- Neural processing engine SDK
  - Runtime for xNN inference on Qualcomm Snapdragon devices
  - Support for models in Caffe, Caffe 2, ONNX and TensorFlow formats

- Links
  - https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk
  - https://developer.qualcomm.com/docs/snpe/index.html

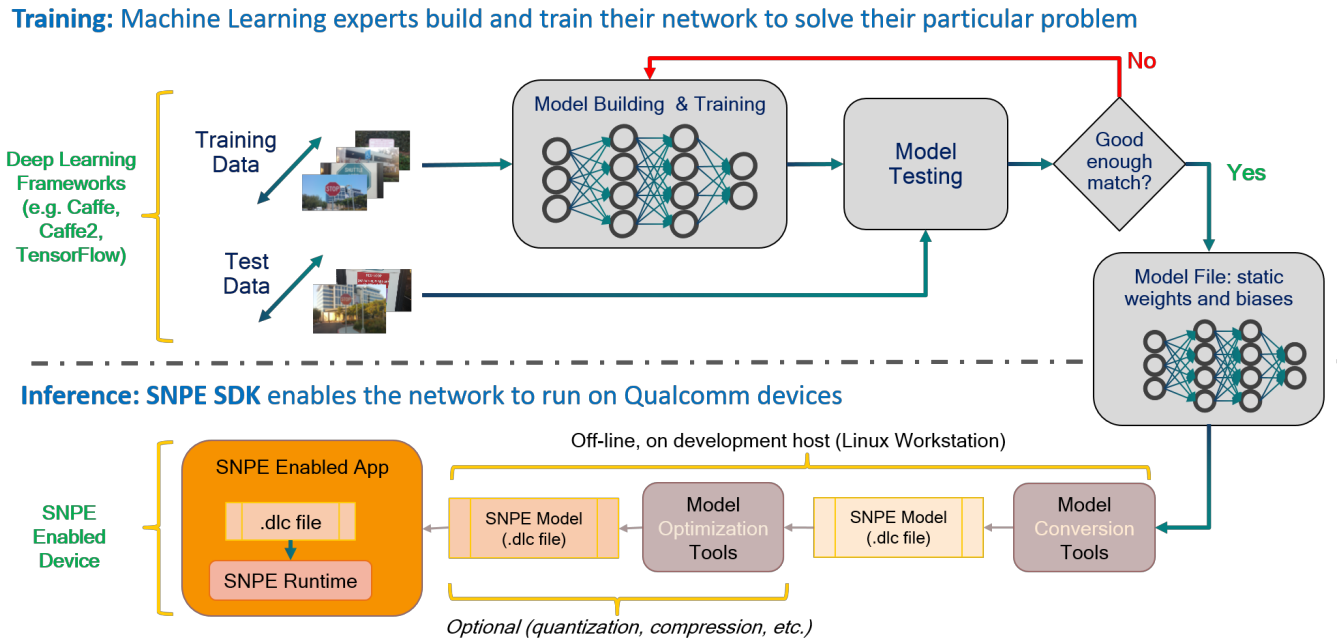


Figure from https://developer.qualcomm.com/docs/snpe/overview.html

UW TVM

- Relay
  - Common high level graph format and opt
    - Pruning, fusion, layout transformation, memory management, ...
  - Registers operators to TVM implementations
  - https://docs.tvm.ai/langref/index.html
- TVM
  - Tensor operator optimization and code gen
    - 1. Tensor expression language to express operators as different program options separating scheduling and hardware intrinsics
    - 2. Automated program optimizer
    - 3. Graph re writer
  - https://tvm.ai and https://arxiv.org/abs/1802.04799
- TOPI: TVM operator inventory
  - Pre made TVM operator recipes
  - https://github.com/dmlc/tvm/tree/master/topi

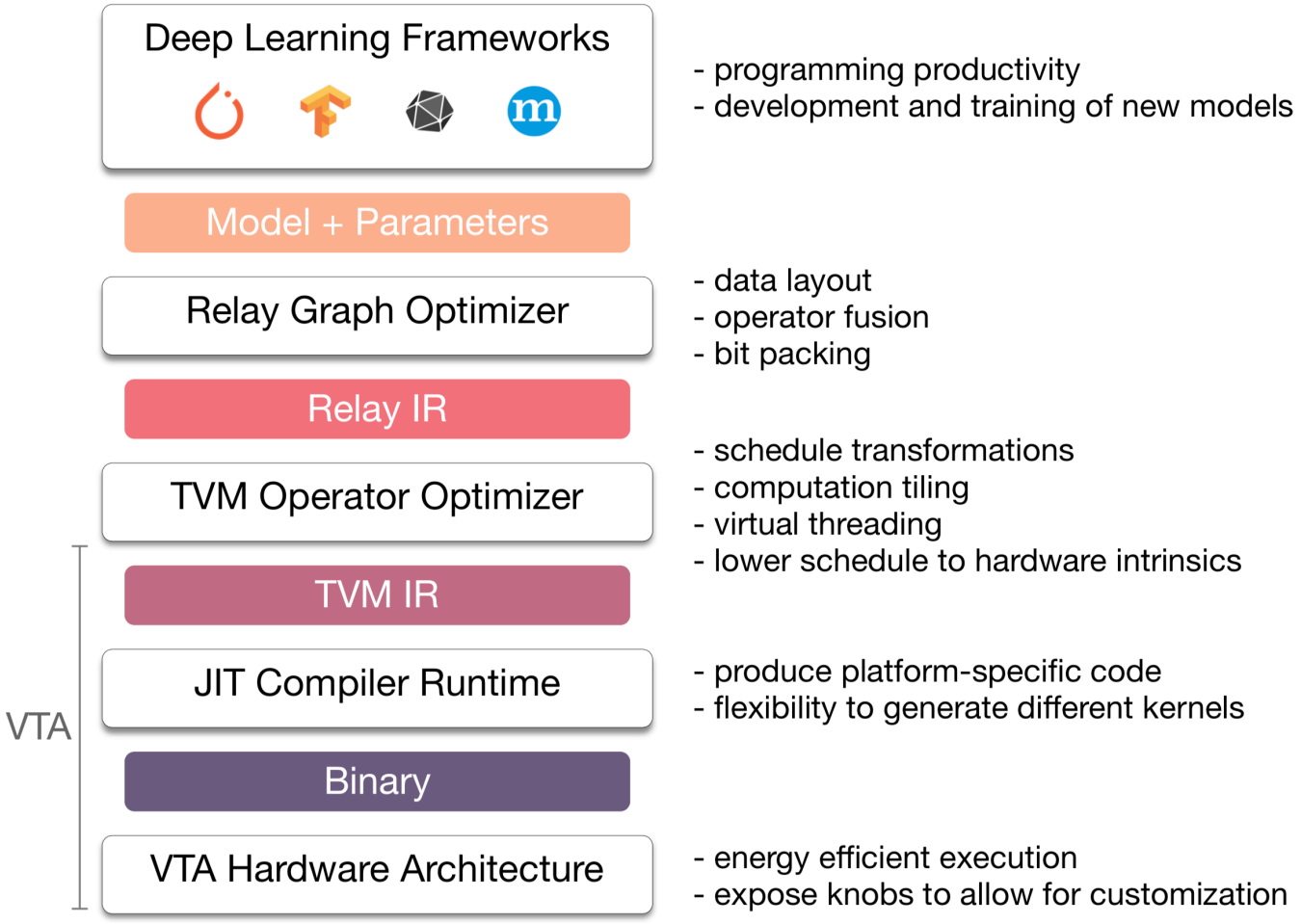


Figure from https://arxiv.org/abs/1807.04188

UW TVM

- Relay
  - Common high level graph format and opt
    - Pruning, fusion, layout transformation, memory management, ...
  - Registers operators to TVM implementations
  - https://docs.tvm.ai/langref/index.html
- TVM
  - Tensor operator optimization and code gen
    - 1. Tensor expression language to express operators as different program options separating scheduling and hardware intrinsics
    - 2. Automated program optimizer
    - 3. Graph re writer
  - https://tvm.ai and https://arxiv.org/abs/1802.04799
- TOPI: TVM operator inventory
  - Pre made TVM operator recipes
  - https://github.com/dmlc/tvm/tree/master/topi

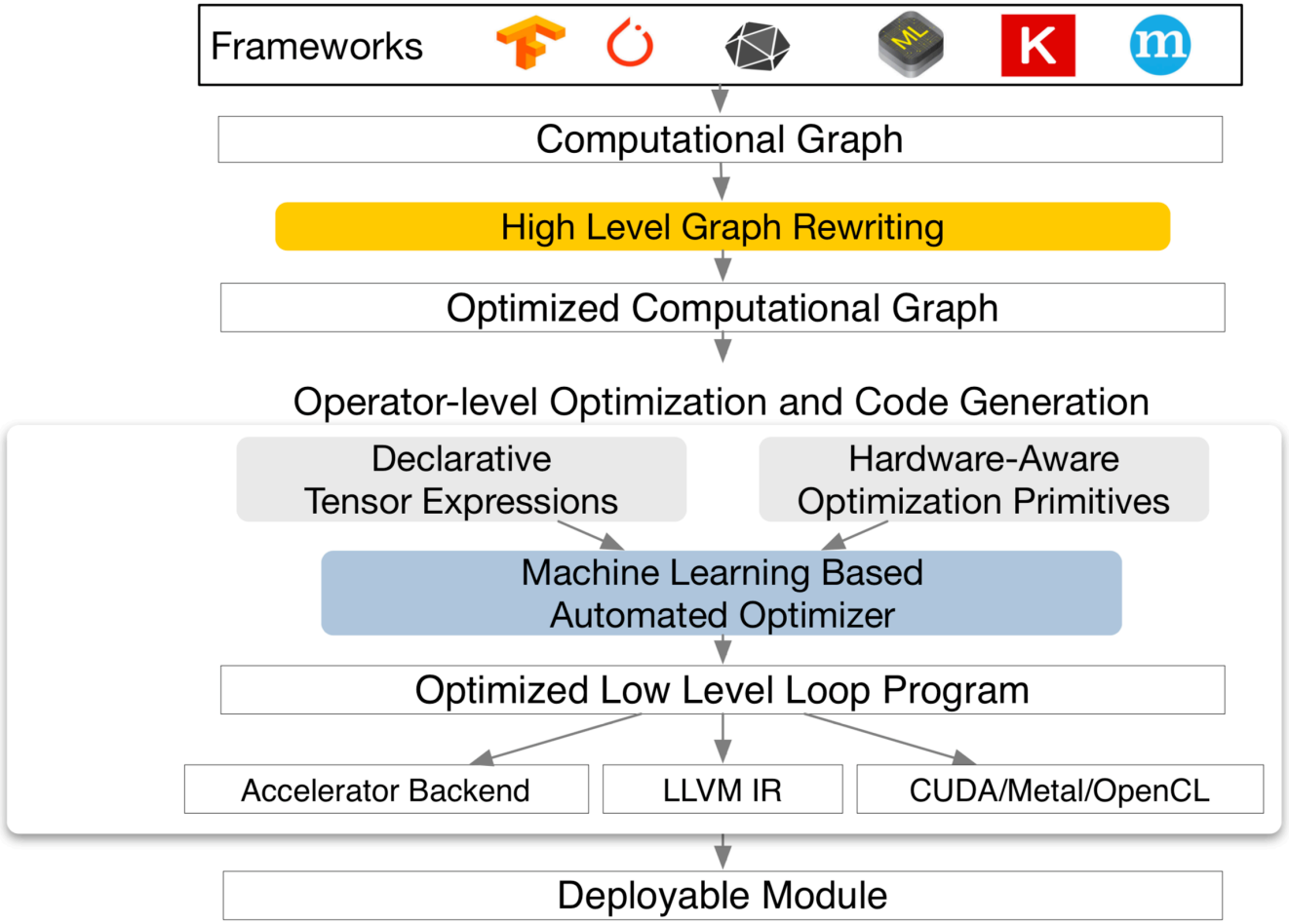


Figure from https://arxiv.org/abs/1802.04799

UW TVM

- Relay
  - Common high level graph format and opt
    - Pruning, fusion, layout transformation, memory management, ...
  - Registers operators to TVM implementations
  - https://docs.tvm.ai/langref/index.html
- TVM
  - Tensor operator optimization and code gen
    - 1. Tensor expression language to express operators as different program options separating scheduling and hardware intrinsics
    - 2. Automated program optimizer
    - 3. Graph re writer
  - https://tvm.ai and https://arxiv.org/abs/1802.04799
- TOPI: TVM operator inventory
  - Pre made TVM operator recipes
  - https://github.com/dmlc/tvm/tree/master/topi

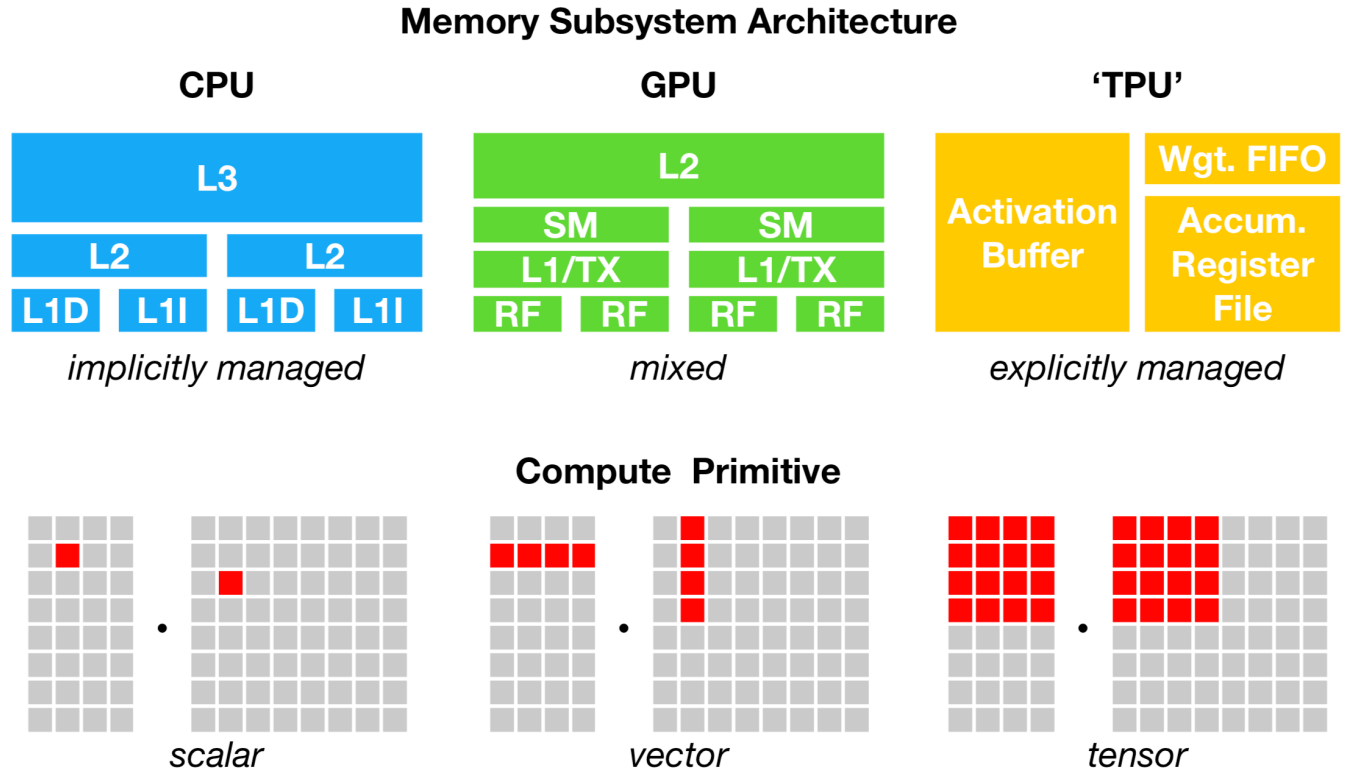


Figure from https://arxiv.org/abs/1802.04799

Xilinx

- Links
  - https://www.xilinx.com/products/design-tools/deephi.html#overview
  - https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

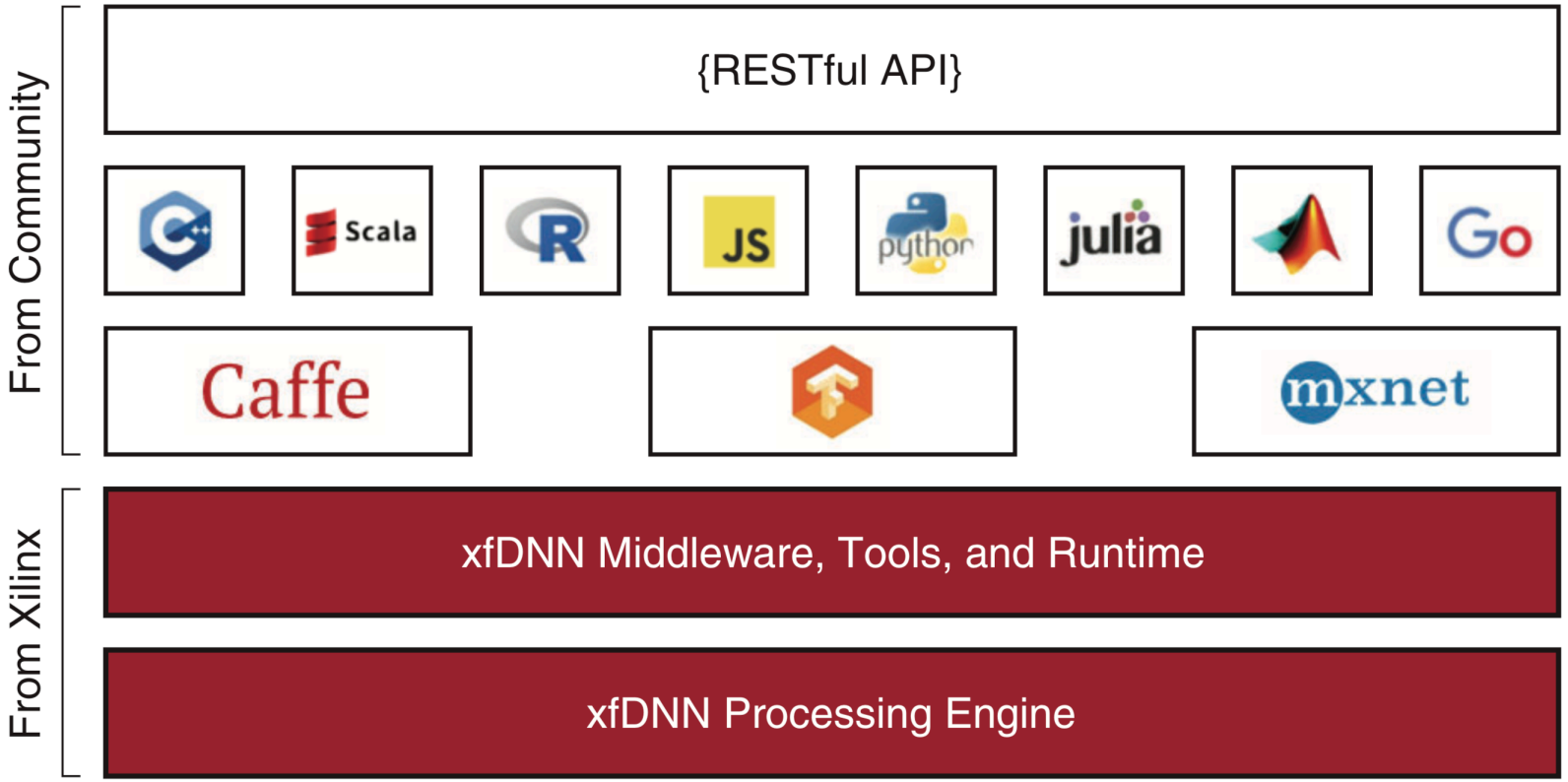
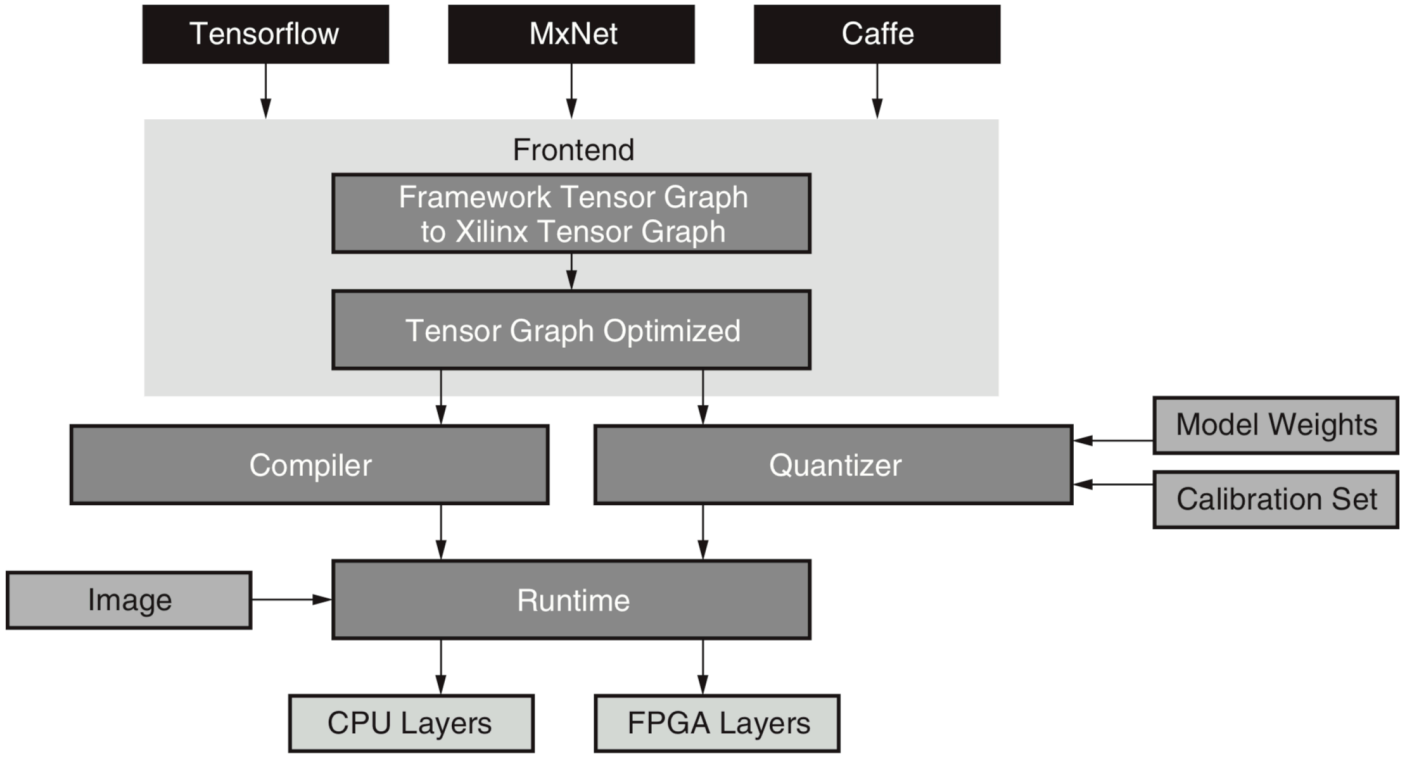


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx



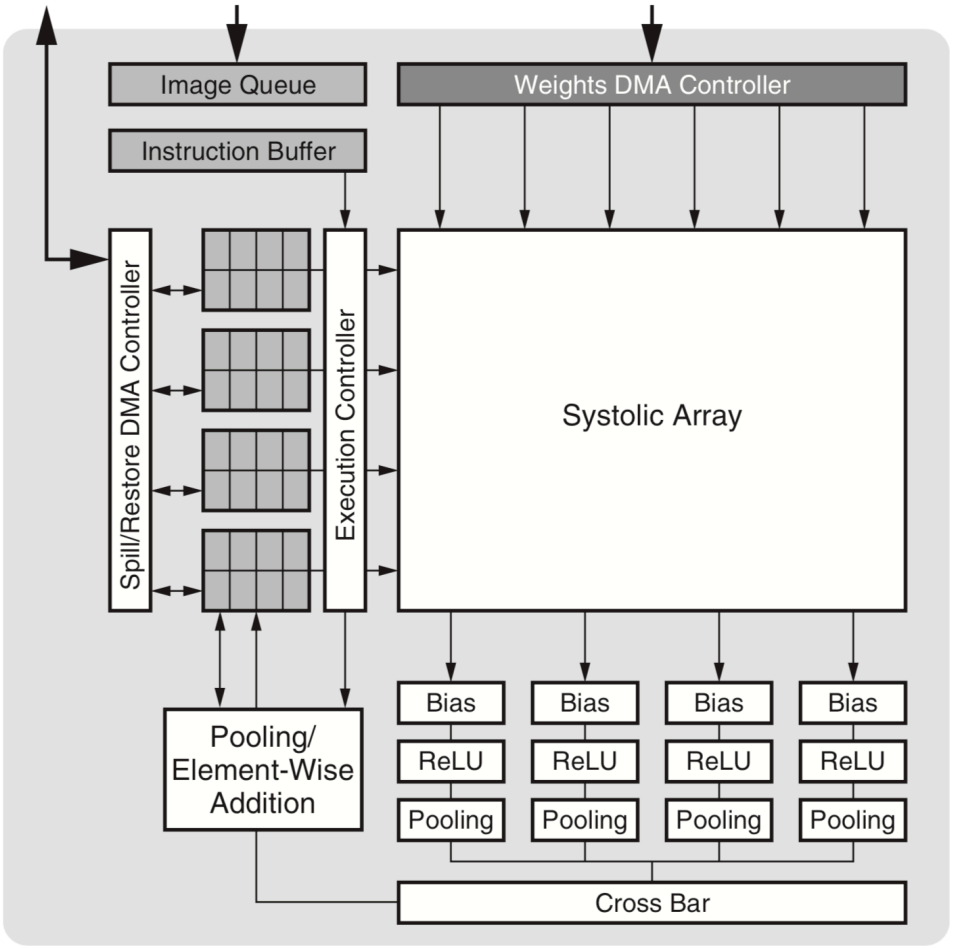


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx

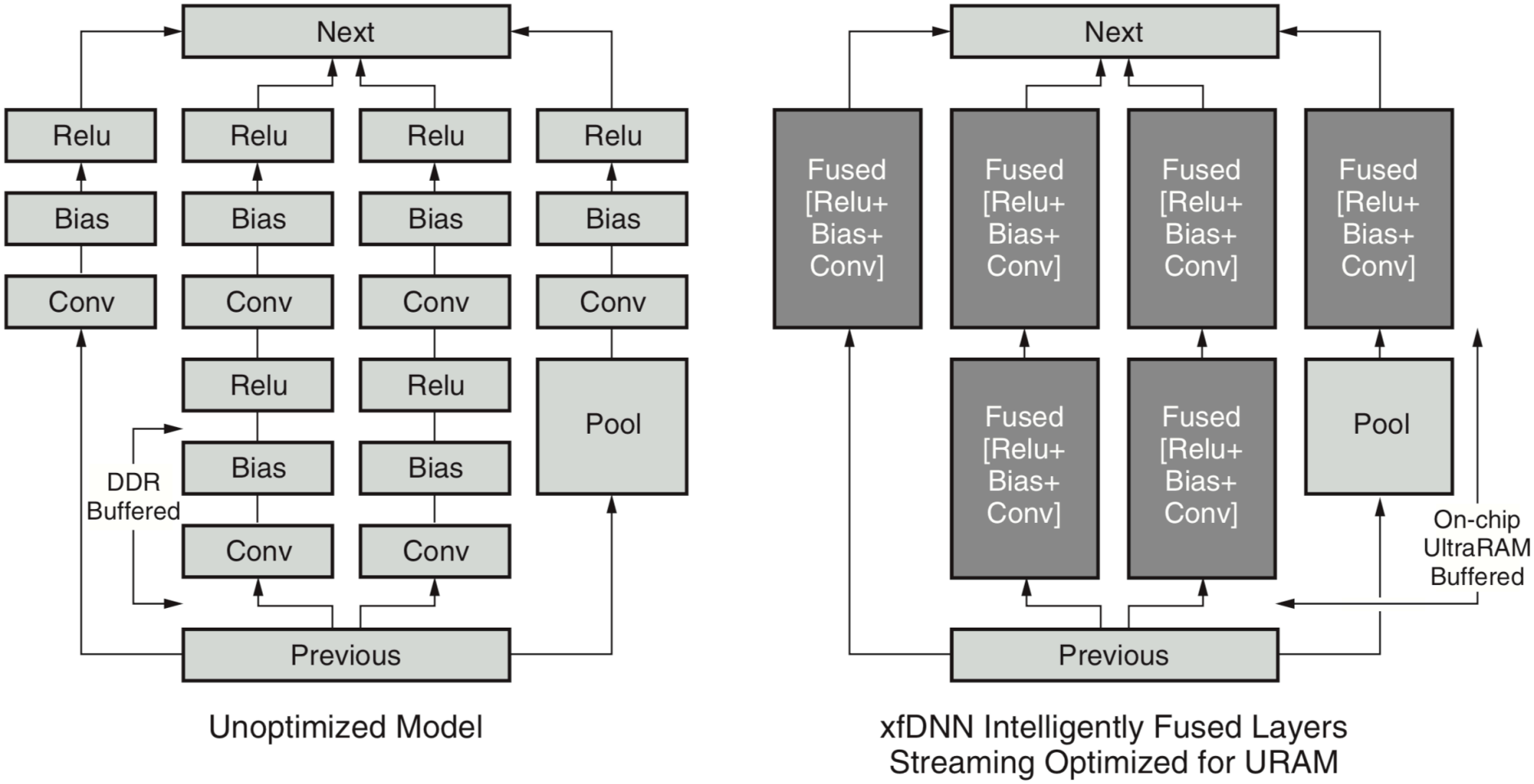
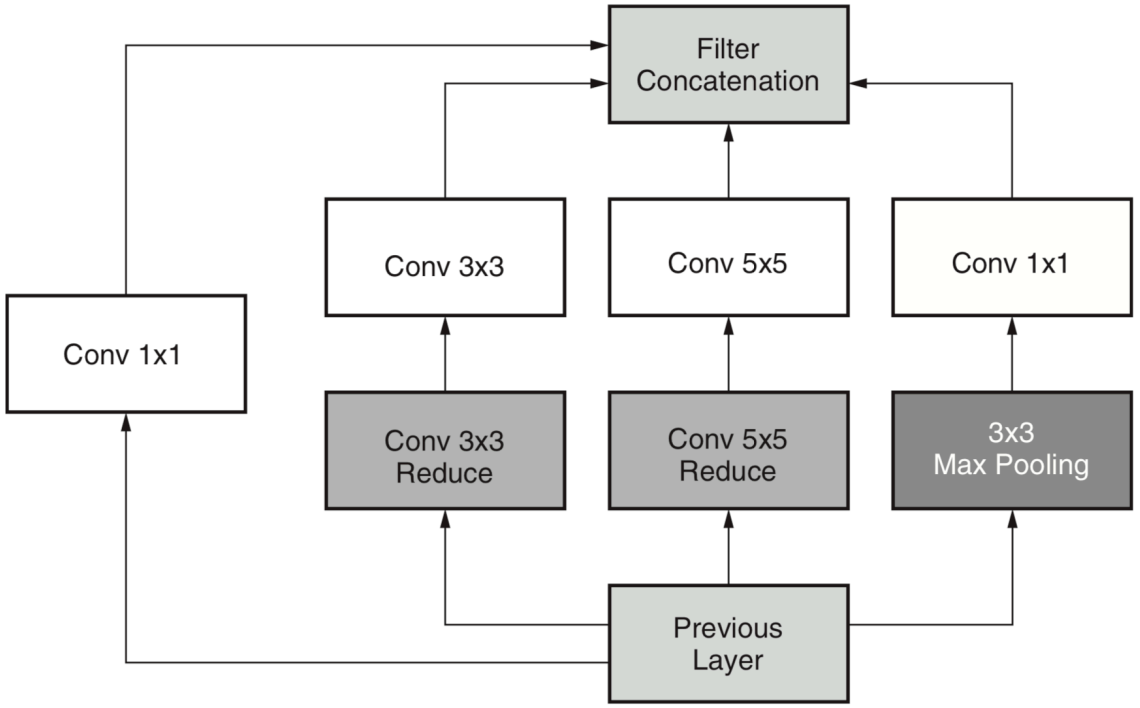
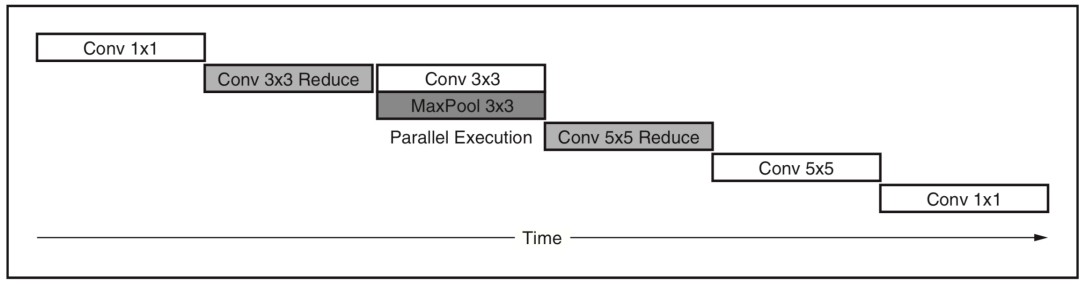


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx





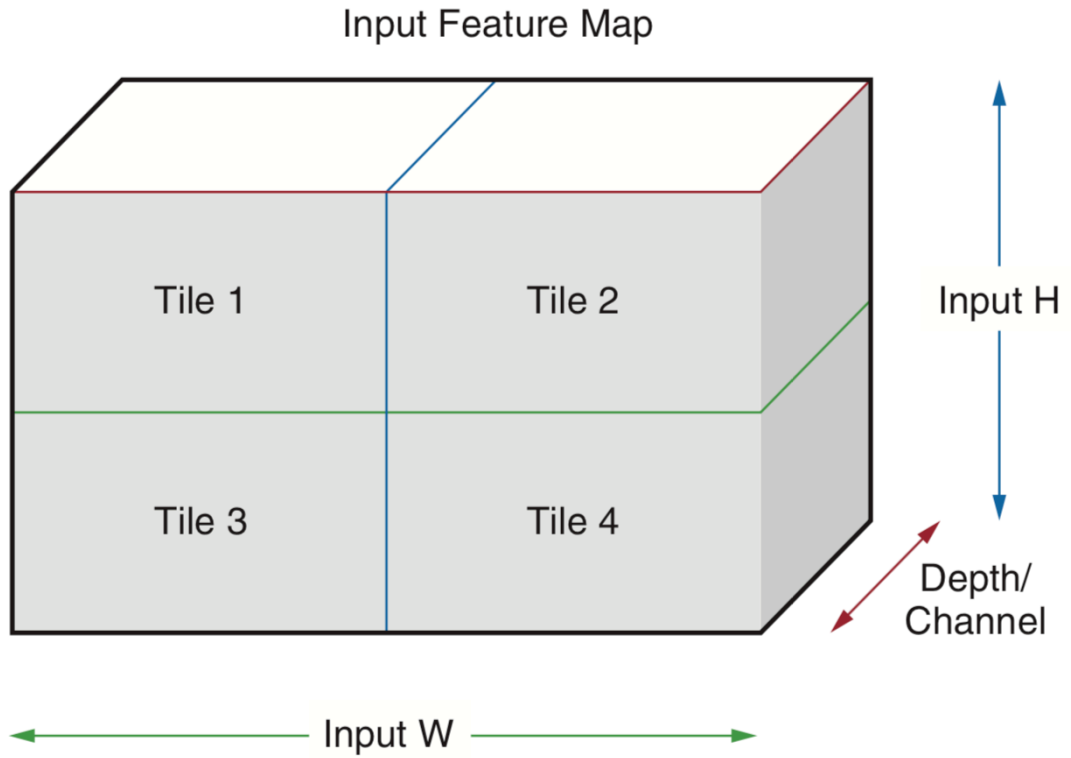


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx

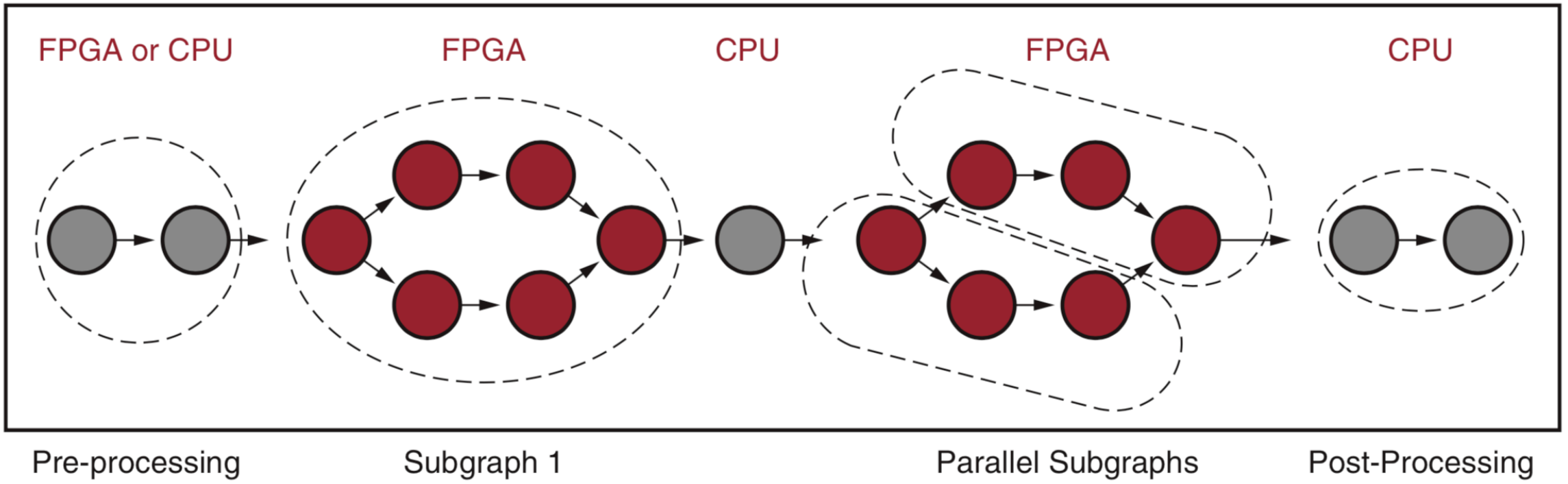


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx

High level software / hardware view for Versal devices

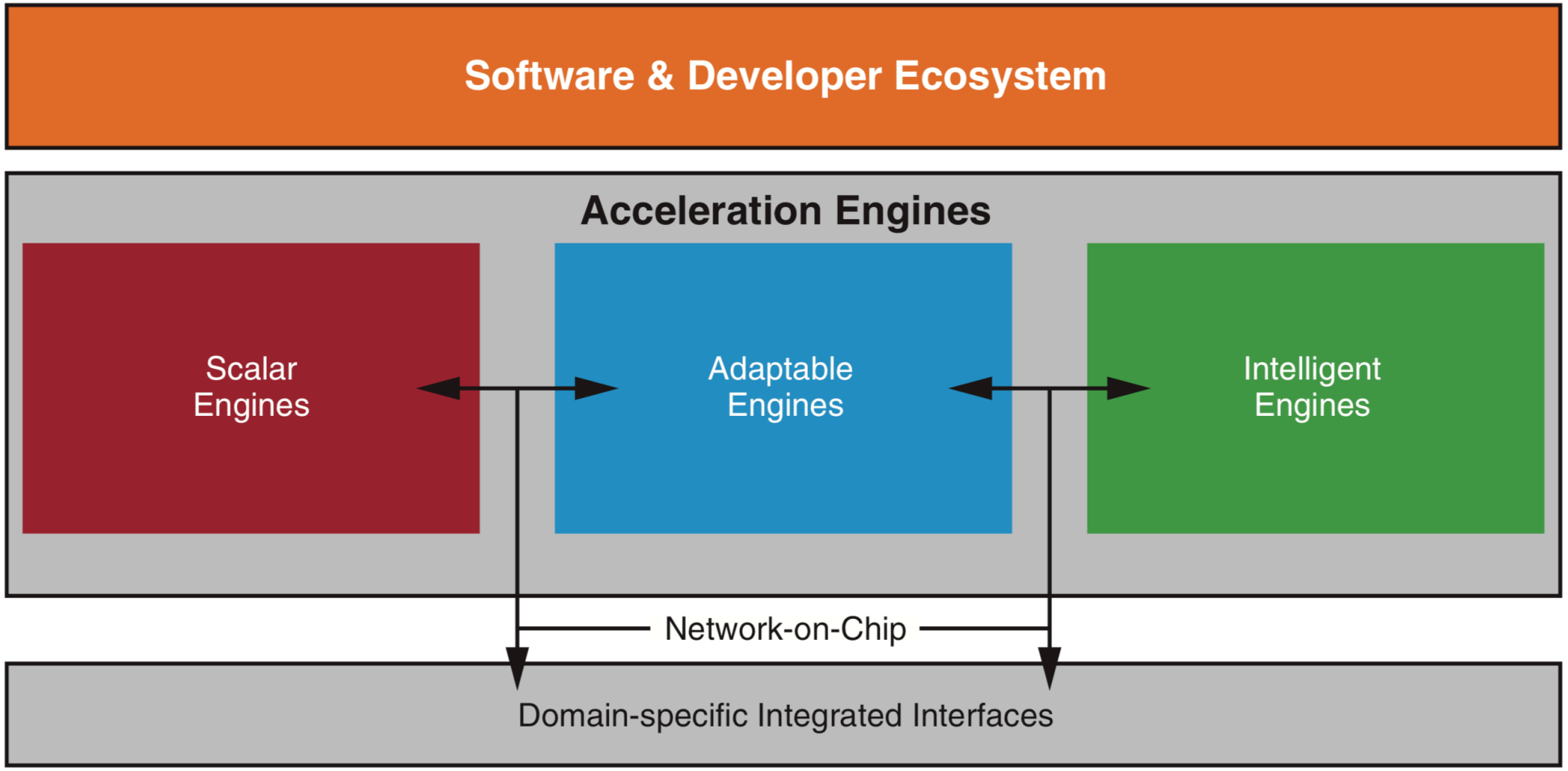


Figure from https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf

Xilinx

High level software / hardware view for Versal devices

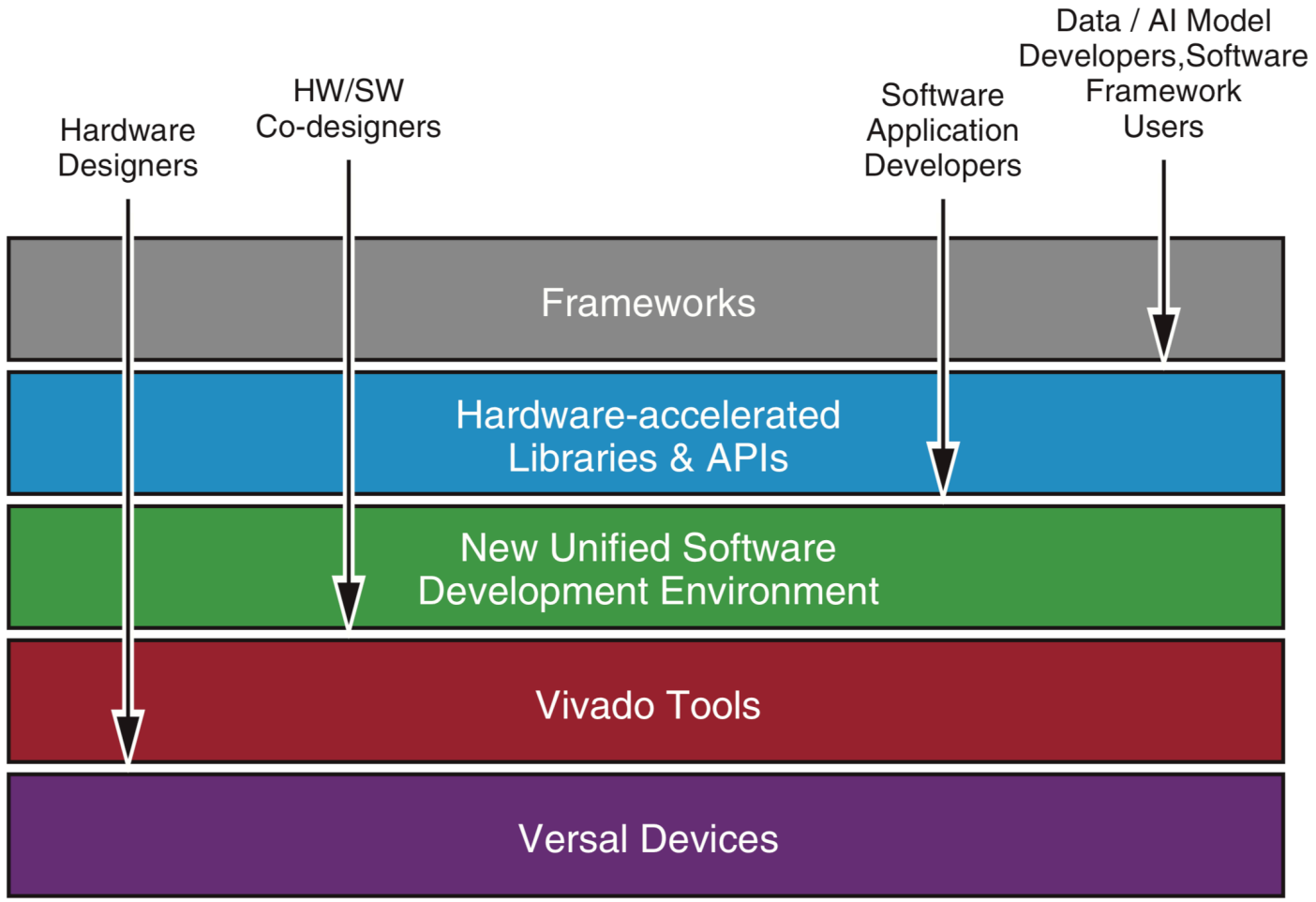


Figure from https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf

Backup – Example Hardware

ARM Reference SoC Diagram

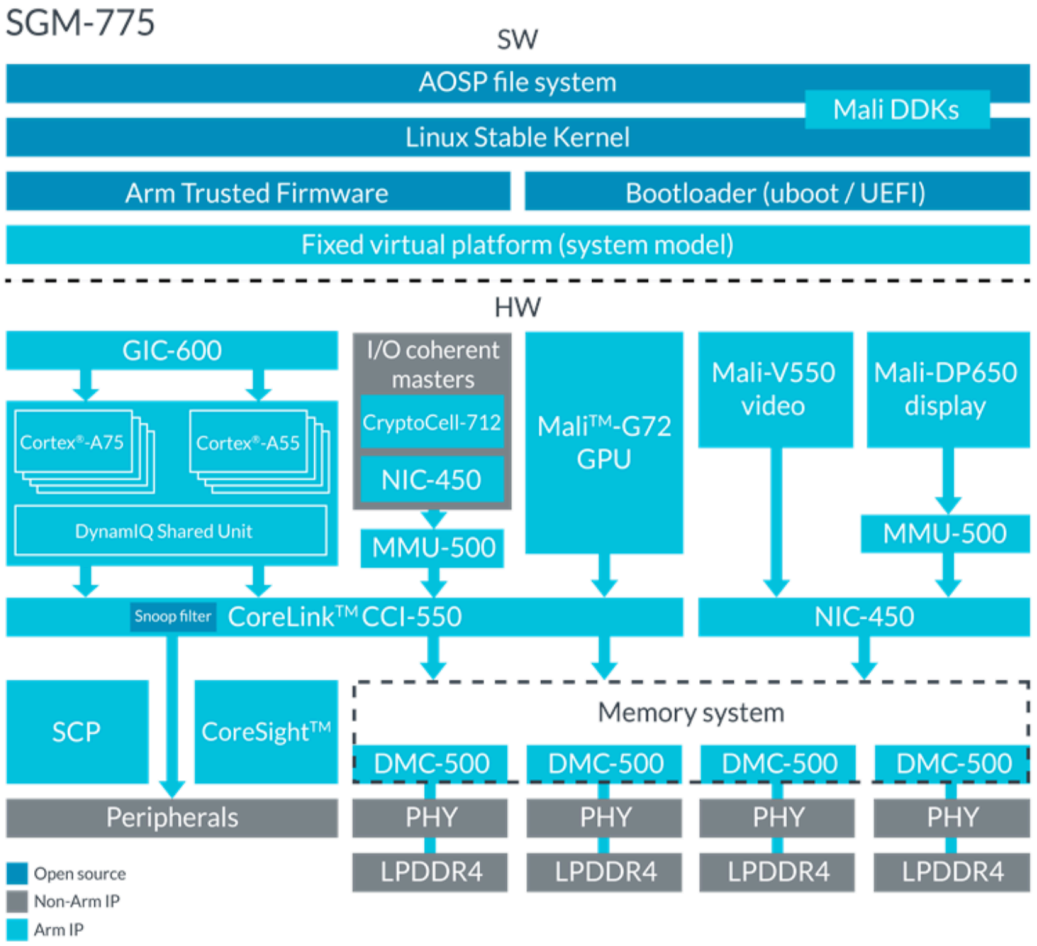


Figure from https://developer.arm.com/products/system-ip

ARM ML Processor

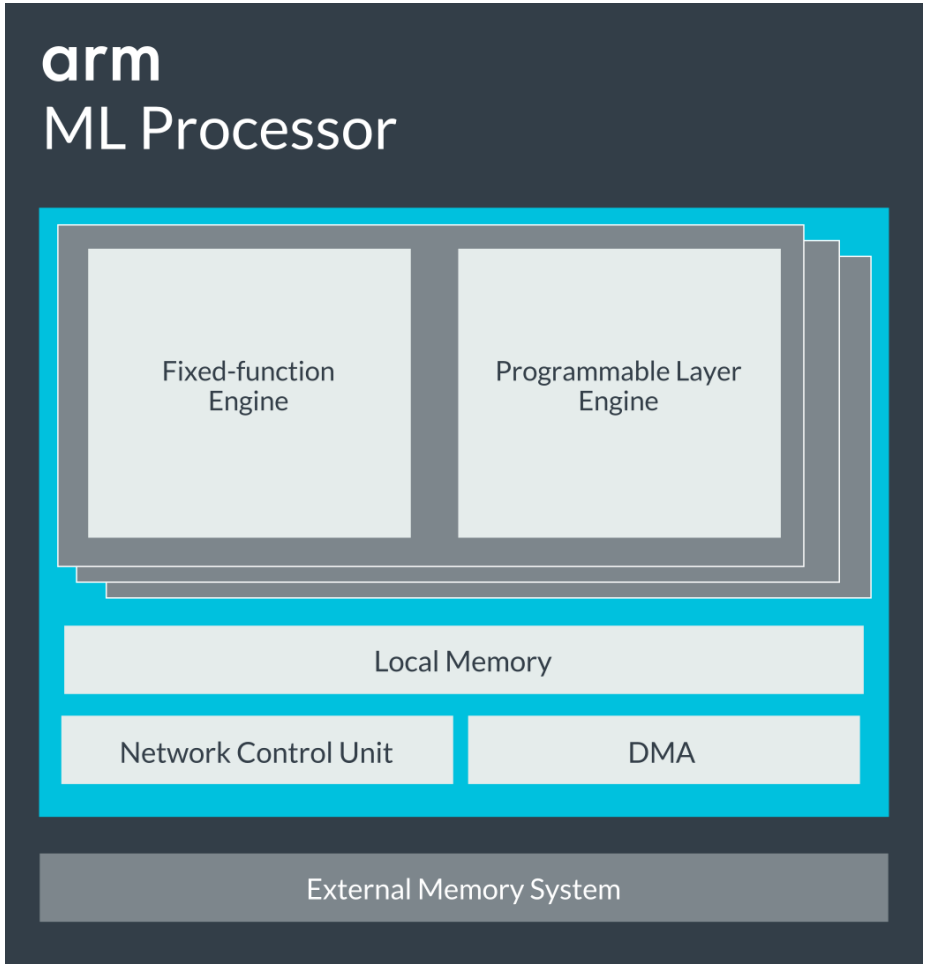
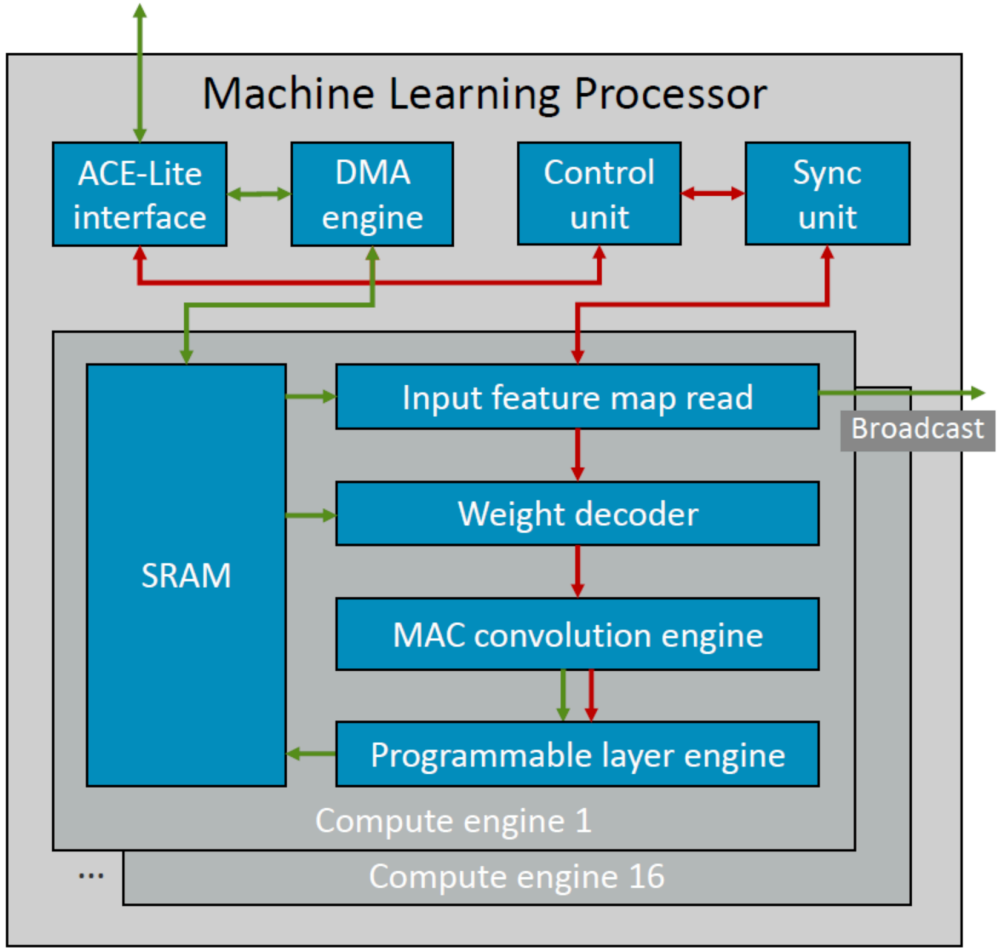


Figure from https://community.arm.com/developer/ip-products/processors/b/ml-ip-blog/posts/arm-ml-processor-exciting-ux-on-edge-devices

ARM ML Processor



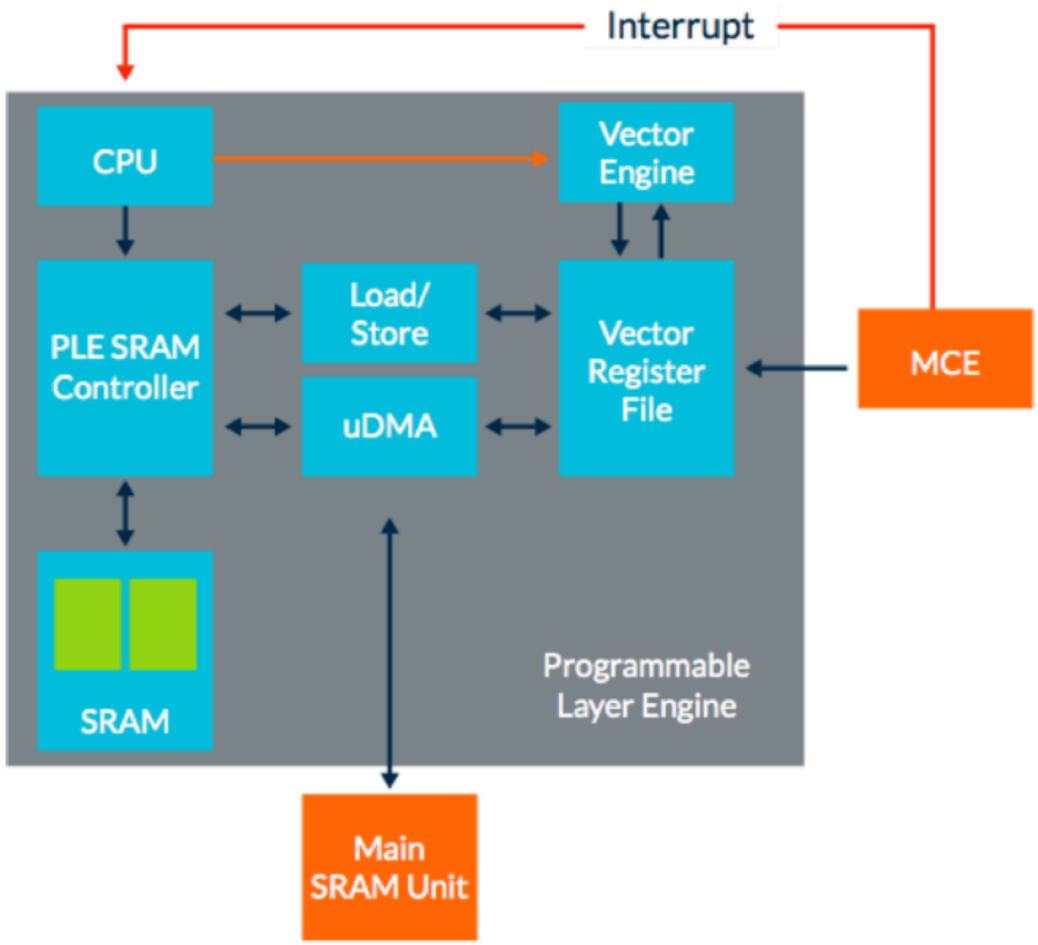


Figure from https://www.anandtech.com/show/12791/arm-details-project-trillium-mlp-architecture

Cadence Tensilica DNA 100

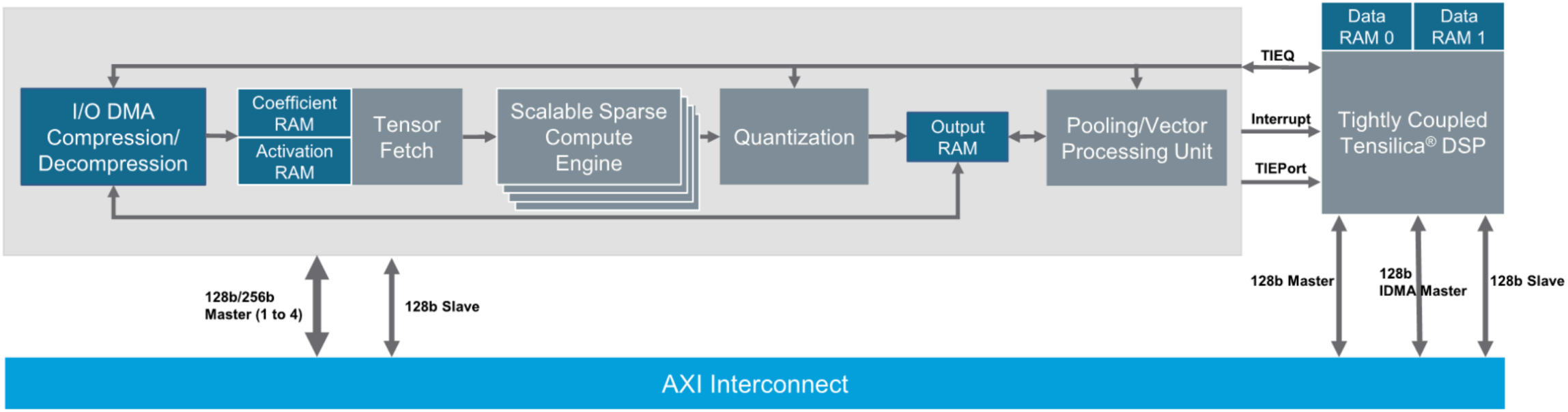


Figure from https://www.anandtech.com/show/13377/cadence-announces-tensilica-dna-100-a-bigger-nn-ip

CEVA CDNN And NeuPro

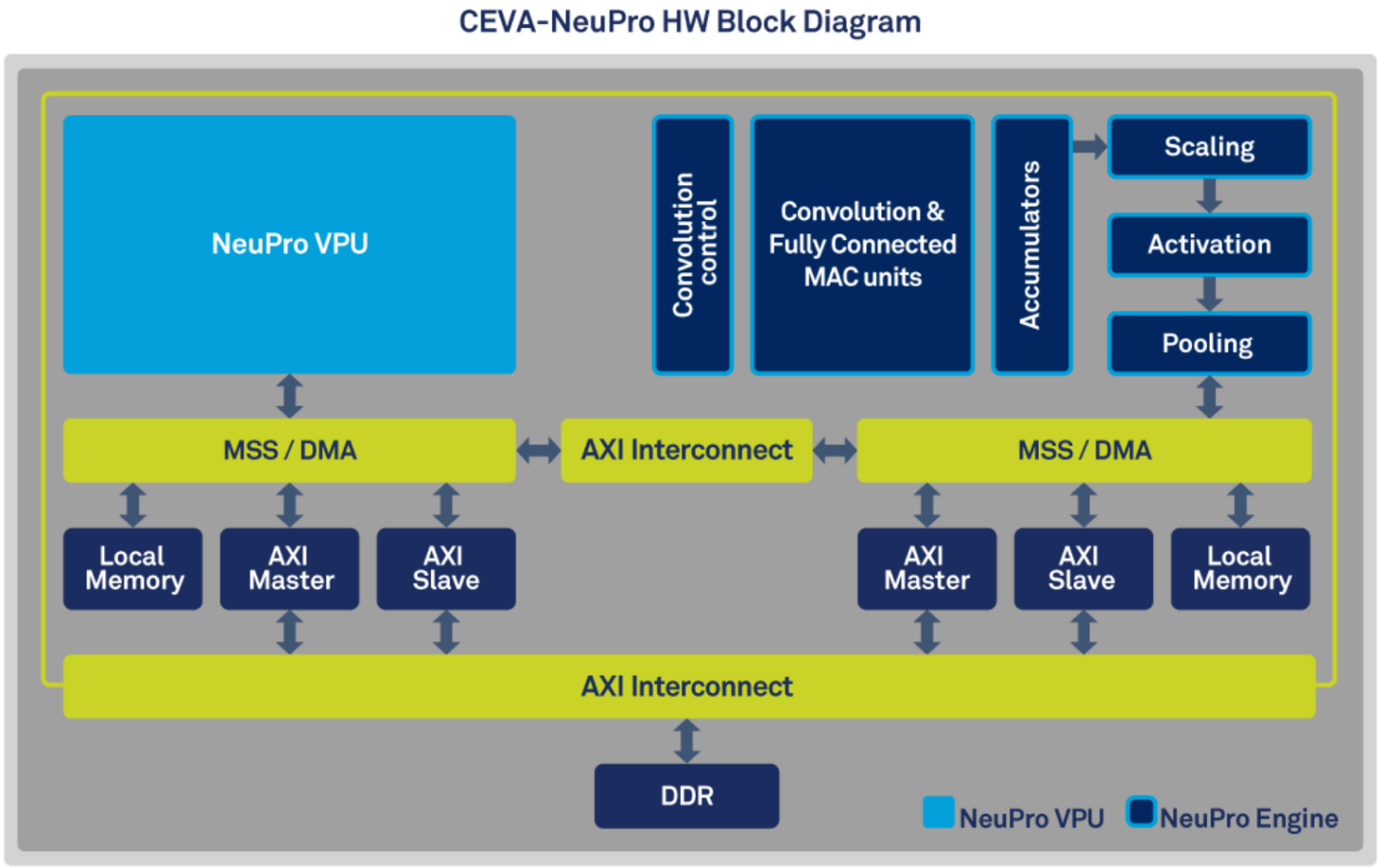


Figure from https://www.ceva-dsp.com/product/ceva-neupro/

Google Coral – Edge TPU

- Edge TPU module
- Compute
  - CPU: 4x ARM A53
  - MCU: 1x ARM M4F
  - GPU: C7000L
  - TPU: Google Edge 4 TOPS (2W for the Edge TPU chip)
- Memory
  - DRAM: 1GB LPDDR4
  - Flash: 8GB eMMC
- I/O
  - WiFi 2x2 MIMO 802.11 b/g/n/ac 2.4/5 GHz
  - Bluetooth 4.1
  - Gigabit Ethernet
  - USB 3.0 type A and C
  - USB 2.0 micro B
  - Audio: 3.5 mm and digital PDM
  - Video: HDMI 2.0a
  - Display: MIPI-DSI 4 lane
  - Camera: MIPI-CSI2 4 lane
  - ...

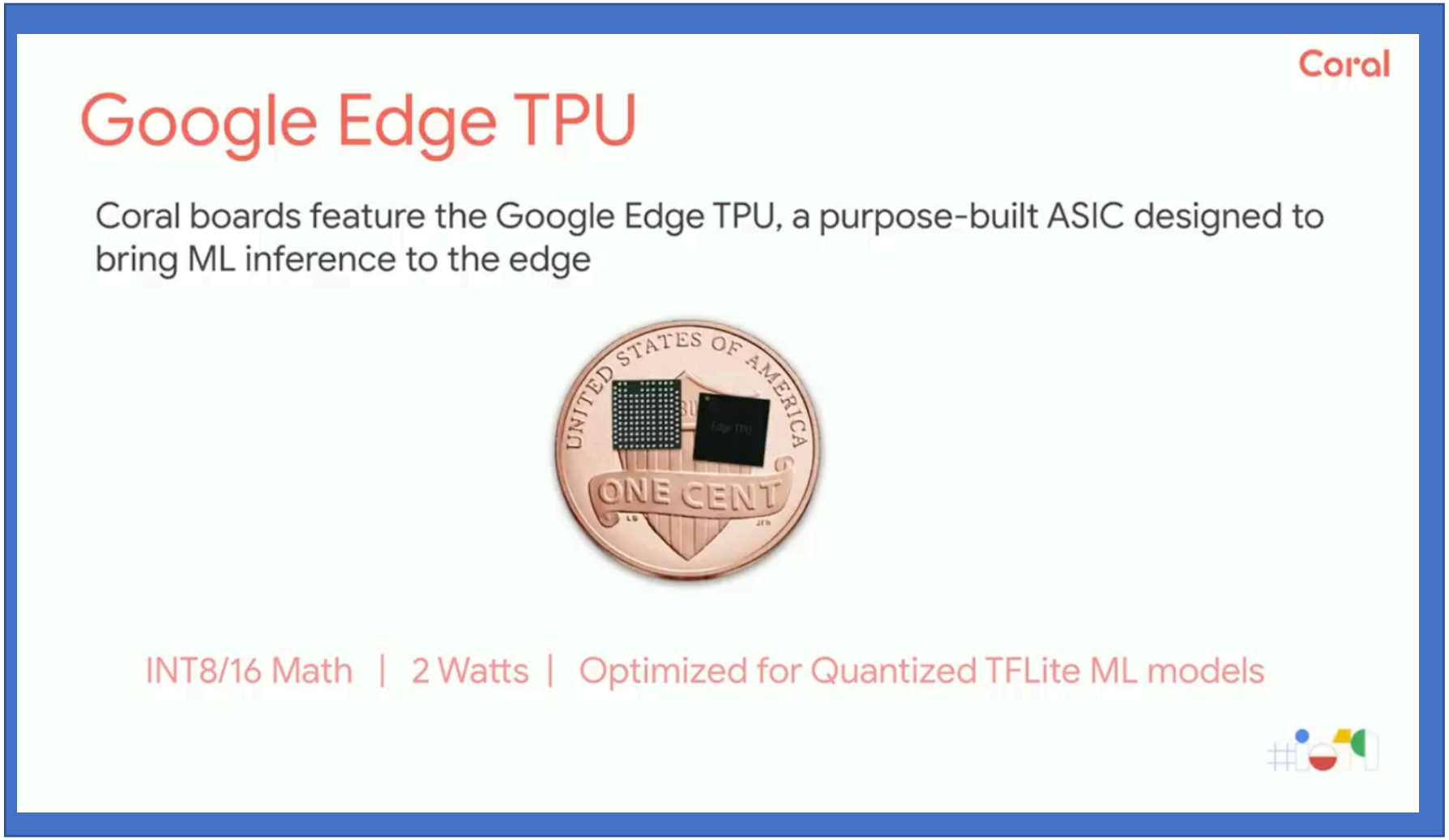


Figure from https://www.youtube.com/watch?v=Jgm25QdF90A

Google Coral – Edge TPU

- Edge TPU module
- Compute
  - CPU: 4x ARM A53
  - MCU: 1x ARM M4F
  - GPU: C7000L
  - TPU: Google Edge 4 TOPS (2W for the Edge TPU chip)
- Memory
  - DRAM: 1GB LPDDR4
  - Flash: 8GB eMMC
- I/O
  - WiFi 2x2 MIMO 802.11 b/g/n/ac 2.4/5 GHz
  - Bluetooth 4.1
  - Gigabit Ethernet
  - USB 3.0 type A and C
  - USB 2.0 micro B
  - Audio: 3.5 mm and digital PDM
  - Video: HDMI 2.0a
  - Display: MIPI-DSI 4 lane
  - Camera: MIPI-CSI2 4 lane
  - …

Coral

Edge TPU performance

Edge TPU coprocessor has the capability of running up to 4 Trillion operations per second (TOPS)

Performance comparison in running various ML models for on-device inferencing

Model architecture	Desktop CPU*	Desktop CPU * + USB Accelerator (USB 3.0) *with Edge TPU*	Embedded CPU **	Dev Board † *with Edge TPU*
MobileNet v1	47 ms	2.2 ms	179 ms	2.2 ms
MobileNet v2	45 ms	2.3 ms	150 ms	2.5 ms
Inception v1	92 ms	3.6 ms	406 ms	3.9 ms
Inception v4	792 ms	100 ms	3,463 ms	100 ms

* *Desktop CPU: 64-bit Intel(R) Xeon(R) E5-1650 v4 @ 3.60GHz* ** *Embedded CPU: Quad-core Cortex-A53 @ 1.5GHz*
† *Dev Board: Quad-core Cortex-A53 @ 1.5GHz + Edge TPU*

Figure from https://www.youtube.com/watch?v=Jgm25QdF90A

Google Coral – Dev Board

- Edge TPU module
- Compute
  - CPU: 4x ARM A53
  - MCU: 1x ARM M4F
  - GPU: C7000L
  - TPU: Google Edge 4 TOPS (2W for the Edge TPU chip)
- Memory
  - DRAM: 1GB LPDDR4
  - Flash: 8GB eMMC
- I/O
  - WiFi 2x2 MIMO 802.11 b/g/n/ac 2.4/5 GHz
  - Bluetooth 4.1
  - Gigabit Ethernet
  - USB 3.0 type A and C
  - USB 2.0 micro B
  - Audio: 3.5 mm and digital PDM
  - Video: HDMI 2.0a
  - Display: MIPI-DSI 4 lane
  - Camera: MIPI-CSI2 4 lane
  - …

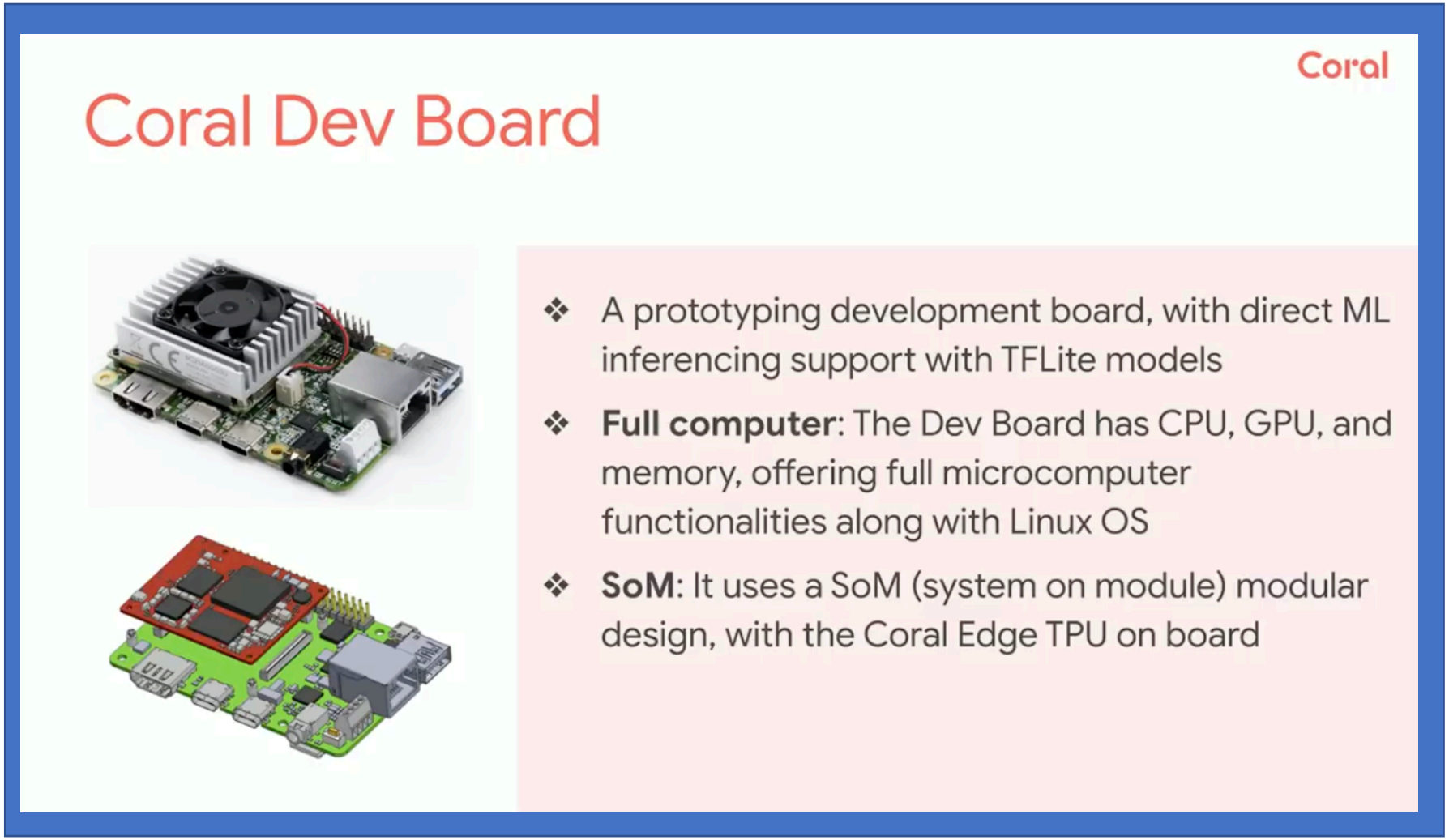


Figure from https://www.youtube.com/watch?v=Jgm25QdF90A

Google Coral – Dev Board

- Edge TPU module
- Compute
  - CPU: 4x ARM A53
  - MCU: 1x ARM M4F
  - GPU: C7000L
  - TPU: Google Edge 4 TOPS (2W for the Edge TPU chip)
- Memory
  - DRAM: 1GB LPDDR4
  - Flash: 8GB eMMC
- I/O
  - WiFi 2x2 MIMO 802.11 b/g/n/ac 2.4/5 GHz
  - Bluetooth 4.1
  - Gigabit Ethernet
  - USB 3.0 type A and C
  - USB 2.0 micro B
  - Audio: 3.5 mm and digital PDM
  - Video: HDMI 2.0a
  - Display: MIPI-DSI 4 lane
  - Camera: MIPI-CSI2 4 lane
  - ...

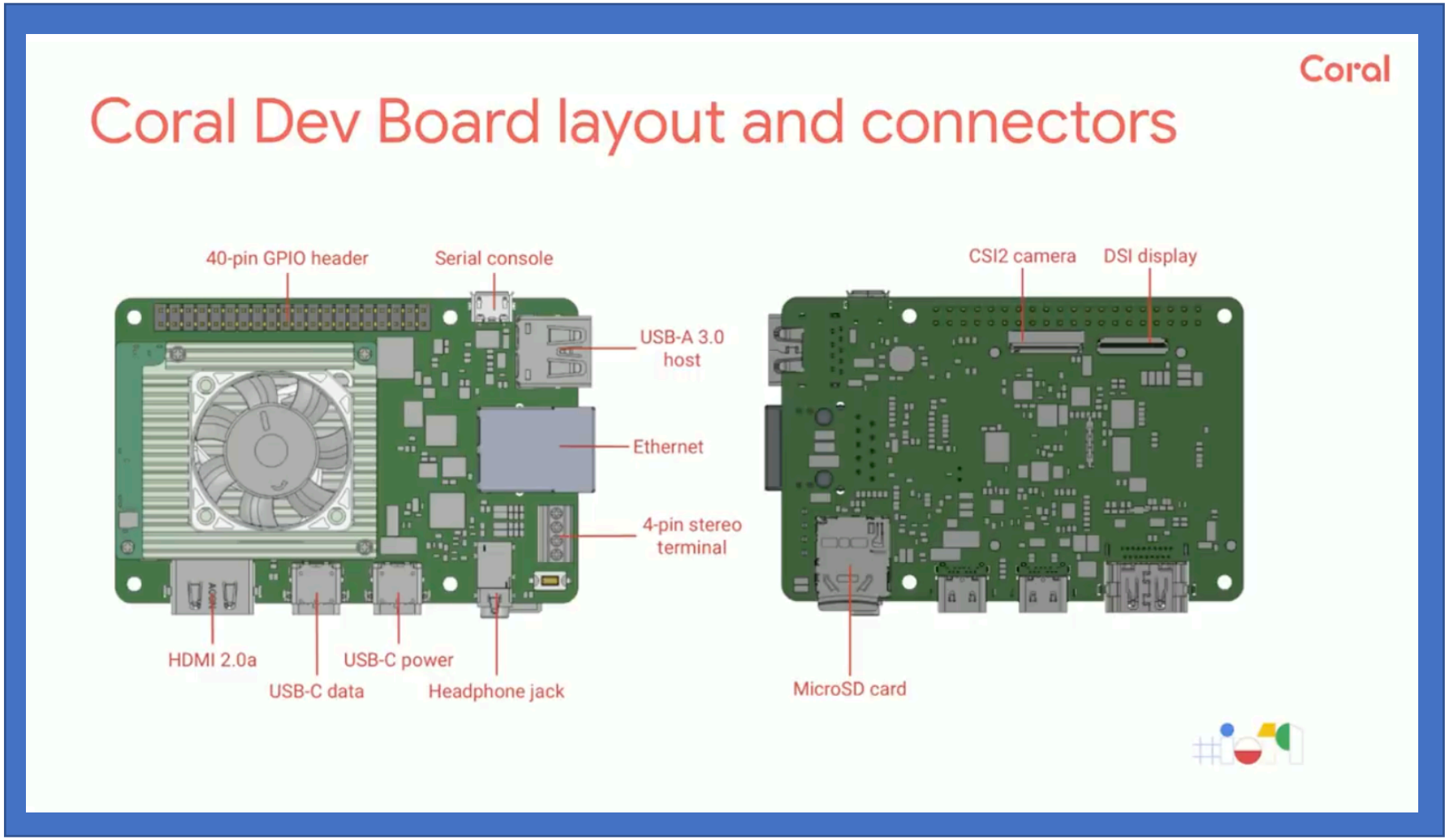


Figure from https://www.youtube.com/watch?v=Jgm25QdF90A

Google Coral – Dev Board

- Edge TPU module
- Compute
  - CPU: 4x ARM A53
  - MCU: 1x ARM M4F
  - GPU: C7000L
  - TPU: Google Edge 4 TOPS (2W for the Edge TPU chip)
- Memory
  - DRAM: 1GB LPDDR4
  - Flash: 8GB eMMC
- I/O
  - WiFi 2x2 MIMO 802.11 b/g/n/ac 2.4/5 GHz
  - Bluetooth 4.1
  - Gigabit Ethernet
  - USB 3.0 type A and C
  - USB 2.0 micro B
  - Audio: 3.5 mm and digital PDM
  - Video: HDMI 2.0a
  - Display: MIPI-DSI 4 lane
  - Camera: MIPI-CSI2 4 lane
  - …

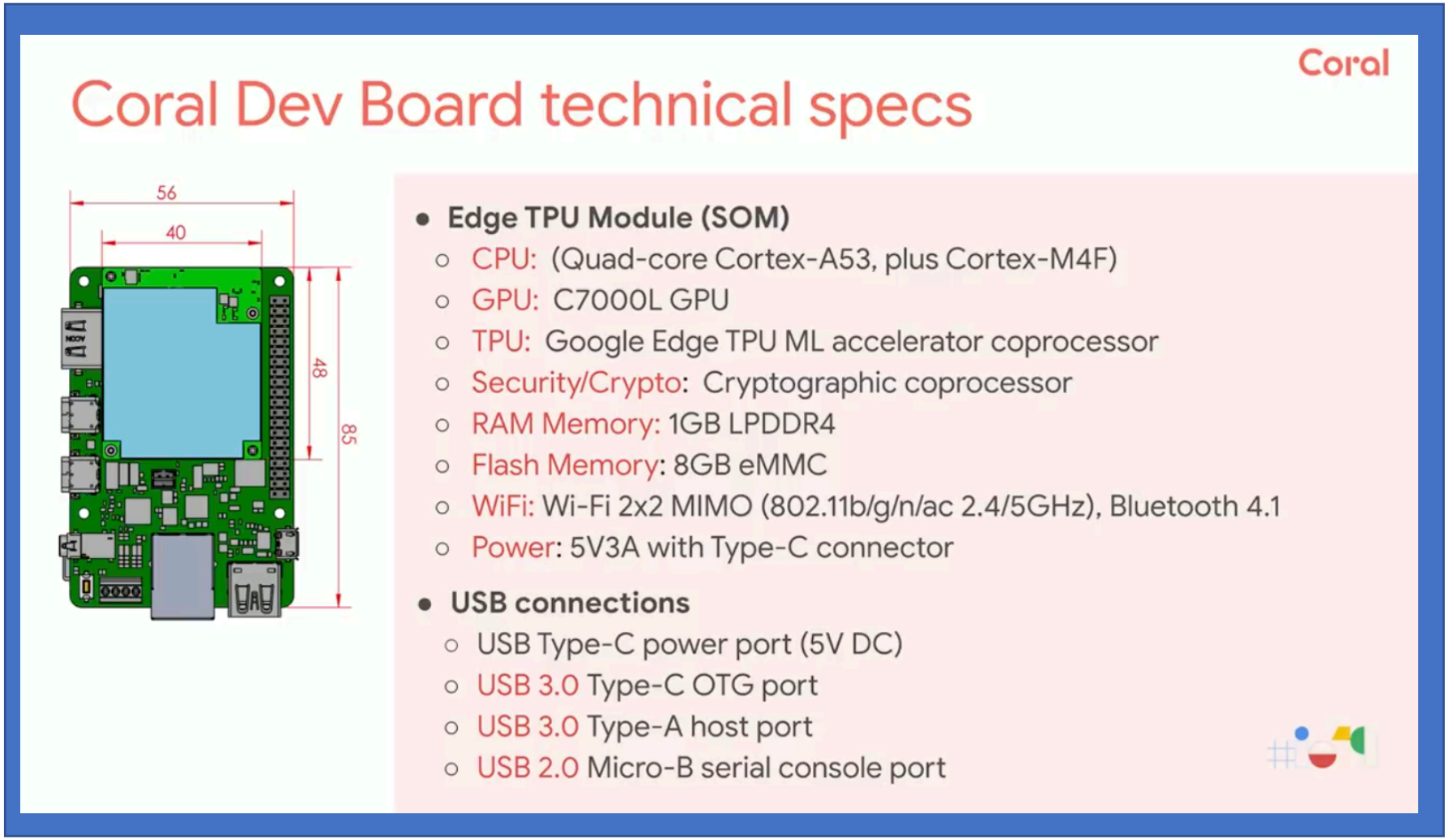


Figure from https://www.youtube.com/watch?v=Jgm25QdF90A

Google Coral – Dev Board

- Edge TPU module
- Compute
  - CPU: 4x ARM A53
  - MCU: 1x ARM M4F
  - GPU: C7000L
  - TPU: Google Edge 4 TOPS (2W for the Edge TPU chip)
- Memory
  - DRAM: 1GB LPDDR4
  - Flash: 8GB eMMC
- I/O
  - WiFi 2x2 MIMO 802.11 b/g/n/ac 2.4/5 GHz
  - Bluetooth 4.1
  - Gigabit Ethernet
  - USB 3.0 type A and C
  - USB 2.0 micro B
  - Audio: 3.5 mm and digital PDM
  - Video: HDMI 2.0a
  - Display: MIPI-DSI 4 lane
  - Camera: MIPI-CSI2 4 lane
  - …

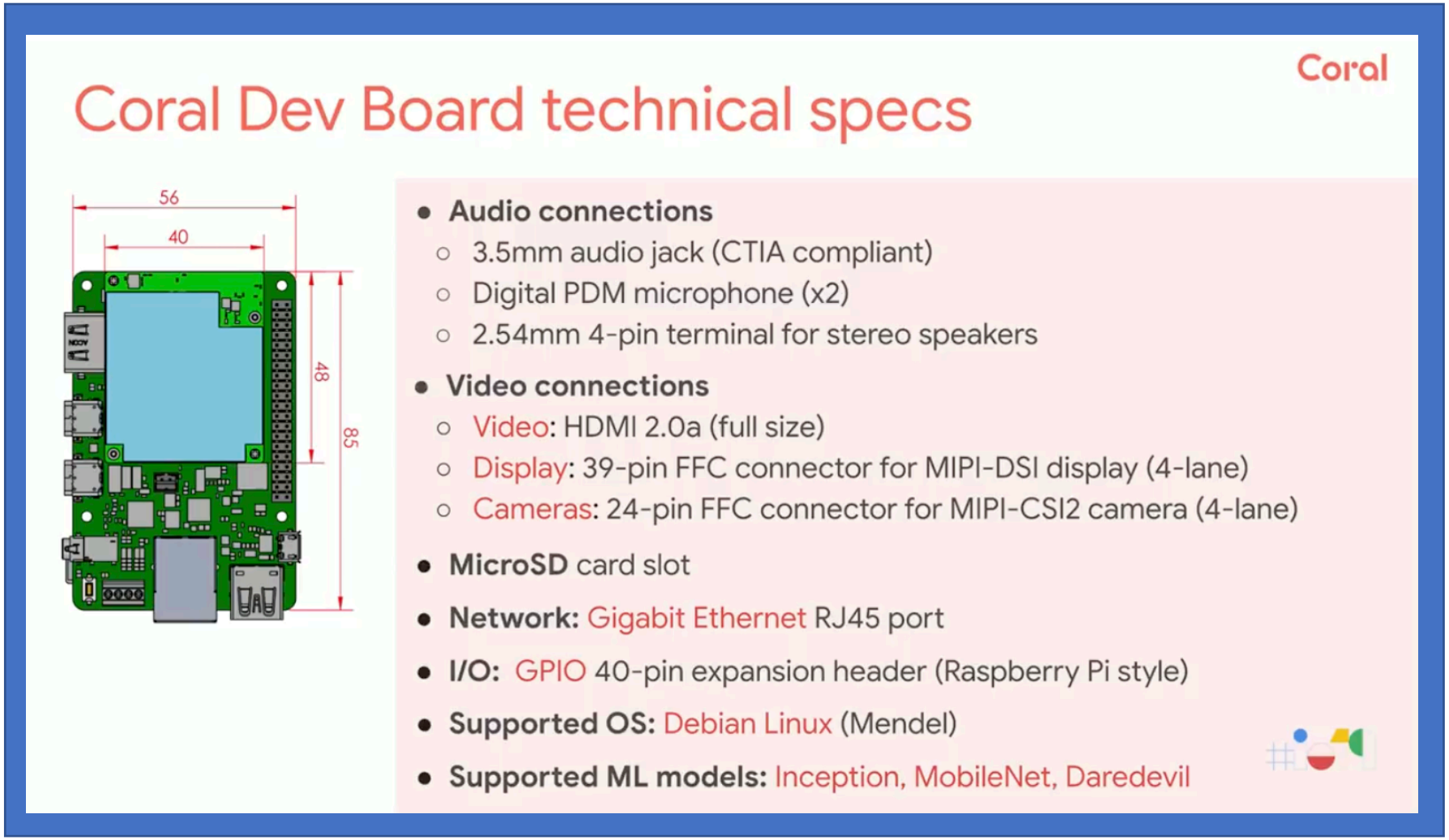


Figure from https://www.youtube.com/watch?v=Jgm25QdF90A

Google Coral – Dev Board

- Edge TPU module
- Compute
  - CPU: 4x ARM A53
  - MCU: 1x ARM M4F
  - GPU: C7000L
  - TPU: Google Edge 4 TOPS (2W for the Edge TPU chip)
- Memory
  - DRAM: 1GB LPDDR4
  - Flash: 8GB eMMC
- I/O
  - WiFi 2x2 MIMO 802.11 b/g/n/ac 2.4/5 GHz
  - Bluetooth 4.1
  - Gigabit Ethernet
  - USB 3.0 type A and C
  - USB 2.0 micro B
  - Audio: 3.5 mm and digital PDM
  - Video: HDMI 2.0a
  - Display: MIPI-DSI 4 lane
  - Camera: MIPI-CSI2 4 lane
  - …

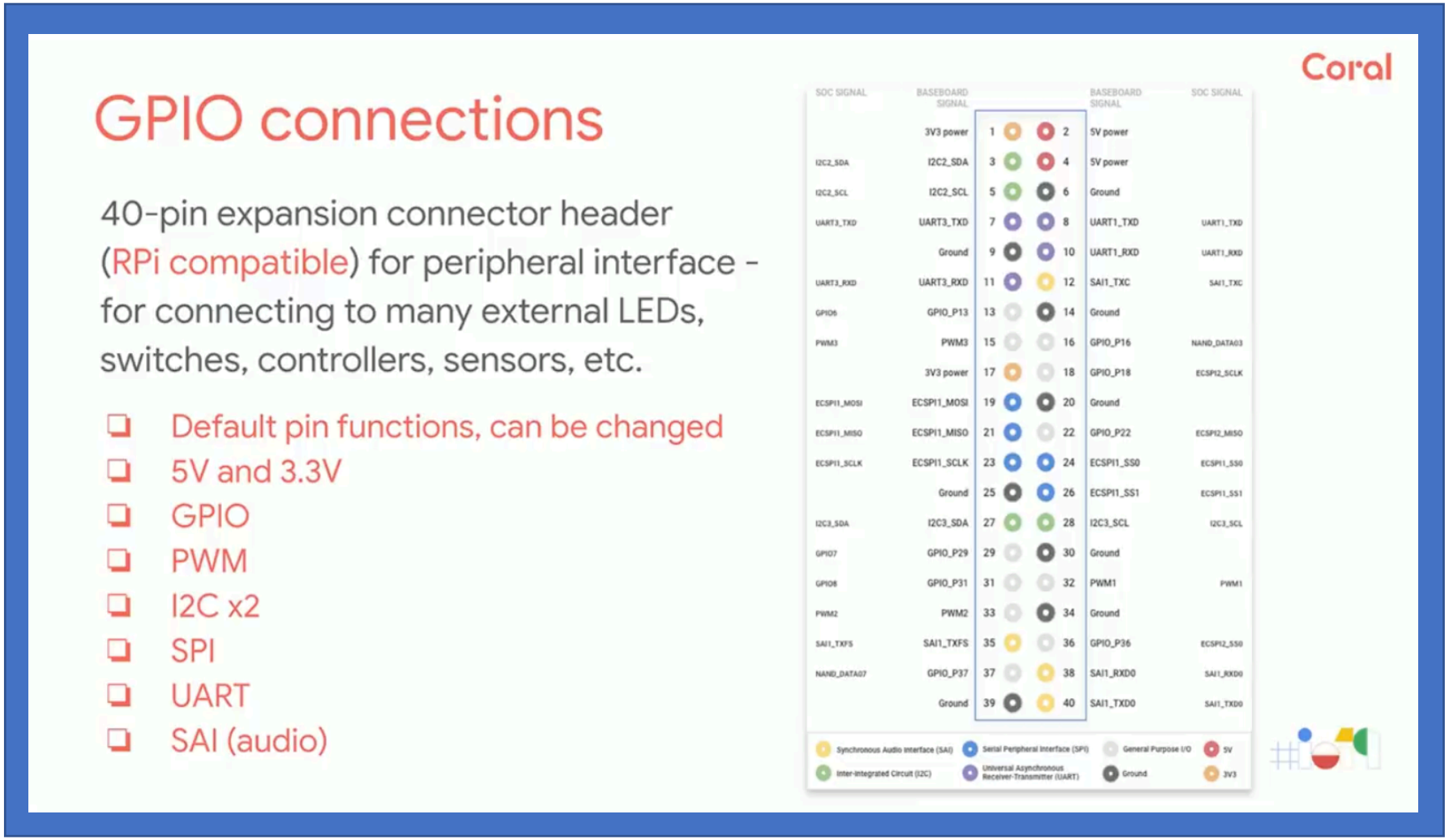


Figure from https://www.youtube.com/watch?v=Jgm25QdF90A

Google Coral – USB Stick

- Edge TPU module
- Compute
  - CPU: 4x ARM A53
  - MCU: 1x ARM M4F
  - GPU: C7000L
  - TPU: Google Edge 4 TOPS (2W for the Edge TPU chip)
- Memory
  - DRAM: 1GB LPDDR4
  - Flash: 8GB eMMC
- I/O
  - WiFi 2x2 MIMO 802.11 b/g/n/ac 2.4/5 GHz
  - Bluetooth 4.1
  - Gigabit Ethernet
  - USB 3.0 type A and C
  - USB 2.0 micro B
  - Audio: 3.5 mm and digital PDM
  - Video: HDMI 2.0a
  - Display: MIPI-DSI 4 lane
  - Camera: MIPI-CSI2 4 lane
  - ...

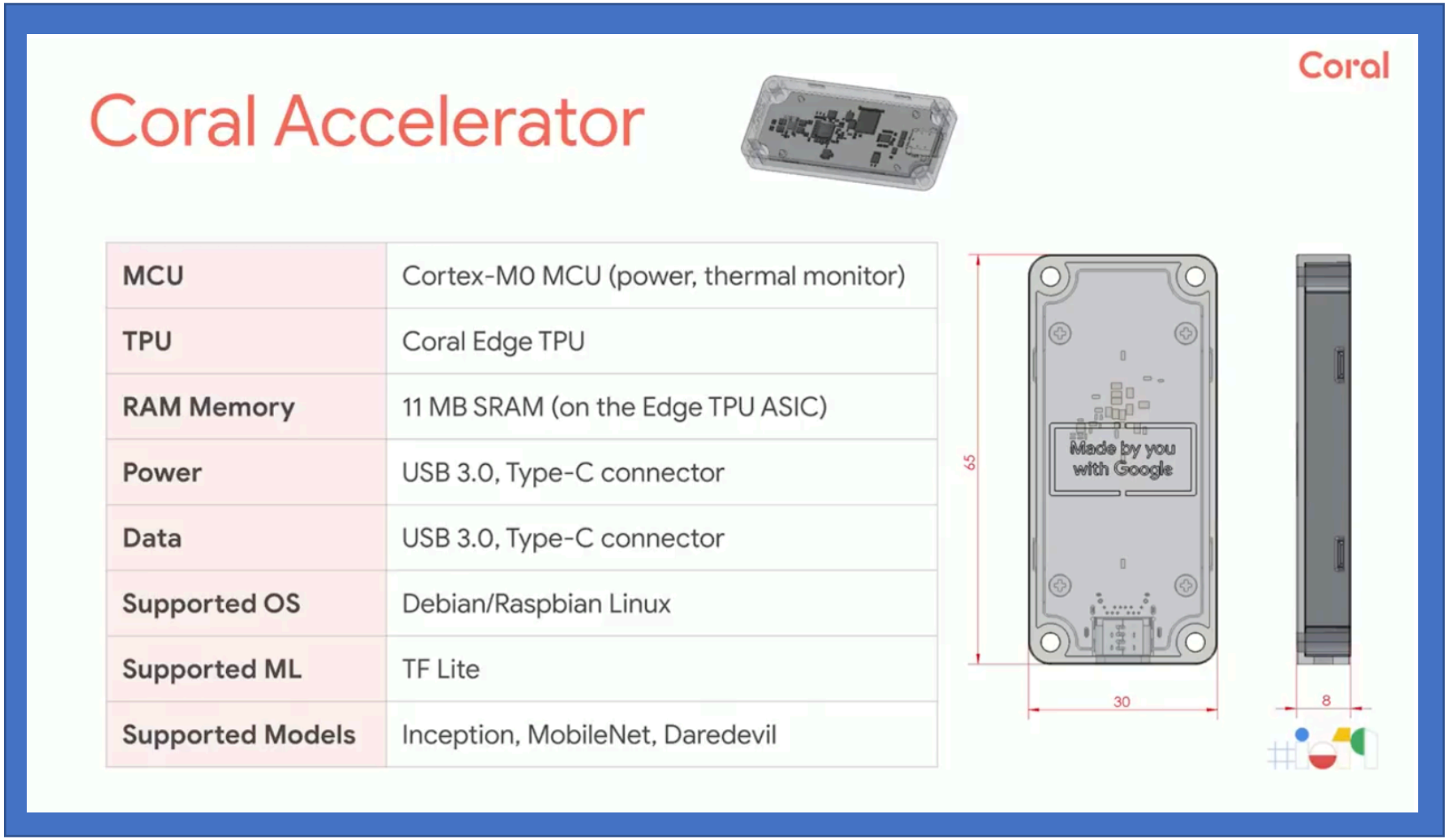


Figure from https://www.youtube.com/watch?v=Jgm25QdF90A

Google Coral – USB Stick

- Edge TPU module
- Compute
  - CPU: 4x ARM A53
  - MCU: 1x ARM M4F
  - GPU: C7000L
  - TPU: Google Edge 4 TOPS (2W for the Edge TPU chip)
- Memory
  - DRAM: 1GB LPDDR4
  - Flash: 8GB eMMC
- I/O
  - WiFi 2x2 MIMO 802.11 b/g/n/ac 2.4/5 GHz
  - Bluetooth 4.1
  - Gigabit Ethernet
  - USB 3.0 type A and C
  - USB 2.0 micro B
  - Audio: 3.5 mm and digital PDM
  - Video: HDMI 2.0a
  - Display: MIPI-DSI 4 lane
  - Camera: MIPI-CSI2 4 lane
  - …

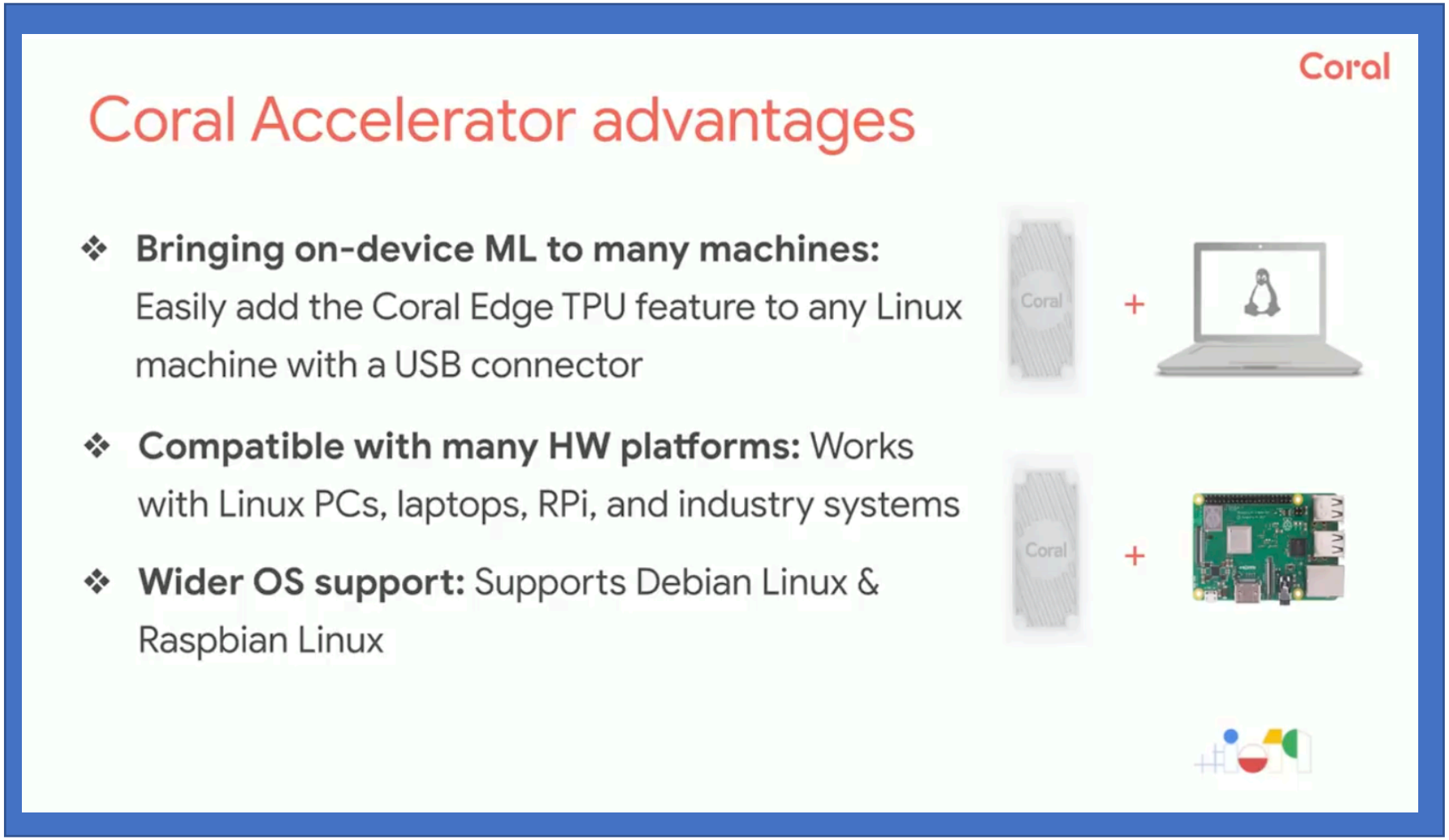


Figure from https://www.youtube.com/watch?v=Jgm25QdF90A

Google TPU V1

- Notes
  - Focused on inference
  - 256 x 256 matrix multiplier with 8b int operands running at 700 MHz for ~ 91.75 TOPS performance
  - 30 GB/s of memory bandwidth
  - 36? MB on device memory

- Links
  - https://arxiv.org/abs/1704.04760

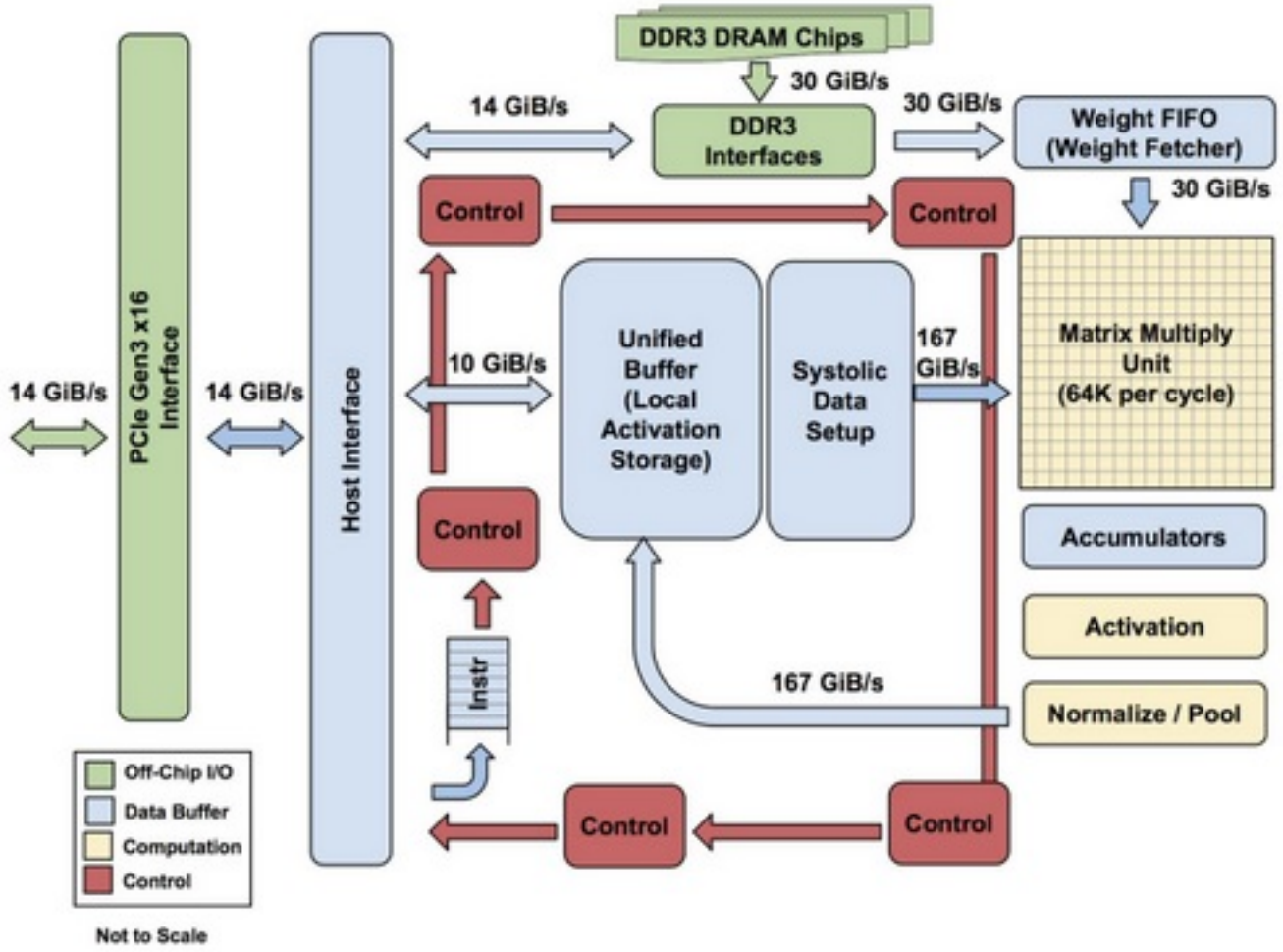


Figure from https://www.nextplatform.com/2017/04/05/first-depth-look-googles-tpu-architecture/

Google TPU V2

- Notes
  - Focused on training and inference
  - Each chip contains 2 cores
  - Each core contains a 128 x 128 matrix multiplier with 16 bfloat operands and 32b float accumulation running at ~ 1.375 GHz for ~ 22.5 TFLOPS performance along with scalar and vector compute
  - Each core has access to 8 GB of HBM with 300 GB/s of memory bandwidth
  - Total per chip of 45 TFLOPS and 16 GB HBM with 600 GB/s of memory bandwidth
  - 4 chips per board
  - 64 boards per TPU pod
  - Likely connected in a 2D torus

- Links
  - http://learningsys.org/nips17/assets/slides/dean-nips17.pdf
  - https://www.theregister.co.uk/2017/12/14/google_tpu2_specs_ish/
  - https://www.nextplatform.com/2017/05/22/hood-googles-tpu2-machine-learning-clusters/

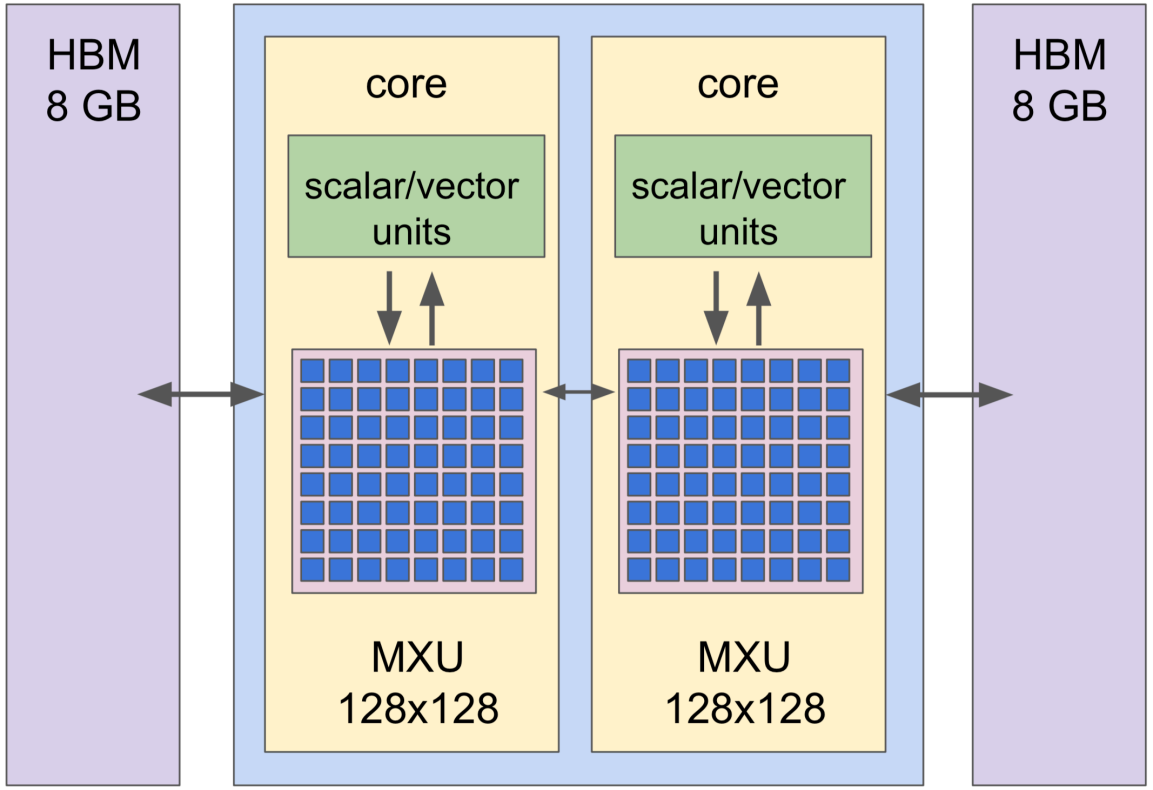
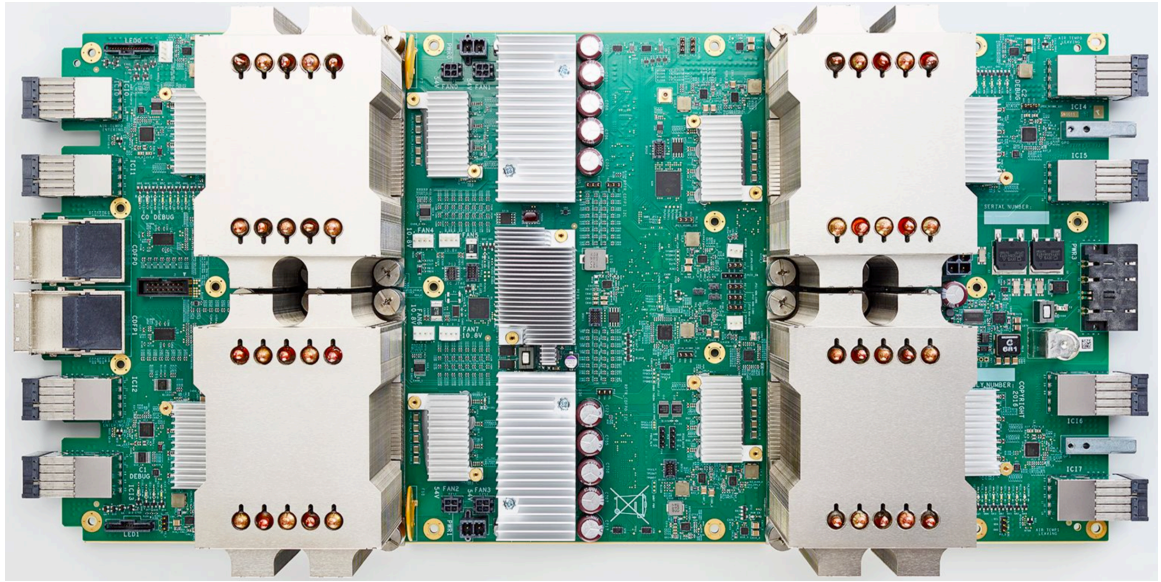


Figure from http://learningsys.org/nips17/assets/slides/dean-nips17.pdf

Google TPU V2

- Notes
  - Focused on training and inference
  - Each chip contains 2 cores
  - Each core contains a 128 x 128 matrix multiplier with 16 bfloat operands and 32b float accumulation running at ~ 1.375 GHz for ~ 22.5 TFLOPS performance along with scalar and vector compute
  - Each core has access to 8 GB of HBM with 300 GB/s of memory bandwidth
  - Total per chip of 45 TFLOPS and 16 GB HBM with 600 GB/s of memory bandwidth
  - 4 chips per board
  - 64 boards per TPU pod
  - Likely connected in a 2D torus



- Links
  - http://learningsys.org/nips17/assets/slides/dean-nips17.pdf
  - https://www.theregister.co.uk/2017/12/14/google_tpu2_specs_ish/
  - https://www.nextplatform.com/2017/05/22/hood-googles-tpu2-machine-learning-clusters/

Figure from http://learningsys.org/nips17/assets/slides/dean-nips17.pdf

Google TPU V2

- Notes
  - Focused on training and inference
  - Each chip contains 2 cores
  - Each core contains a 128 x 128 matrix multiplier with 16 bfloat operands and 32b float accumulation running at ~ 1.375 GHz for ~ 22.5 TFLOPS performance along with scalar and vector compute
  - Each core has access to 8 GB of HBM with 300 GB/s of memory bandwidth
  - Total per chip of 45 TFLOPS and 16 GB HBM with 600 GB/s of memory bandwidth
  - 4 chips per board
  - 64 boards per TPU pod
  - Likely connected in a 2D torus

- Links
  - http://learningsys.org/nips17/assets/slides/dean-nips17.pdf
  - https://www.theregister.co.uk/2017/12/14/google_tpu2_specs_ish/
  - https://www.nextplatform.com/2017/05/22/hood-googles-tpu2-machine-learning-clusters/

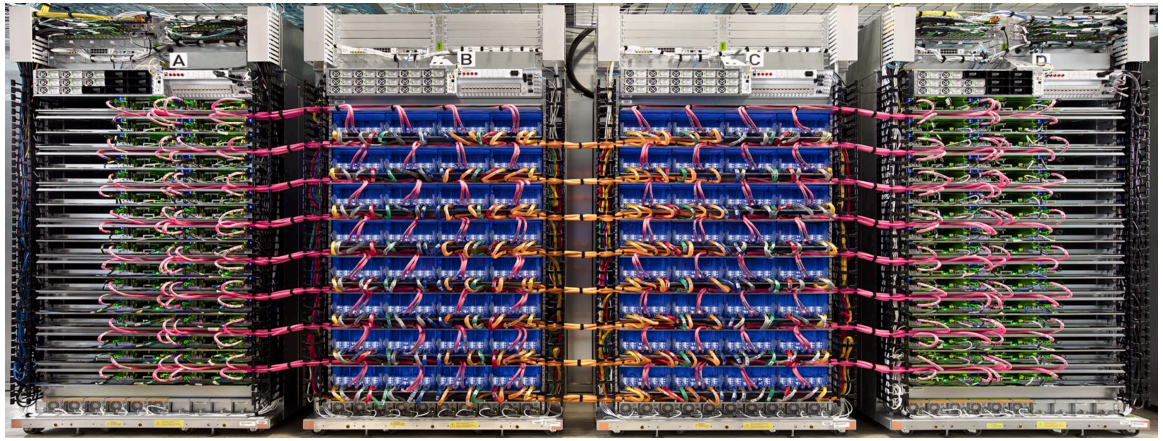
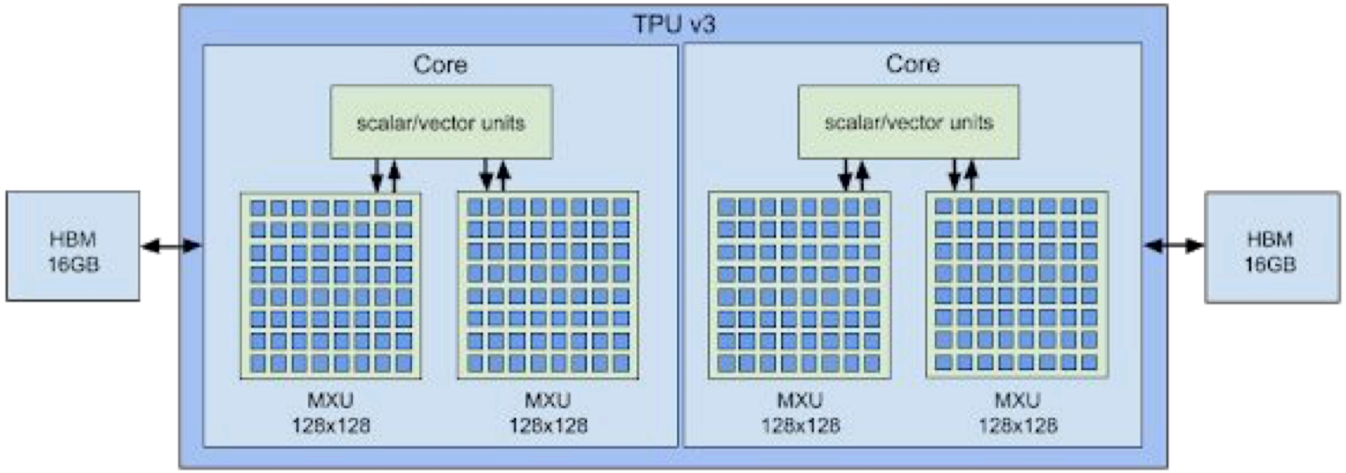


Figure from http://learningsys.org/nips17/assets/slides/dean-nips17.pdf

Google TPU V3

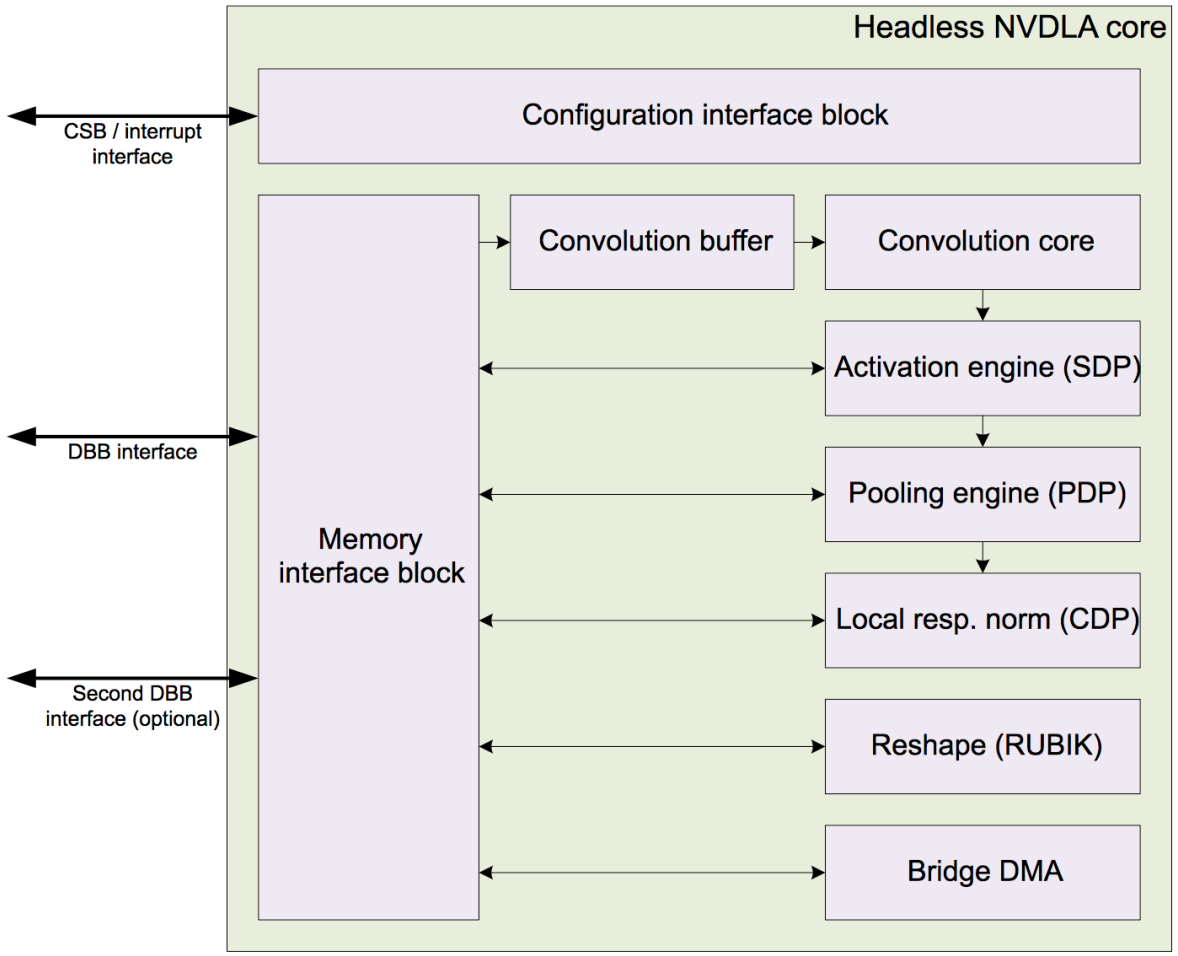
- Notes
  - Each chip contains 2 cores, each core contains 2x 128x128 matrix multipliers with a set of shared scalar and vector units
  - Each core has access to 16 GB of HBM
  - So effectively 2x the compute and 2x the memory vs V2
  - Additional modifications made to the pod configuration and connectivity are also likely



- Links
  - https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor/
  - https://pliss2019.github.io/albert_cohen_slides.pdf

Figure from https://pliss2019.github.io/albert_cohen_slides.pdf

Nvidia NVDLA



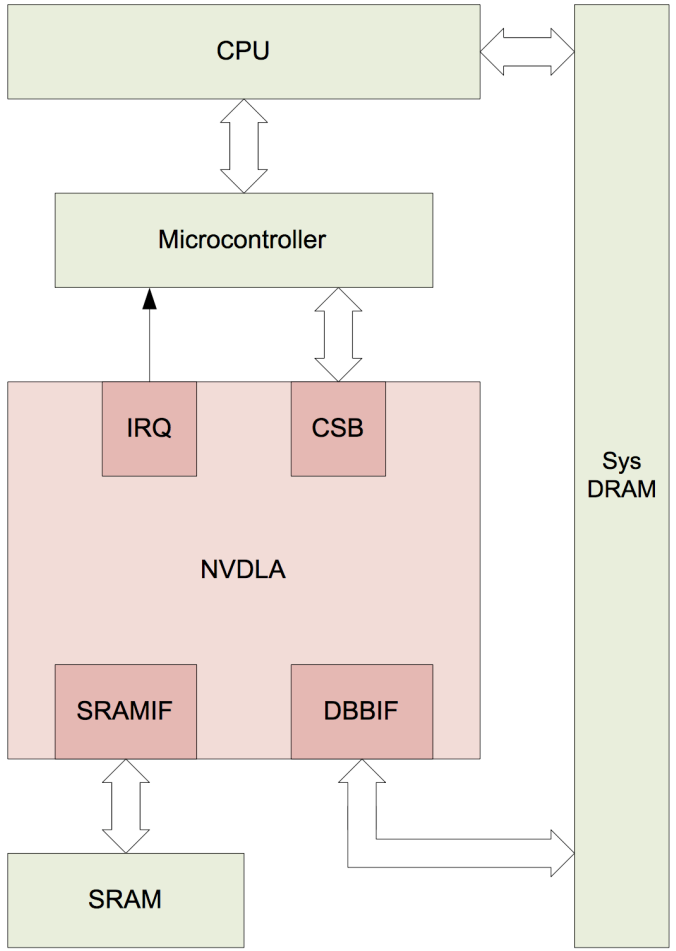


Figure from http://nvdla.org/primer.html

Nvidia Titan RTX

- Partial specs
  - 24 GB VRAM connected via GDDR6 at 672 GB/s
  - Device clock 1.350 / 1.770 GHz
  - 6 MB L2 cache
  - 576 tensor cores with 125 TFLOPS FP16 (FP32 accumulation), 250 TOPS INT8, 500 TOPS INT4
  - 16.3 TFLOPS FP32 and 0.51 FP64
  - 18.6 B transistors and 280 W power in 12 nm FFN

- Links
  - https://www.anandtech.com/show/13668/nvidia-unveils-rtx-titan-2500-top-turing

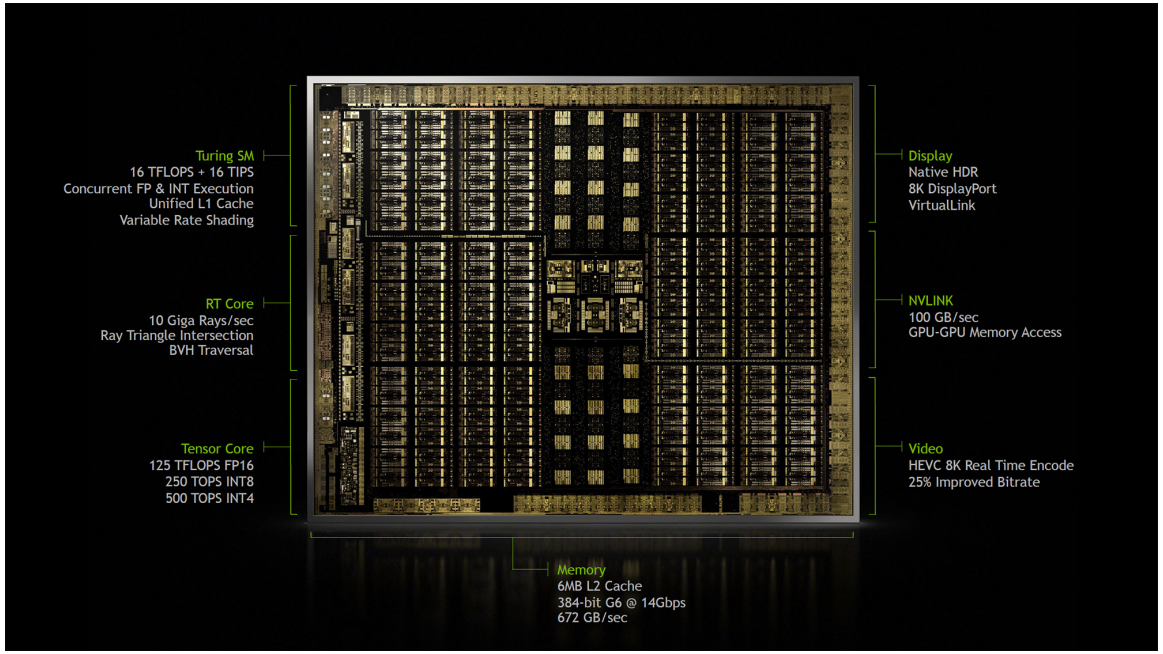


Figure from https://www.anandtech.com/show/13668/nvidia-unveils-rtx-titan-2500-top-turing

Nvidia Turing

Figure from https://www.anandtech.com/print/13282/nvidia-turing-architecture-deep-dive

Nvidia Turing

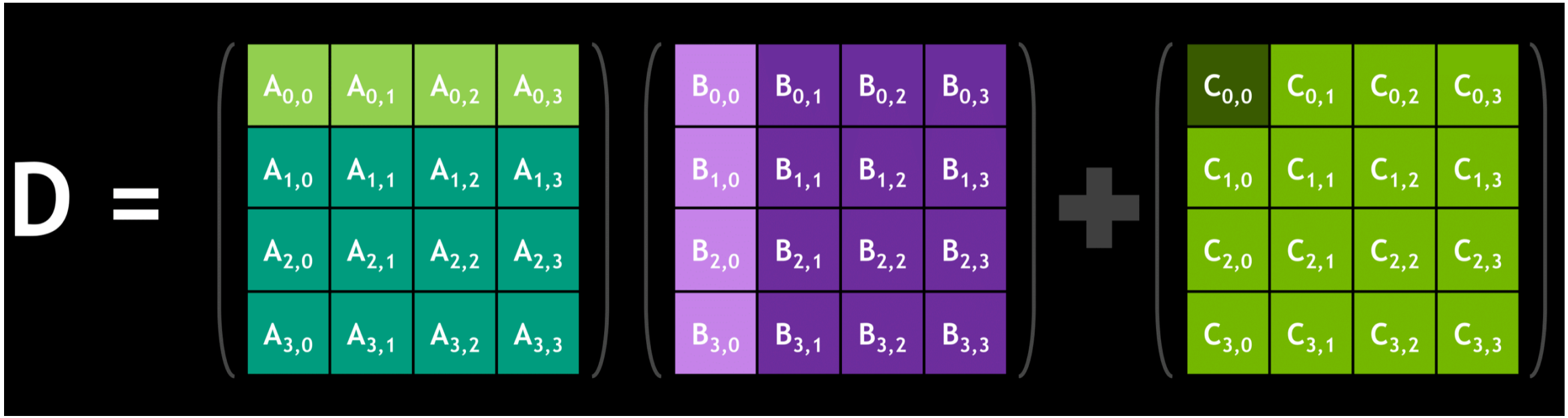


Figure from https://www.anandtech.com/print/13282/nvidia-turing-architecture-deep-dive

Nvidia Turing

NVIDIA GeForce x80 Ti Specification Comparison				
	RTX 2080 Ti Founder's Edition	RTX 2080 Ti	GTX 1080 Ti	GTX 980 Ti
CUDA Cores	4352	4352	3584	2816
ROPs	88	88	88	96
Core Clock	1350MHz	1350MHz	1481MHz	1000MHz
Boost Clock	1635MHz	1545MHz	1582MHz	1075MHz
Memory Clock	14Gbps GDDR6	14Gbps GDDR6	11Gbps GDDR5X	7Gbps GDDR5
Memory Bus Width	352-bit	352-bit	352-bit	384-bit
VRAM	11GB	11GB	11GB	6GB
Single Precision Perf.	14.2 TFLOPs	13.4 TFLOPs	11.3 TFLOPs	6.1 TFLOPs
"RTX-OPS"	78T	78T	N/A	N/A
TDP	260W	250W	250W	250W
GPU	TU102	TU102	GP102	GM200
Architecture	Turing	Turing	Pascal	Maxwell
Manufacturing Process	TSMC 12nm "FFN"	TSMC 12nm "FFN"	TSMC 16nm	TSMC 28nm
Launch Date	09/20/2018	09/20/2018	03/10/2017	06/01/2015
Launch Price	$1199	$999	MSRP: $699 Founders: $699	$649

NVIDIA Turing GPU Comparison				
	TU102	TU104	TU106	GP102
CUDA Cores	4608	3072	2304	3840
SMs	72	48	36	30
Texture Units	288	192	144	240
RT Cores	72	48	36	N/A
Tensor Cores	576	384	288	N/A
ROPs	96	64	64	96
Memory Bus Width	384-bit	256-bit	256-bit	384-bit
L2 Cache	6MB	4MB	4MB	3MB
Register File (Total)	18MB	12MB	9MB	7.5MB
Architecture	Turing	Turing	Turing	Pascal
Manufacturing Process	TSMC 12nm "FFN"	TSMC 12nm "FFN"	TSMC 12nm "FFN"	TSMC 16nm
Die Size	754mm2	545mm2	445mm2	471mm2

Figure from https://www.anandtech.com/print/13282/nvidia-turing-architecture-deep-dive

Nvidia Turing

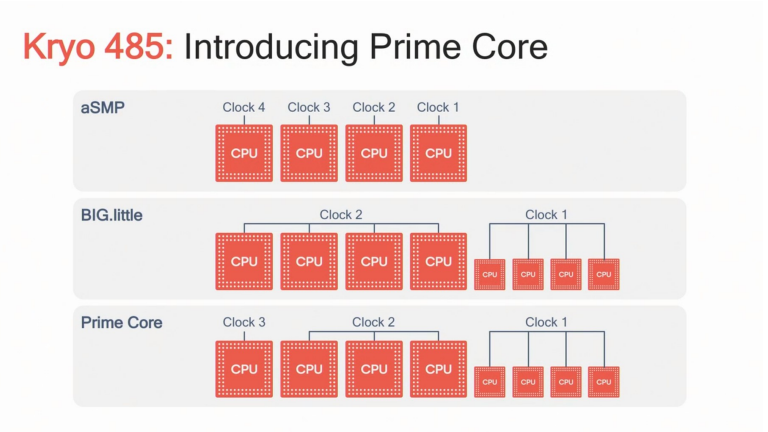
NVIDIA Memory Bandwidth per FLOP (In Bits)			
GPU	**Bandwidth/FLOP**	**Total CUDA FLOPs**	**Total Bandwidth**
RTX 2080	0.36 bits	10.06 TFLOPs	448GB/sec
GTX 1080	0.29 bits	8.87 TFLOPs	320GB/sec
GTX 980	0.36 bits	4.98 TFLOPs	224GB/sec
GTX 680	0.47 bits	3.25 TFLOPs	192GB/sec
GTX 580	0.97 bits	1.58 TFLOPs	192GB/sec

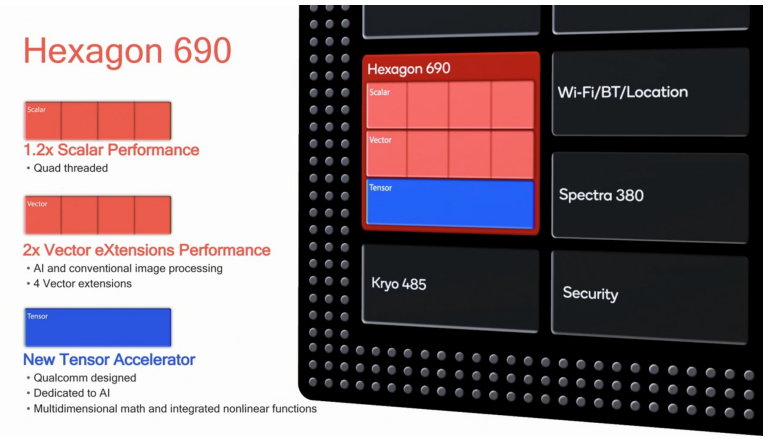
GPU Memory Math: GDDR6 vs. HBM2 vs. GDDR5X						
	NVIDIA GeForce RTX 2080 Ti (GDDR6)	**NVIDIA GeForce RTX 2080 (GDDR6)**	**NVIDIA Titan V (HBM2)**	**NVIDIA Titan Xp**	**NVIDIA GeForce GTX 1080 Ti**	**NVIDIA GeForce GTX 1080**
Total Capacity	11 GB	8 GB	12 GB	12 GB	11 GB	8 GB
B/W Per Pin	14 Gb/s		1.7 Gb/s	11.4 Gbps	11 Gbps	
Chip capacity	1 GB (8 Gb)		4 GB (32 Gb)	1 GB (8 Gb)		
No. Chips/KGSDs	11	8	3	12	11	8
B/W Per Chip/Stack	56 GB/s		217.6 GB/s	45.6 GB/s	44 GB/s	
Bus Width	352-bit	256-bit	3092-bit	384-bit	352-bit	256-bit
Total B/W	616 GB/s	448GB/s	652.8 GB/s	547.7 GB/s	484 GB/s	352 GB/s
DRAM Voltage	1.35 V		1.2 V (?)	1.35 V		

Figure from https://www.anandtech.com/print/13282/nvidia-turing-architecture-deep-dive

Qualcomm Snapdragon 855

- Partial specs
  - CPU
    - 1x A76 derivative at 2.84 GHz with 1x 512 KB L2 (prime core)
    - 3x A76 derivative at 2.42 GHz with 3x 256 KB L2
    - 4x A55 derivative at 1.80 GHz with 4x 128 KB L2
    - 2 MB L3
  - GPU
    - Adreno 640
  - DSP
    - Hexagon 690 with ~ 7 TOPS total
    - 4x scalar threads
    - 4x vector of 1024b each
    - 1 tensor accelerator that can work in parallel
  - Memory
    - 4x 16b LPDDR4x at 2133 MHz for 34.1 GB/s
    - 3 MB system level cache





- Links
  - https://www.qualcomm.com/products/snapdragon-855-mobile-platform
  - https://www.qualcomm.com/media/documents/files/snapdragon-855-mobile-platform-product-brief.pdf
  - https://www.anandtech.com/print/13680/snapdragon-855-going-into-detail
  - https://www.anandtech.com/print/13786/snapdragon-855-performance-preview

Figure from https://www.anandtech.com/print/13786/snapdragon-855-performance-preview

UW VTA

- Versatile tensor accelerator (VTA)
  - Programmable accelerator that exposes RISC like abstractions for compute and memory at the tensor level
  - https://tvm.ai/vta and https://arxiv.org/abs/1807.04188

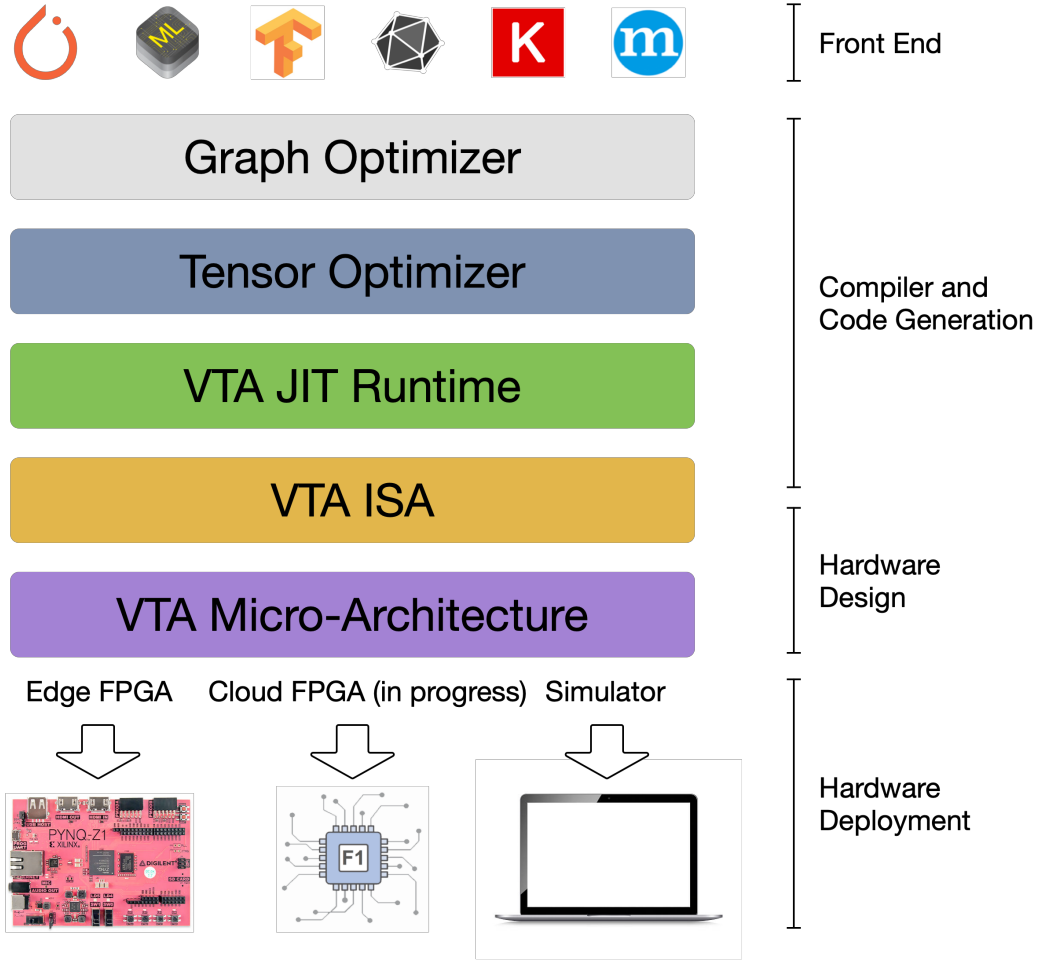


Figure from https://arxiv.org/abs/1807.04188

UW VTA

- Versatile tensor accelerator (VTA)
  - Programmable accelerator that exposes RISC like abstractions for compute and memory at the tensor level
  - https://tvm.ai/vta and https://arxiv.org/abs/1807.04188

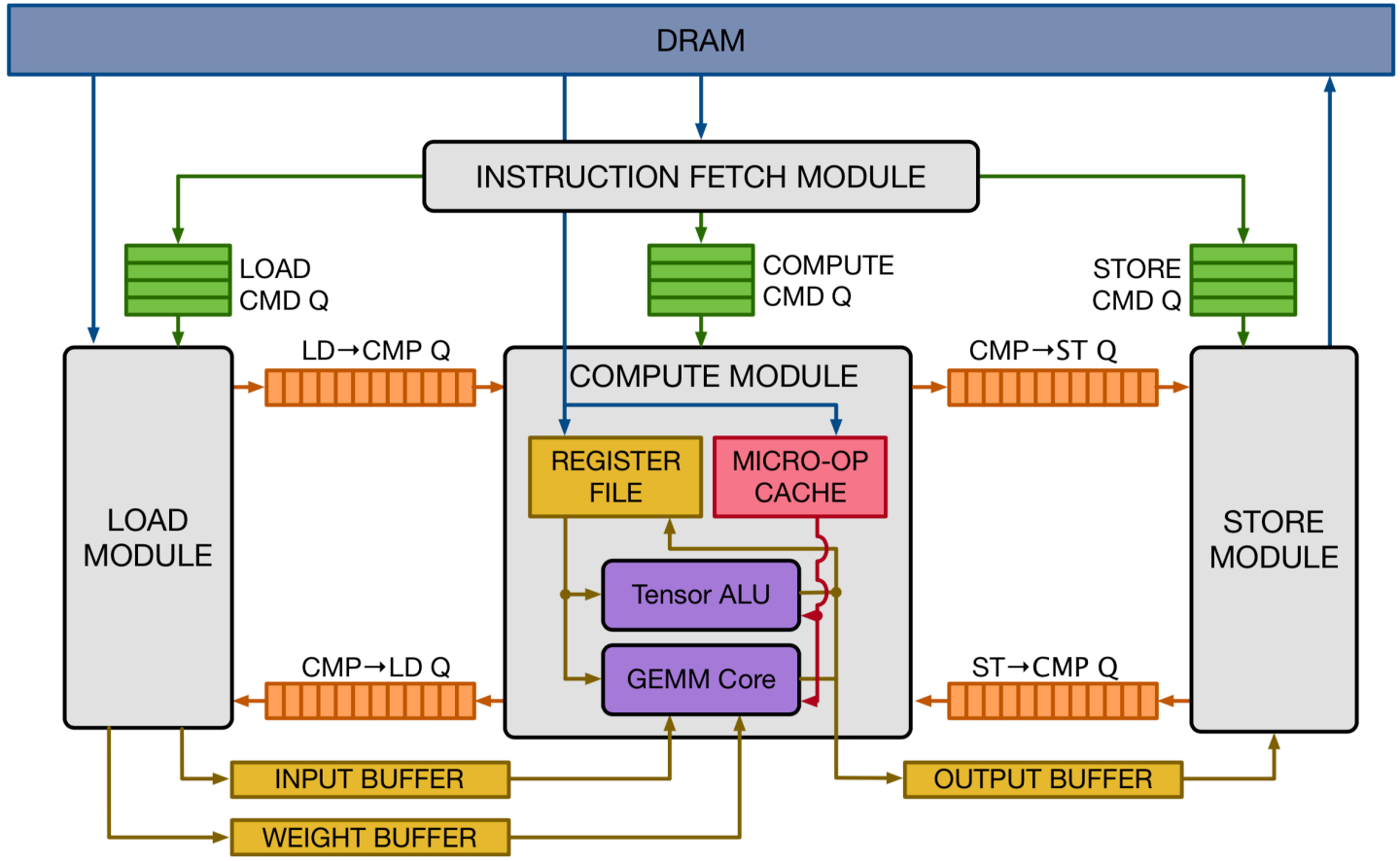


Figure from https://arxiv.org/abs/1807.04188

Xilinx Versal SoC Architecture

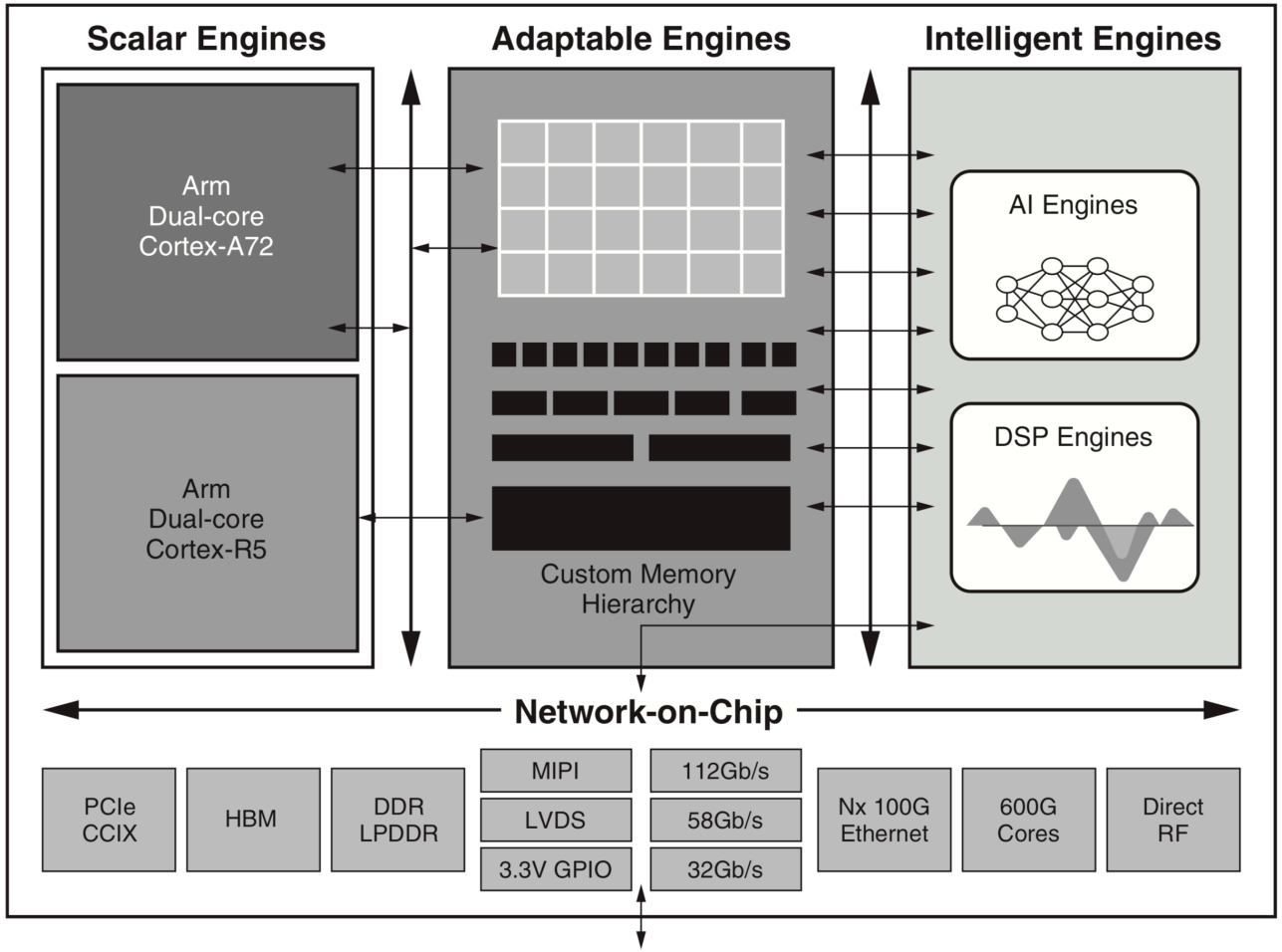


Figure from https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf

Xilinx 2D Array Of AI Engines

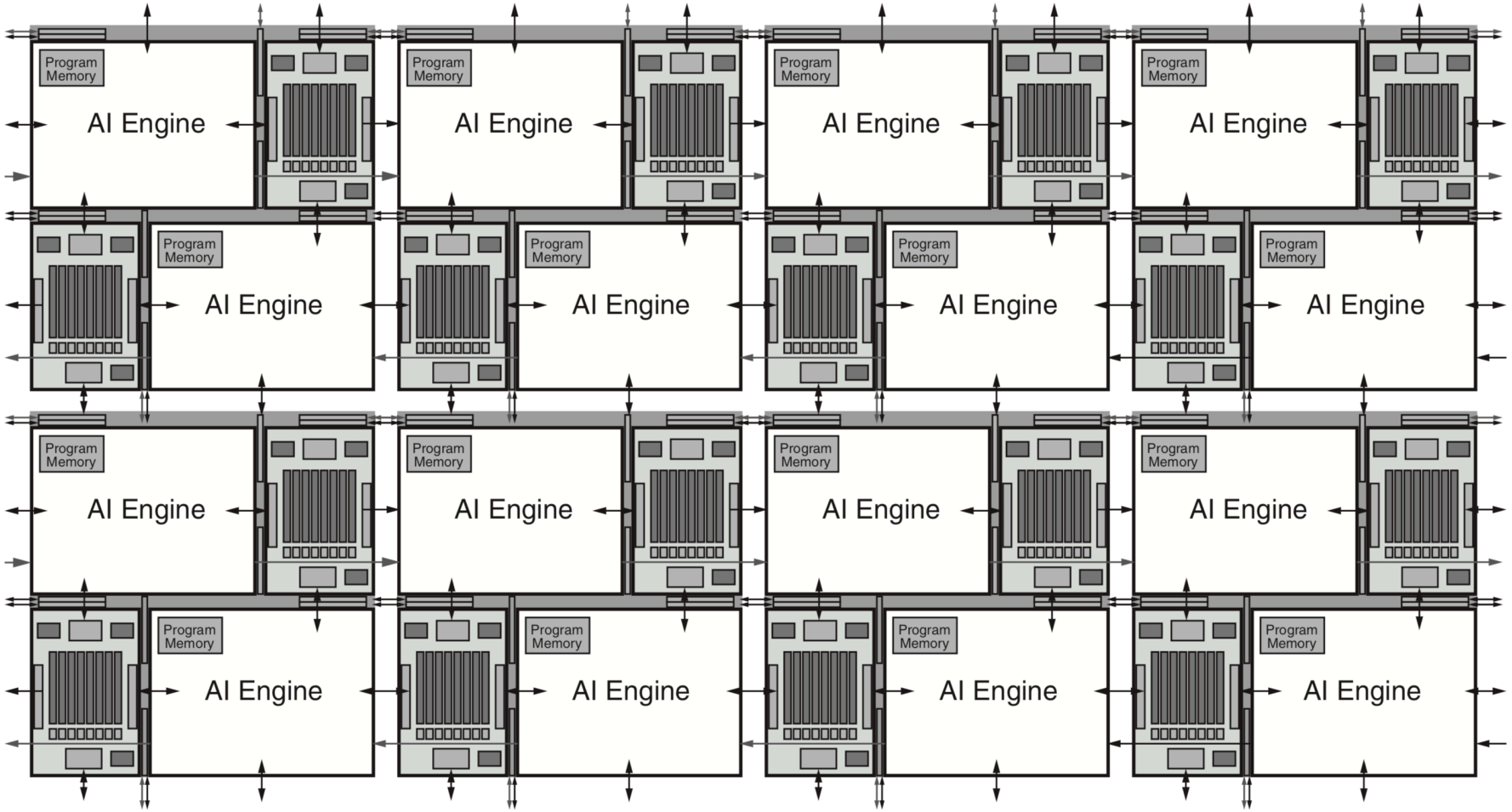


Figure from https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf

Xilinx AI Engine

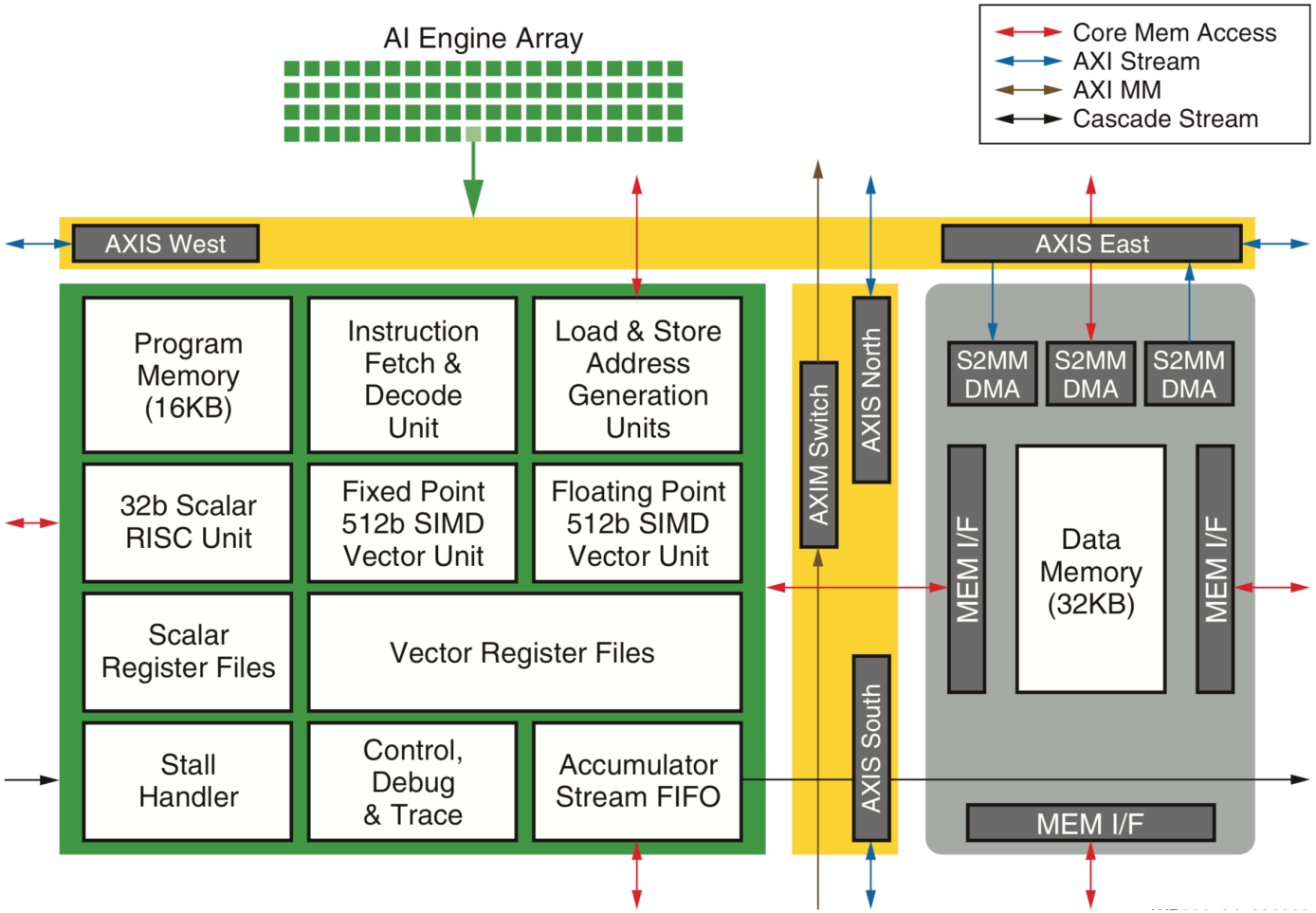


Figure from https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf

References

Networks

- Estimating or propagating gradients through stochastic neurons for conditional computation
  - https://arxiv.org/abs/1308.3432
- Training deep neural networks with low precision multiplications
  - https://arxiv.org/abs/1412.7024
- Hardware-oriented approximation of convolutional neural networks
  - https://arxiv.org/abs/1604.03168
- Quantized neural networks: training neural networks with low precision weights and activations
  - https://arxiv.org/abs/1609.07061
- Mixed precision training
  - https://arxiv.org/abs/1710.03740
- Quantization and training of neural networks for efficient integer-arithmetic-only inference
  - https://arxiv.org/abs/1712.05877
- Mixed precision training of convolutional neural networks using integer operations
  - https://arxiv.org/abs/1802.00930
- Quantizing deep convolutional networks for efficient inference: A whitepaper
  - https://arxiv.org/abs/1806.08342

Networks

- Neural network approximation
  - https://zsc.github.io/megvii-pku-dl-course/slides/Lecture5(Neural%20Network%20Approximation).pdf

- Ristretto: a framework for empirical study of resource-efficient inference in convolutional neural networks
  - http://lepsucd.com/?page_id=621
  - http://lepsucd.com/?page_id=637
  - https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8318896
- High performance ultra-low-precision convolutions on mobile devices
  - https://arxiv.org/abs/1712.02427

- IEEE 754
  - https://en.wikipedia.org/wiki/IEEE_754
- bfloat16 floating-point format
  - https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

Networks

- Optimal brain damage
  - http://yann.lecun.com/exdb/publis/pdf/lecun-90b.pdf
- Optimal brain surgeon and general network pruning
  - https://authors.library.caltech.edu/54981/1/Optimal%20Brain%20Surgeon%20and%20general%20network%20pruning.pdf
- Efficient and accurate approximations of nonlinear convolutional networks
  - https://arxiv.org/abs/1411.4229
- Accelerating very deep convolutional networks for classification and detection
  - https://arxiv.org/abs/1505.06798
- Learning efficient convolutional networks through network slimming
  - https://arxiv.org/abs/1708.06519
- To prune or not to prune: exploring the efficacy of pruning for model compression
  - https://openreview.net/pdf?id=Sy1iIDkPM

Software

- Netscope CNN analyzer
  - https://dgschwend.github.io/netscope/quickstart.html
- Model zoo
  - https://modelzoo.co
- Python review
  - http://web.stanford.edu/class/cs224n/lectures/python-review.pdf
- Python numpy tutorial
  - http://cs231n.github.io/python-numpy-tutorial/

Software

- An introduction to TensorFlow
  - http://web.stanford.edu/class/cs224n/lectures/lecture6.pdf
  - http://web.stanford.edu/class/cs224n/readings/tensorflow_tutorial_code.zip
- Introduction to TensorFlow
  - https://www.youtube.com/watch?v=PicxU81owCs#t=3m16s
- TensorFlow tutorial
  - http://web.stanford.edu/class/cs224s/lectures/Tensorflow-tutorial.pdf
  - https://github.com/pbhatnagar3/cs224s-tensorflow-tutorial
- TensorFlow quantization aware training
  - https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/quantize
- TensorFlow models research slim
  - https://github.com/tensorflow/models/tree/master/research/slim
  - https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md
- TensorFlow models official
  - https://github.com/tensorflow/models/tree/master/official

Software

- Introduction to PyTorch code examples
  - https://cs230-stanford.github.io/pytorch-getting-started.html
- Hardware and software
  - http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf
- Welcome to PyTorch tutorials
  - https://pytorch.org/tutorials/
- PyTorch examples
  - https://github.com/jcjohnson/pytorch-examples
- The incredible PyTorch
  - https://github.com/ritchieng/the-incredible-pytorch
- PyTorch docs torchvision models
  - https://pytorch.org/docs/stable/torchvision/models.html
- Pretrained models PyTorch
  - https://github.com/Cadene/pretrained-models.pytorch
- Pytorch mobilenet
  - https://github.com/marvis/pytorch-mobilenet

Software

- Glow: graph lowering compiler techniques for neural networks
  - https://arxiv.org/abs/1805.00907
- Compiler for neural network hardware accelerators
  - https://github.com/pytorch/glow
- Glow
  - https://facebook.ai/developers/tools/glow
- Halide a language for fast, portable computation on images and tensors
  - http://halide-lang.org
- Loopy: transformation-based generation of high-performance CPU/GPU code
  - https://github.com/inducer/loopy
- TVM: an automated end-to-end optimizing compiler for deep learning
  - https://arxiv.org/abs/1802.04799
- End to end deep learning compiler stack for CPUs, GPUs and specialized accelerators
  - https://tvm.ai
- Designing computer systems for software 2.0
  - https://iscaconf.org/isca2018/docs/Kunle-ISCA-Keynote-2018.pdf

Software

- Learning to optimize tensor programs
  - https://arxiv.org/abs/1805.08166

- TensorFlow XLA overview
  - https://www.tensorflow.org/extend/xla/
- Tensor comprehensions: framework-agnostic high-performance machine learning abstractions
  - https://arxiv.org/abs/1802.04730
- Tensor comprehensions
  - https://facebook.ai/developers/tools/tensorcomprehensions

- ONNX
  - https://facebook.ai/developers/tools/onnx
- Open neural network exchange
  - https://github.com/onnx

Software

- The LLVM Compiler Infrastructure
  - https://llvm.org
- LLVM: a compilation framework for lifelong program analysis & transformation
  - https://llvm.org/pubs/2004-01-30-CGO-LLVM.html

- OpenMPI: open source high performance computing
  - https://www.open-mpi.org
- OpenMP
  - https://www.openmp.org
- OpenCL
  - https://www.khronos.org/opencl/
- CUDA
  - https://developer.nvidia.com/cuda-zone

Software

- BLAS (basic linear algebra subprograms)
  - http://www.netlib.org/blas/
- Automatically tuned linear algebra software (ATLAS)
  - http://math-atlas.sourceforge.net
- BLAS-like library instantiation software framework
  - https://github.com/flame/blis
- High-performance object-based library for DLA computations
  - https://github.com/flame/libflame
- FFTW
  - http://www.fftw.org
- Spiral software / hardware generation for performance
  - http://www.spiral.net/index.html
- cuDNN
  - https://developer.nvidia.com/cudnn

Hardware

- John Hennessy and David Patterson 2017 ACM A.M. Turing award lecture
  - https://www.youtube.com/watch?v=3LVeEjsn8Ts
- The future of computing: a conversation with John Hennessy (Google I/O '18)
  - https://www.youtube.com/watch?v=Azt8Nc-mtKM

- Amdahl's law and its proof
  - https://www.geeksforgeeks.org/computer-organization-amdahls-law-and-its-proof/
- Performance modeling
  - http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/SPRING-2013/Lect07_short.pdf
- Roofline: an insightful visual performance model for multicore architectures
  - https://dl.acm.org/citation.cfm?id=1498785

Hardware

- Moore's law and Dennard scaling
  - http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture15.pdf
- The future of microprocessors
  - https://cacm.acm.org/magazines/2011/5/107702-the-future-of-microprocessors/
- The accelerator store: a shared memory framework for accelerator-based systems
  - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.226.994&rep=rep1&type=pdf
- Dark silicon and the end of multicore scaling
  - https://www.cc.gatech.edu/~hadi/doc/paper/2012-toppicks-dark_silicon.pdf
- Is dark silicon useful?
  - https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6241647&tag=1
- Dark memory and accelerator-rich system optimization in the dark silicon era
  - https://arxiv.org/abs/1602.04183
- Co-designing accelerators and soc interfaces using gem5-aladdin
  - http://vlsiarch.eecs.harvard.edu/wp-content/uploads/2016/08/shao_micro2016.pdf

Hardware

- The impact of Moore's law and loss of Dennard scaling
  - https://indico.cern.ch/event/397113/contributions/1837780/attachments/1215934/1775678/Talk_2016-01-21.pdf
- The scaling of MOSFETS, Moore's law, and ITRS
  - http://userweb.eng.gla.ac.uk/fikru.adamu-lema/Chapter_02.pdf
- Integrated power management, leakage control and process compensation technology for advanced processes
  - https://www.design-reuse.com/articles/20296/power-management-leakage-control-process-compensation.html
- An introduction to computation technologies in deep learning
  - https://zsc.github.io/megvii-pku-dl-course/slides18/dl-comp-tech.pdf

Hardware

- Dynamic random-access memory
  - https://en.wikipedia.org/wiki/Dynamic_random-access_memory
- Static random-access memory
  - https://en.wikipedia.org/wiki/Static_random-access_memory

Hardware

- Vector-matrix multiply and winner-take-all as an analog classifier
  - https://ieeexplore.ieee.org/document/6519956
- Evaluation of an analog accelerator for linear algebra
  - https://ieeexplore.ieee.org/document/7551423
- Charge-mode parallel architecture for vector-matrix multiplication
  - https://ieeexplore.ieee.org/document/974781
- Analysis and design of a passive switched-capacitor matrix multiplier for approximate computing
  - https://ieeexplore.ieee.org/document/7579580
- SysML 18: Jonathan Binas, Analog electronic deep networks for fast and efficient inference
  - https://www.youtube.com/watch?reload=9&v=8t0Yunt5kE4

Hardware

- Fast algorithms for convolutional neural networks
  - https://arxiv.org/abs/1509.09308
- Efficient processing of deep neural networks: a tutorial and survey
  - https://arxiv.org/abs/1703.09039
- Tutorial on hardware architectures for deep neural networks
  - http://eyeriss.mit.edu/tutorial.html
- Understanding the limitations of existing energy-efficient design approaches for deep neural networks
  - http://www.rle.mit.edu/eems/wp-content/uploads/2018/02/2018_SysML_final.pdf
- Eyeriss v2: a flexible and high-performance accelerator for emerging deep neural networks
  - https://arxiv.org/abs/1807.07928
- NVDLA primer
  - http://nvdla.org/primer.html
- The Nvidia Turing GPU architecture deep dive: prelude to GeForce RTX
  - https://www.anandtech.com/print/13282/nvidia-turing-architecture-deep-dive

Hardware

- ARM system IP
  - https://developer.arm.com/products/system-ip
- ARM details ”Project Trillium" machine learning processor architecture
  - https://www.anandtech.com/show/12791/arm-details-project-trillium-mlp-architecture
- Cadence announces the Tensilica DNA 100 IP: bigger artificial intelligence
  - https://www.anandtech.com/show/13377/cadence-announces-tensilica-dna-100-a-bigger-nn-ip
- An in-depth look at Google’s first tensor processing unit (TPU)
  - https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu
- In-datacenter performance analysis of a tensor processing unit
  - https://arxiv.org/abs/1704.04760
- First in-depth look at Google’s TPU architecture
  - https://www.nextplatform.com/2017/04/05/first-depth-look-googles-tpu-architecture/
- Under the hood of Google’s TPU2 machine learning clusters
  - https://www.nextplatform.com/2017/05/22/hood-googles-tpu2-machine-learning-clusters/
- Tearing apart Google’s TPU 3.0 AI coprocessor
  - https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor/

Hardware

- Xilinx AI engines and their applications
  - https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf
- Versal: the first adaptive compute acceleration platform (ACAP)
  - https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf