Implementation

Arthur J. Redfern
arthur.redfern@utdallas.edu
Mar 11, 2019
Mar 13, 2019
Mar 25, 2019

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

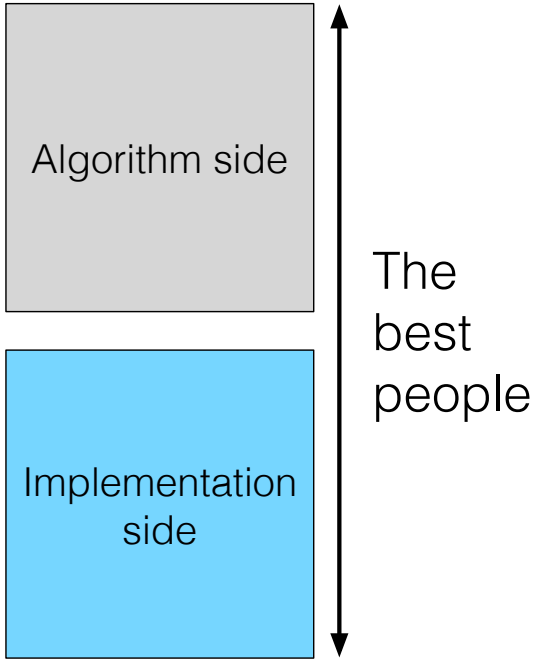
Outline

- Motivation
- Networks
- Hardware
- Software
- Performance

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



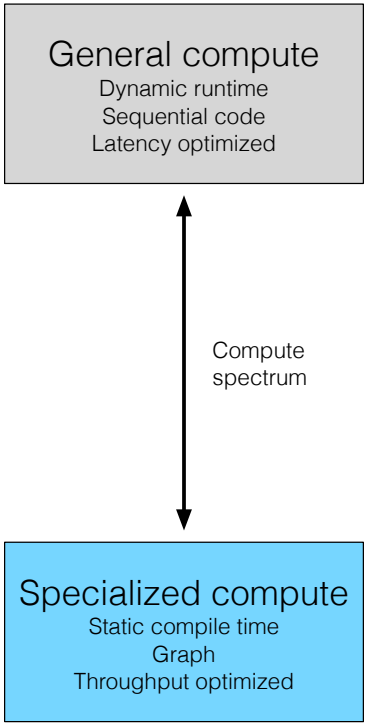
The Future Of Hardware And Software

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, ...

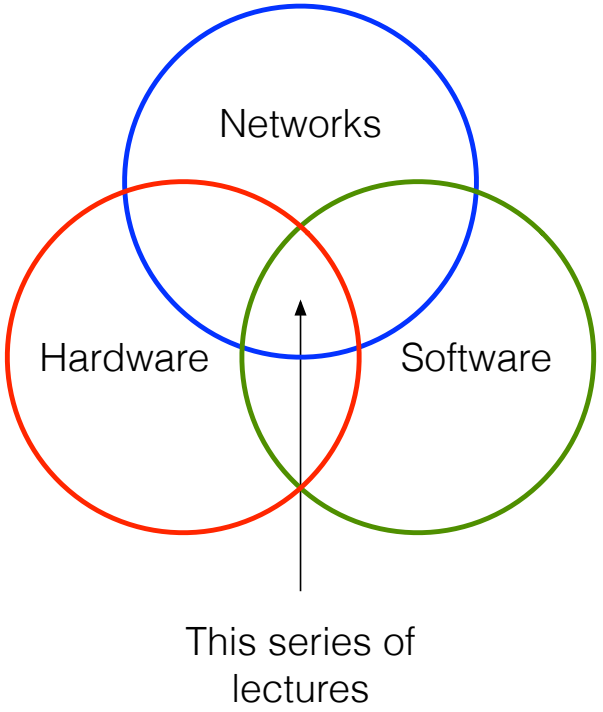


Co Design For Optimality

- Design networks for optimal performance on hardware
  - Layer sizes and operations designed to efficiently map to hardware
  - Quantization and simplification to reduce memory, reduce data movement and improve compute
- Design hardware to be an optimal target for networks
  - Memory sized such that most feature maps remain on device
  - DMA allowing parallel background data movement with transformation for remaining off device transfers
  - Computational primitive approach to big compute for ASIC efficiency with mathematical generality
  - Small general host approach for future proofing
  - Deterministic control via sequencing through a low level compile time optimized graph
- Design software to optimally map networks to hardware
  - High level hardware agnostic graph as a starting point
  - Graph compiler creating low level graphs with edges mapped to memory and nodes mapped to hardware accelerators
  - Runtime bootstrap, initialization and execution graphs with full control of hardware

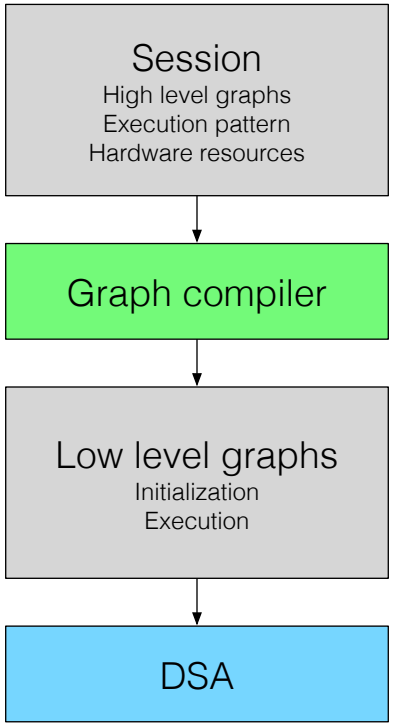
A Lecturer's Apology

- Presentation of material in this lecture is sequential
  - Networks
  - Hardware
  - Software
- But all of these topics are actually optimized together at the same time
  - As such, there are dependencies in the material
- Presentation / review 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



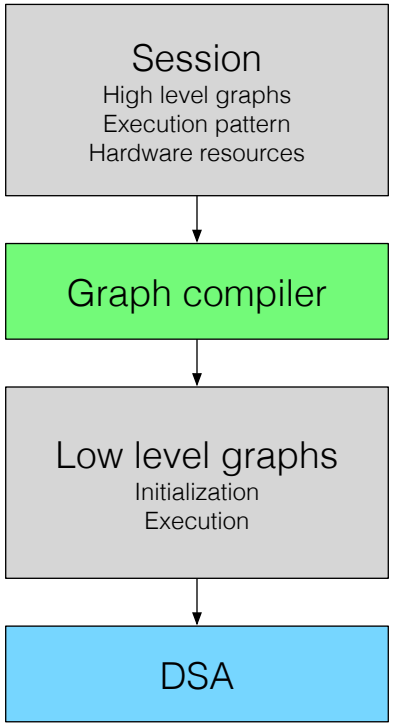
Software Preview

- High level graphs
  - Network specification
- Session
  - The graph compiler input
  - High level graphs + execution pattern
  - Hardware resources
- Graph compiler
  - Compile high level graphs to low level graphs
  - Edges are memory and nodes are operators
  - Separate initialization and execution graphs
  - Exploits compile time information
  - Everything is on the graph to simplify hardware
  - Fully deterministic



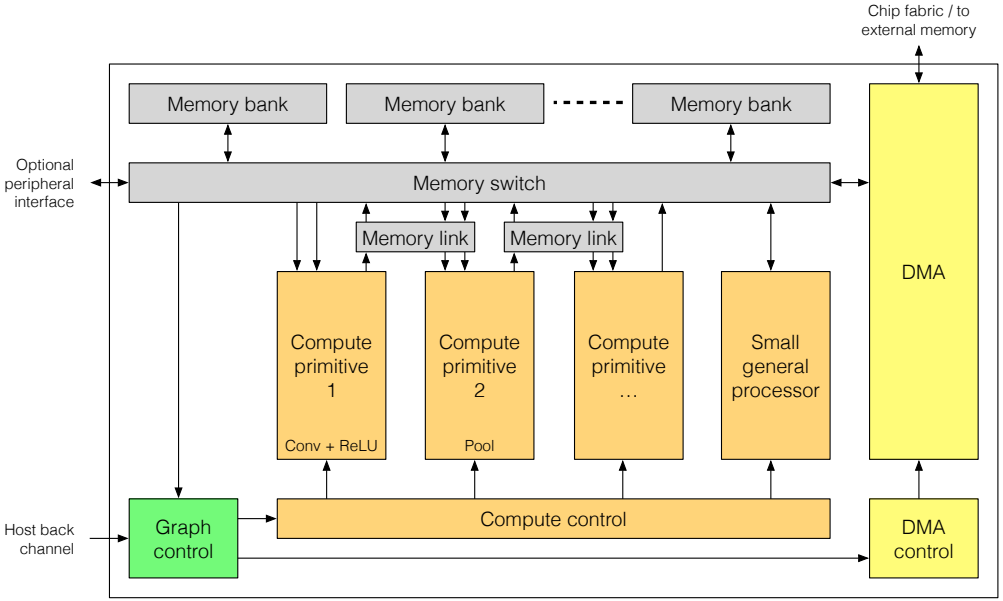
Software Preview

- Low level graphs
  - Map 1 to 1 to hardware
  - Node descriptors encode structure
  - Node instructions control operations
  - Operations include instruction movement, data movement and compute
- Runtime
  - Initialization
    - Called 1x
    - Link static data
    - Run initialization graph
  - Execution
    - Called for each new input
    - Link dynamic data
    - Run execution graph



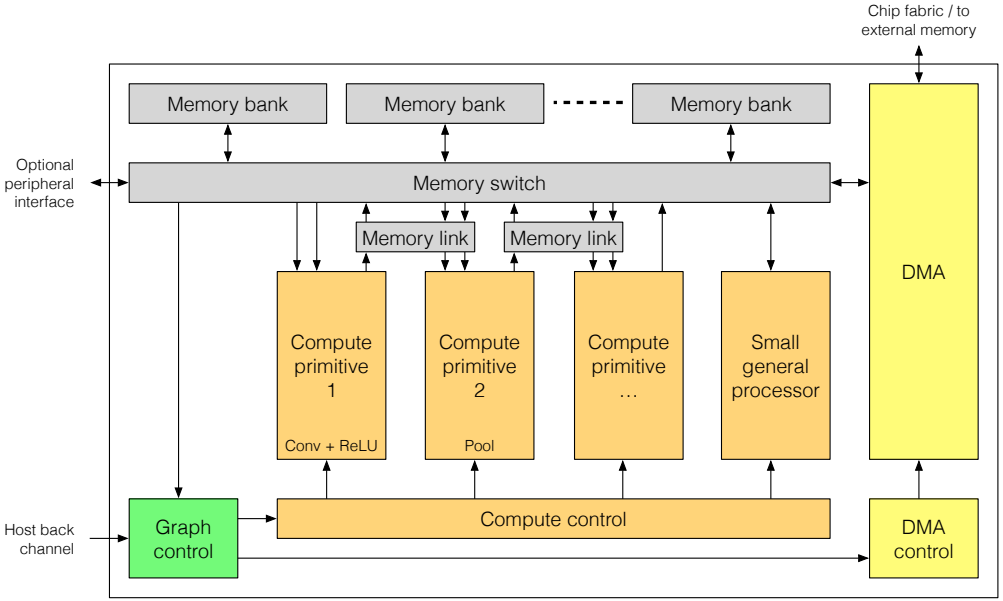
Hardware Preview

- Graph control
  - Uses node descriptors to sequence node instructions that control compute and DMA operations
- Compute and DMA control
  - Parallel compute and DMA control allows parallel data movement and compute
  - Used in a ping pong fashion for efficiency
- Memory
  - Sized to allow most feature maps to remain on device



Hardware Preview

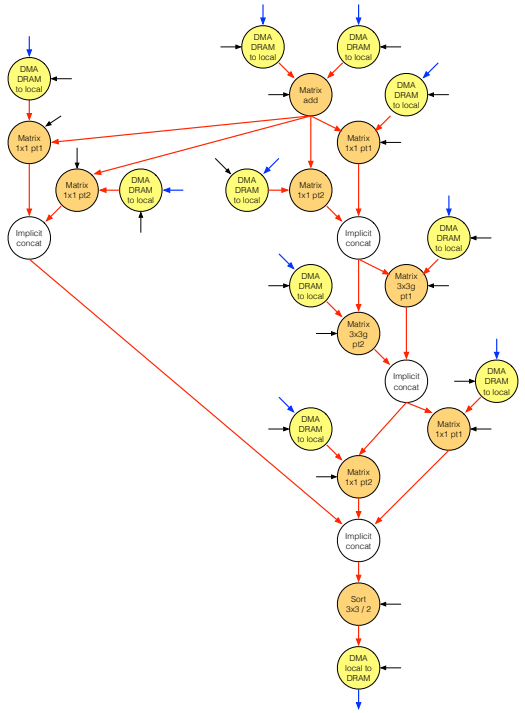
- Compute primitives
  - Use node instructions to setup and execute compute primitives
  - Compute primitives implement low level graph compute operations
  - Compute primitives include matrix, sort and a small general processor
- DMA primitives
  - Use node instructions to setup and execute DMA primitives
  - DMA primitives implement low level graph internal / external data movement operations
  - DMA primitives include multidimensional transformations and compression



Networks

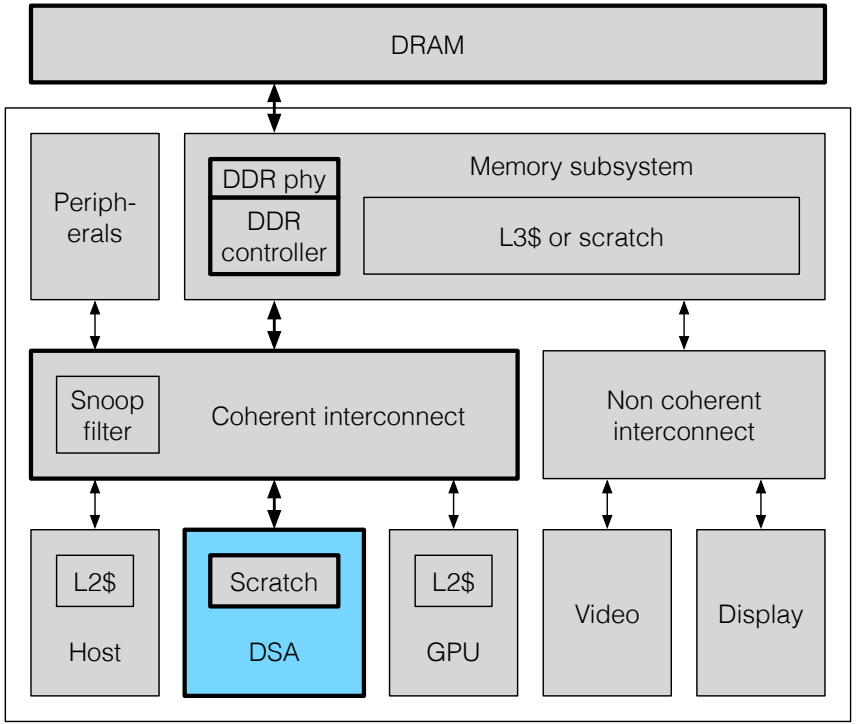
Bookkeeping

- 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 =
$(N_i + N_o) L_r L_c$

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

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

Quantization

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

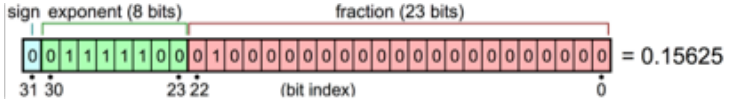
- Common data formats
  - 64 bit float 53.11 (IEEE 754 double)
  - 32 bit float 24.8 (IEEE 754 single)
  - 16 bit float 11.5 (IEEE 754 half; more precision less range than bfloat16)
  - 16 bit float 8.8 (bfloat16; more range less precision than IEEE 754 half)
  - 16 bit fixed (signed and unsigned)
  - 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

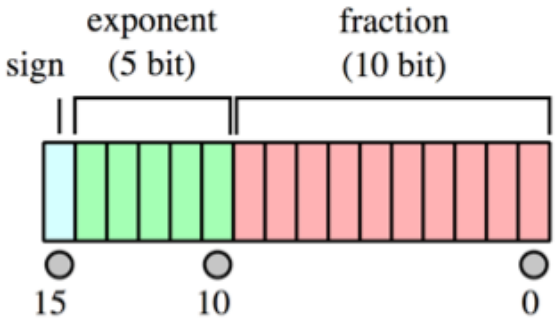
Data Formats For Network Training

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

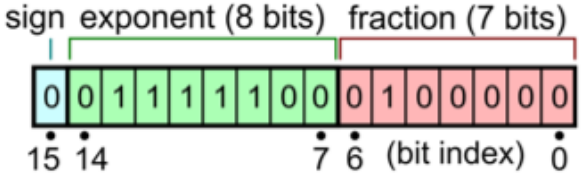
- IEEE 754 float 32: 1 bit sign, 8 bits exponent, 23 bits significand
  - Ignoring special values signaled by exponent values of 0 and 255
  - Value = sign x range x precision
    = $(-1)^{sign}$ x $2^{exponent - 127}$ x $1.significand_2$
    = {–1, 1} x $2^{\{-126, \ldots, 127\}}$ x $[1, 2)_{23 \text{ bits precision}}$



- 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 = $(-1)^{sign}$ x $2^{exponent - 15}$ x $1.significand_2$
    = {–1, 1} x $2^{\{-14, \ldots, 15\}}$ x $[1, 2)_{10 \text{ bits precision}}$



- bfloat 16: 1 bit sign, 8 bits exponent, 7 bits significand
  - Ignoring special values signaled by exponent values of 0 and 255
  - Value = $(-1)^{sign}$ x $2^{exponent - 127}$ x $1.significand_2$
    = {–1, 1} x $2^{\{-126, \ldots, 127\}}$ x $[1, 2)_{7 \text{ bits precision}}$



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

- Signed example for $\{\mathbf{X}_q, s_X\} = quantize(\mathbf{X}, bits)$
  - Target range of $\{-2^{bits-1} - 1, \ldots, 2^{bits-1} - 1\}$
    - 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 $\{0, \ldots, 2^{bits} - 1\}$
    - 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)

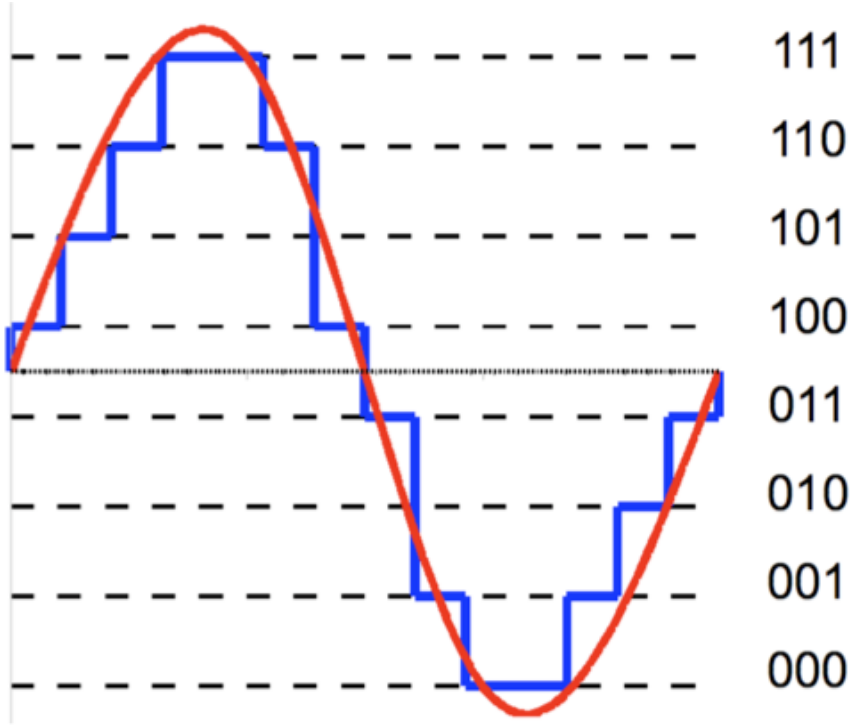


Figure from https://en.wikipedia.org/wiki/Quantization_(signal_processing)

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
  - $\mathbf{Y}_q \approx s_c ((\mathbf{H} / s_H)(\mathbf{X} / s_X) + (\mathbf{V} / (s_H s_X)))$
    $\approx (s_c / (s_H s_X)) (\mathbf{H X} + \mathbf{V})$

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

- This implies we can approximate floating point CNN style 2D convolution $\mathbf{Y} = \mathbf{H X} + \mathbf{V}$ 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)

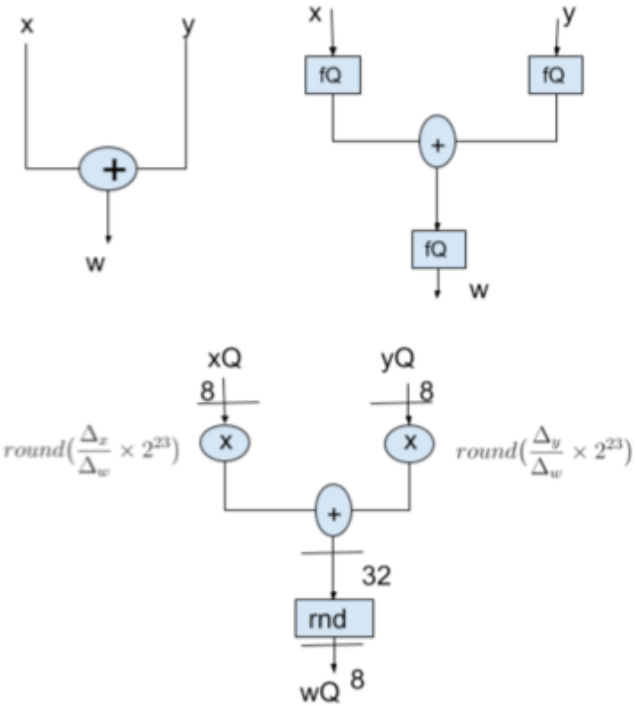


Figure from https://arxiv.org/abs/1806.08342

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

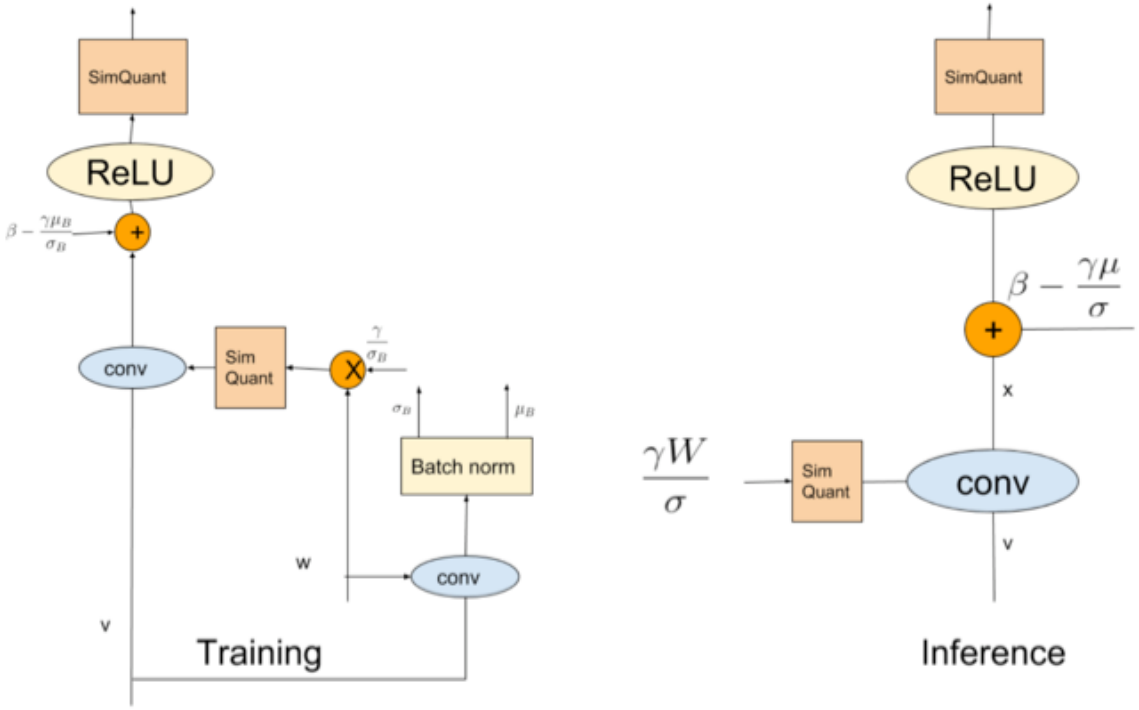


Figure from https://arxiv.org/abs/1806.08342

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

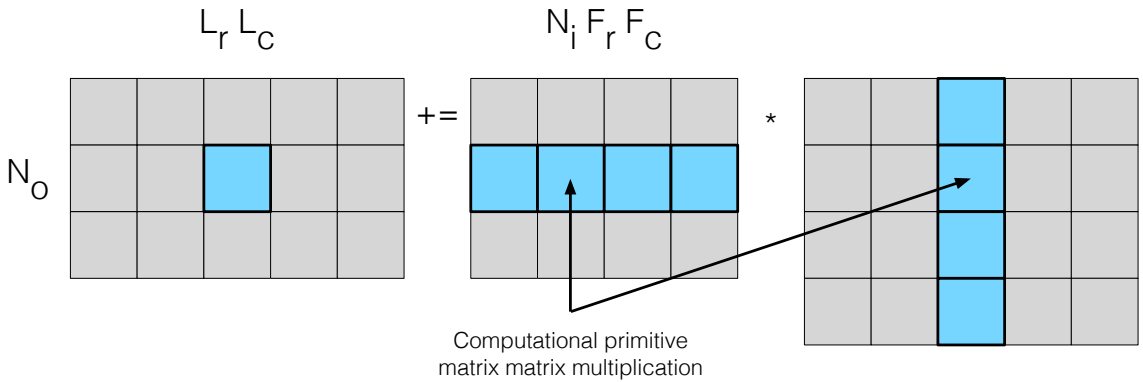
For more details see:

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

Efficient Sizing

CNN Style 2D Convolution Computation

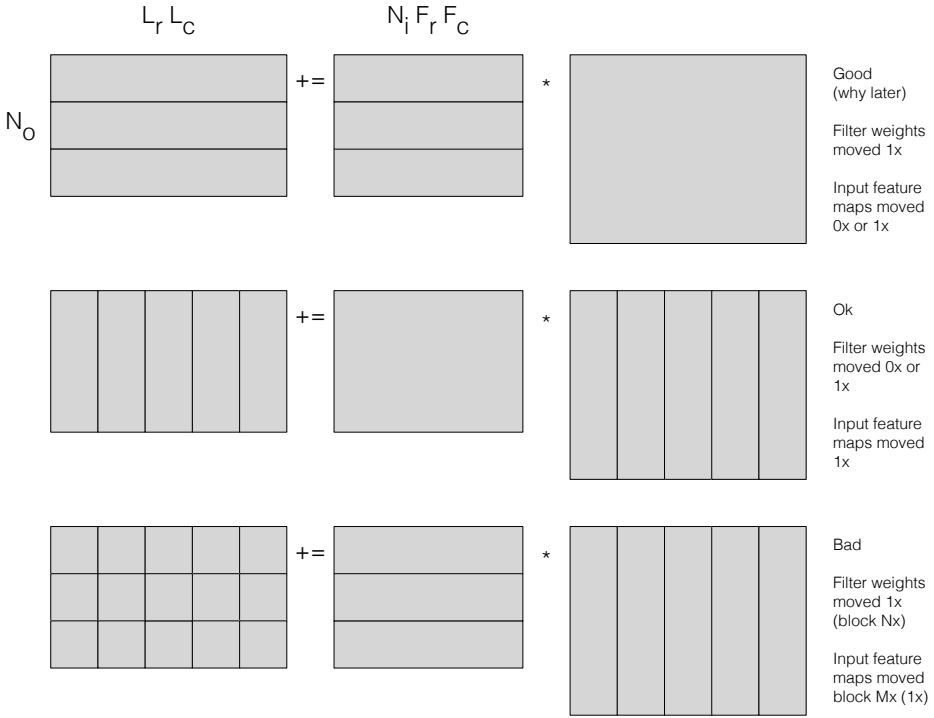
- A preview
  - We're going to specify hardware with an optimized matrix matrix multiplication primitive
  - Large matrix matrix multiplication is going to be implemented via tiled smaller matrix matrix multiplication
  - Data movement limits compute
- Tiling efficiency
  - In this figure N_o, L_r L_c 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



Simplification

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

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
    - 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

Hardware

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

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

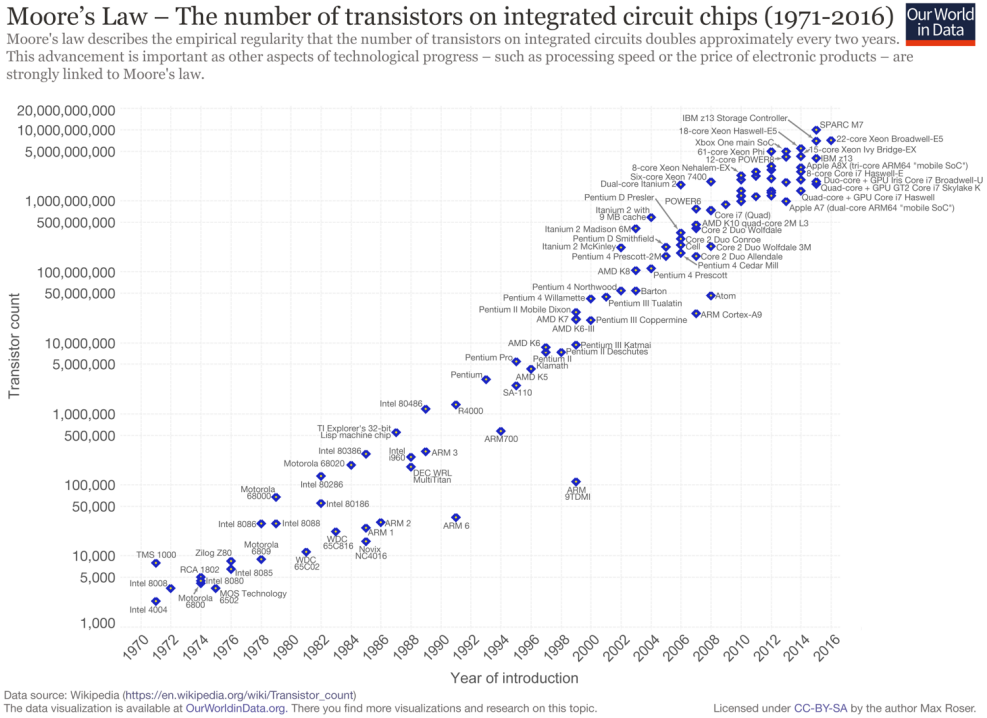


Figure from https://en.wikipedia.org/wiki/Moore%27s_law

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

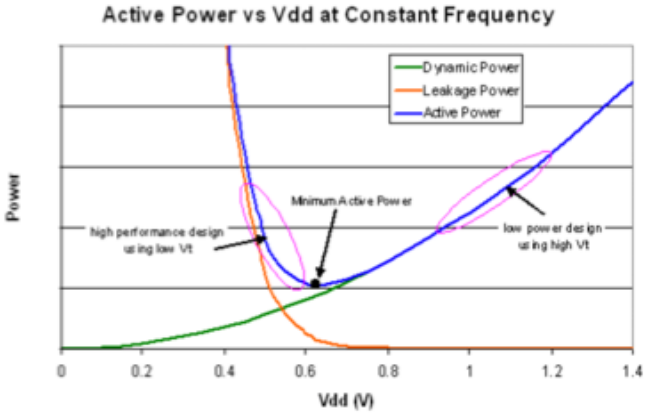
- C' = C/2
- D' = 4D — 4x transistors
- E' = E/8
- f' = 2f — 2x frequency
- P' = P — 1x power

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

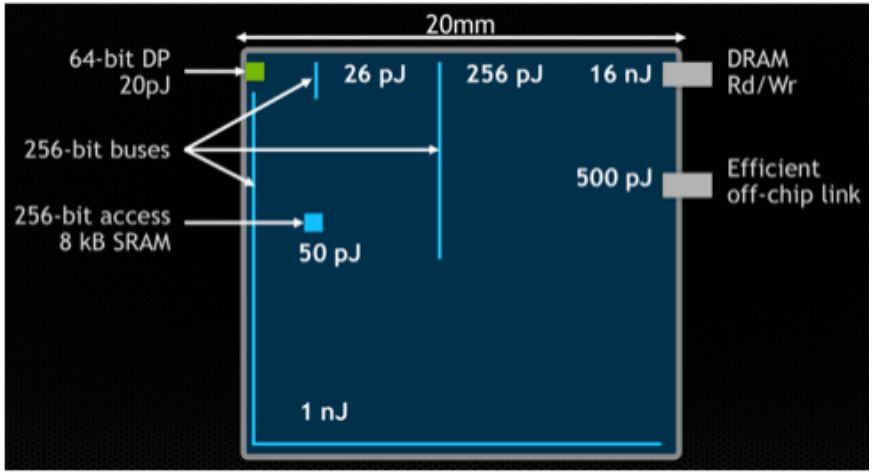
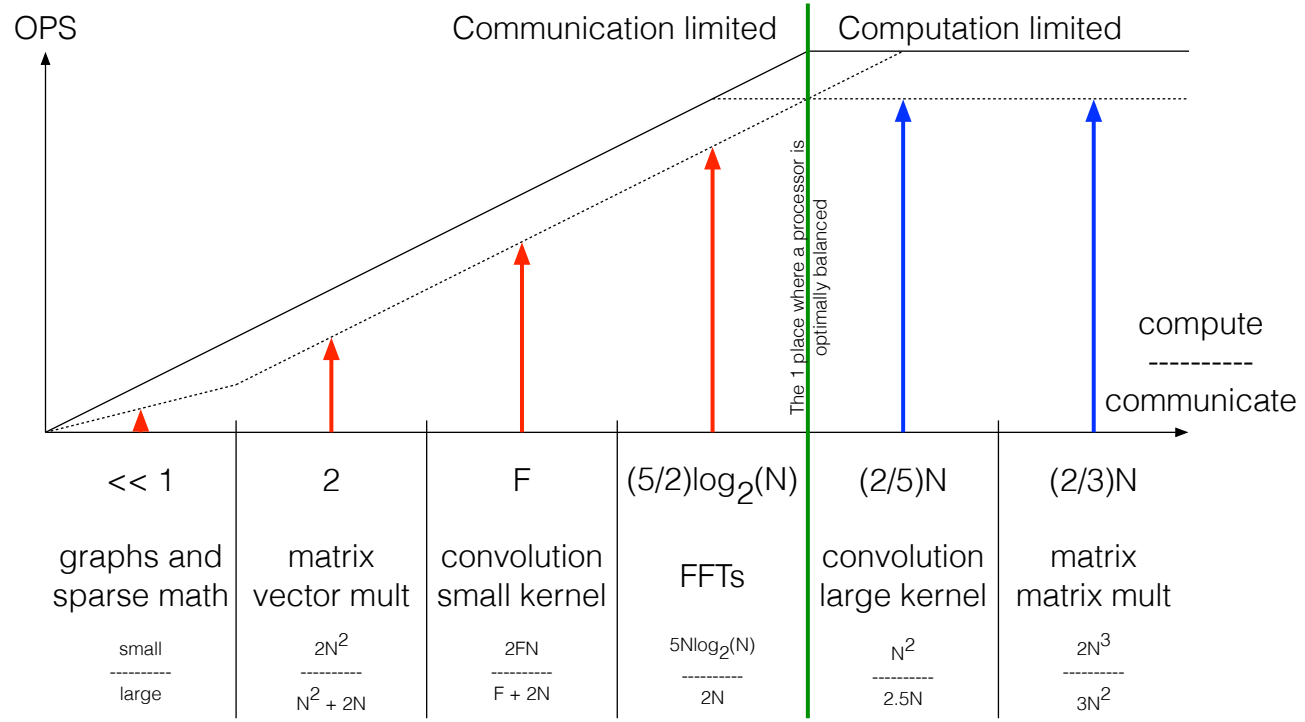


Figure from B. Dally SC415 Nvidia the path to exascale

Roofline Model



Putting 1 And 1 And 1 And 1 Together

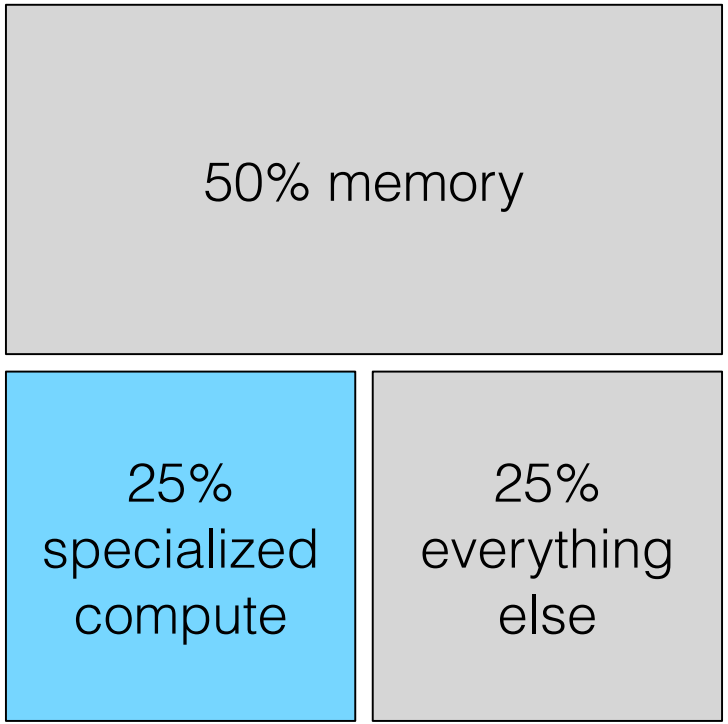
- Power is the problem, only part of the device can be on at a time, moving data takes the most power and compute is limited by data movement

- 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

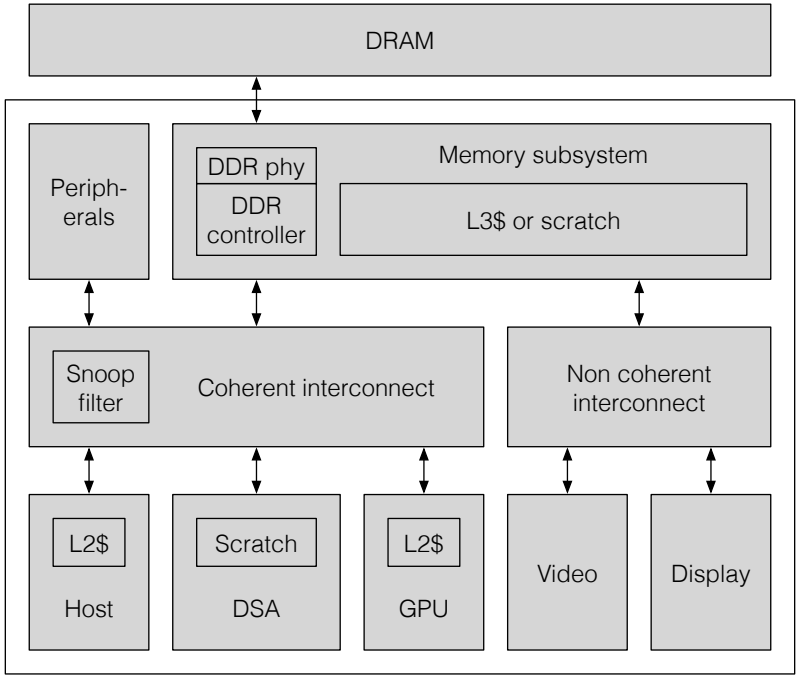
- Big compute device trends ≈
  - 50% memory
    - If off device data movement suddenly became very low power and very high throughput then this number will reduce
  - 25% optimized compute
  - 25% everything else

- 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



A Generic SoC Architecture

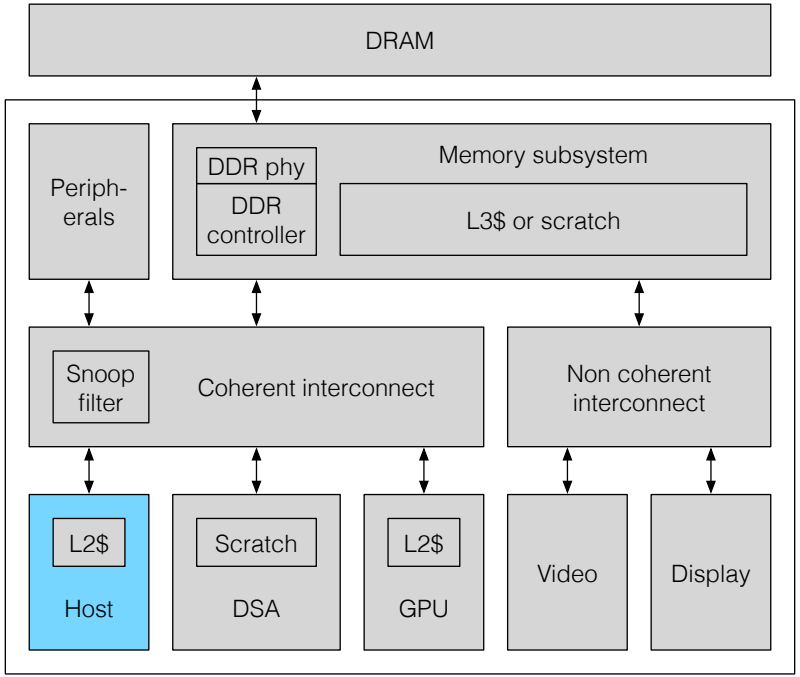
- External DRAM (not part of the SoC)
- Coherent
  - Host L2$
  - GPU L2$
  - L3$
  - DRAM (part)
- IO coherent
  - Peripherals
  - DSA
- Non coherent
  - Video
  - Display
  - L3 scratch
  - DRAM (part)



Host

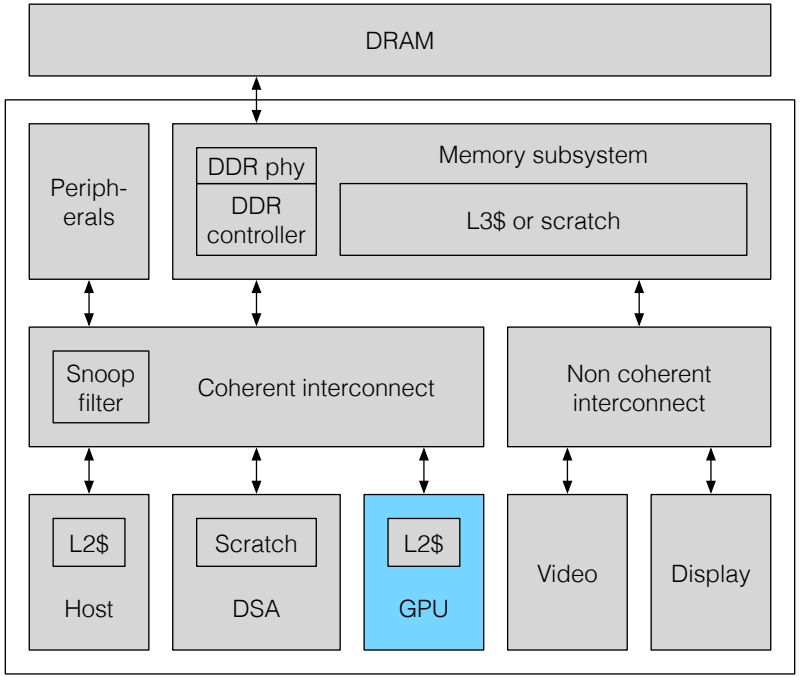
- General compute that can do everything
  - Example architectures: ARM, RISC-V, x86
  - This is what you want for optimizing the performance of large amounts of runtime dynamic hardware agnostic code
  - This is not what you want for optimizing the performance of small amounts compile time static hardware 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

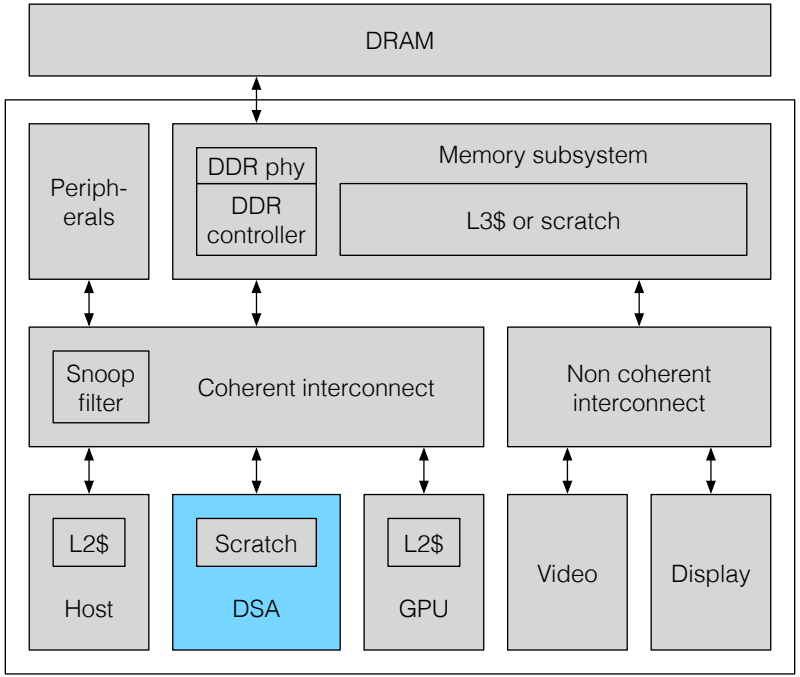


Domain Specific Architecture

- 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

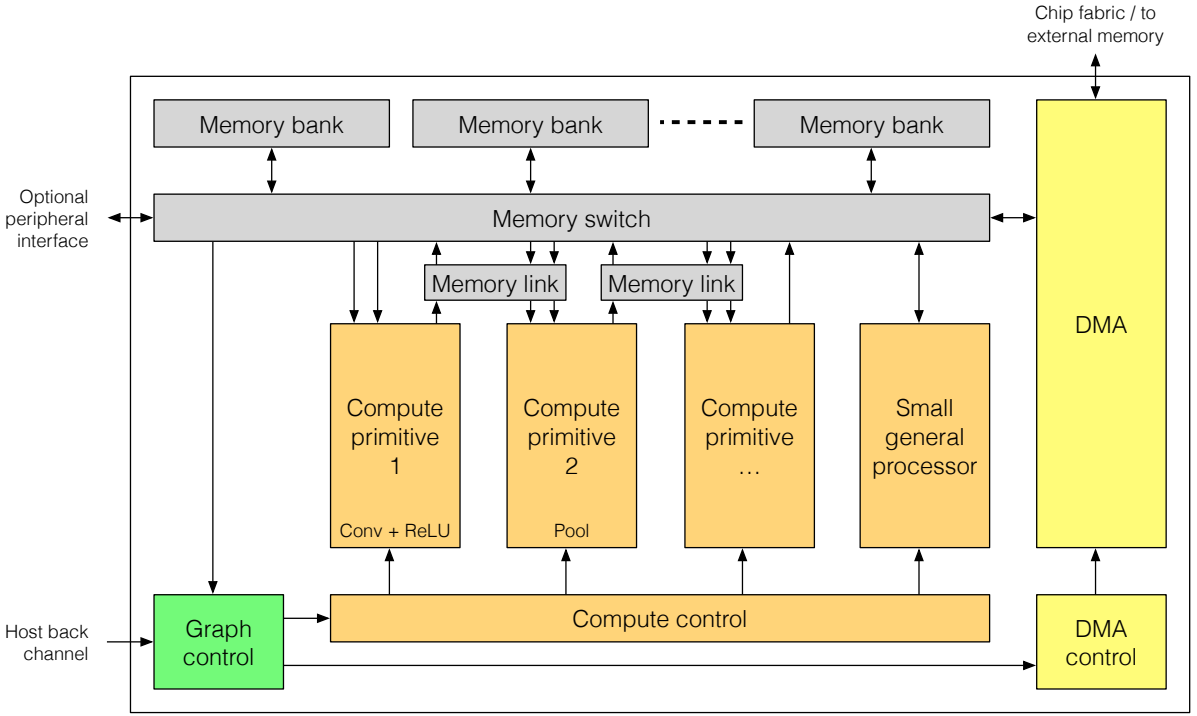
- Can include on the coherent or non coherent interconnect depending on the application
  - Coherent
  - IO coherent
  - Non coherent

- Question: What is the optimal DSA for xNNs?
  - What type of control is needed?
  - What type of memory is needed?
  - What type of data movement is needed?
  - What type of compute is needed?



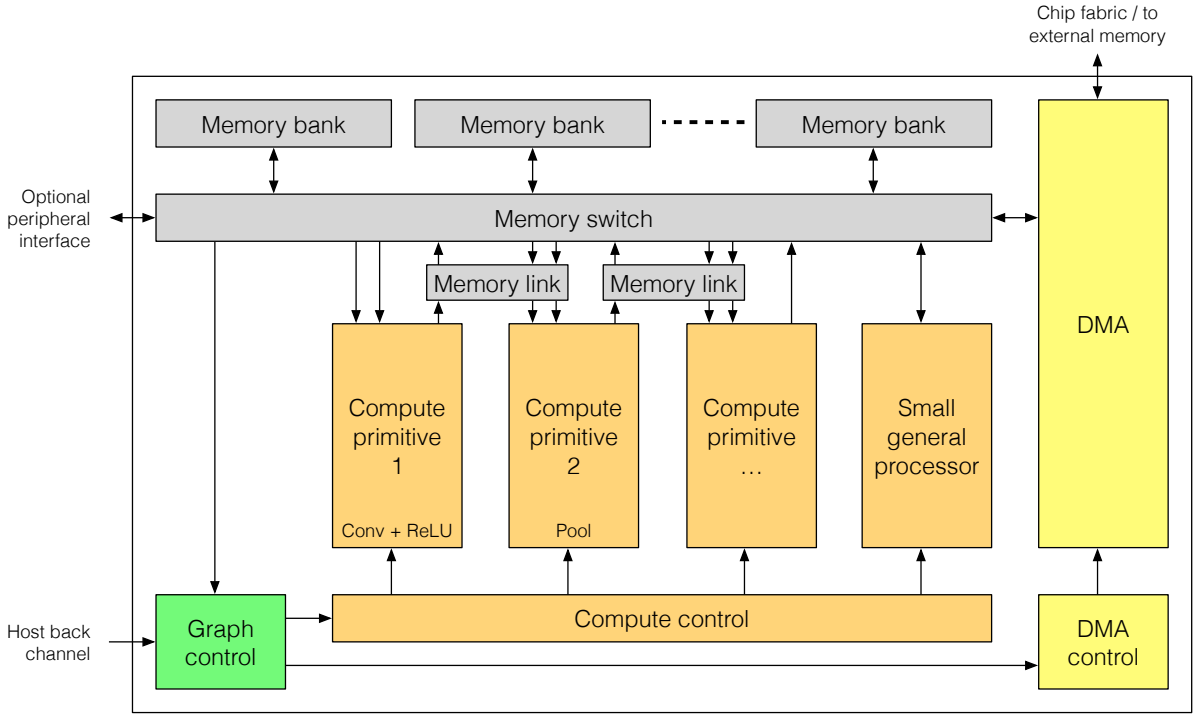
Computational Primitive Defined Domains

- 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
  - Graph control
  - Memory
  - Compute control
  - Computational primitives
  - DMA control
  - DMA
  - Small general processor



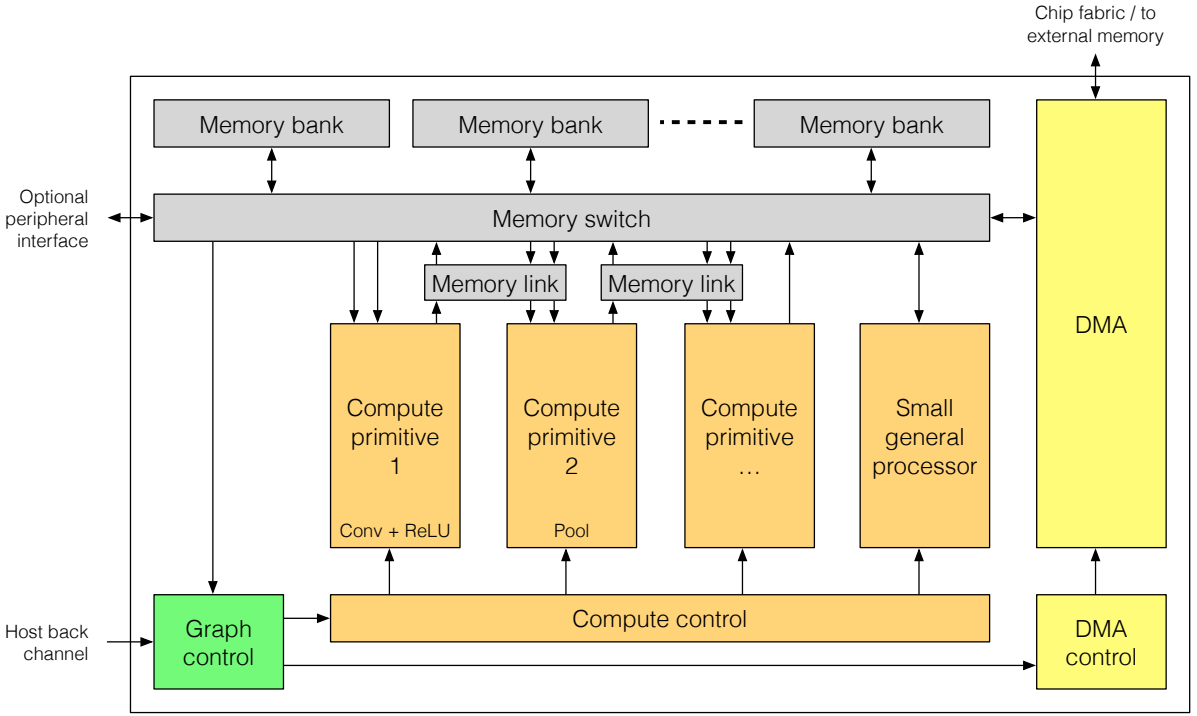
DSA Components

- Graph control
  - Uses node descriptors to sequence node instructions that control compute and DMA operations
- Compute and DMA control
  - Pass instructions to compute primitives and DMA
  - Parallel compute and DMA control allows parallel data movement and compute
  - Used in a ping pong fashion for efficiency
- Memory
  - Sized to allow most feature maps to remain on device



DSA Components

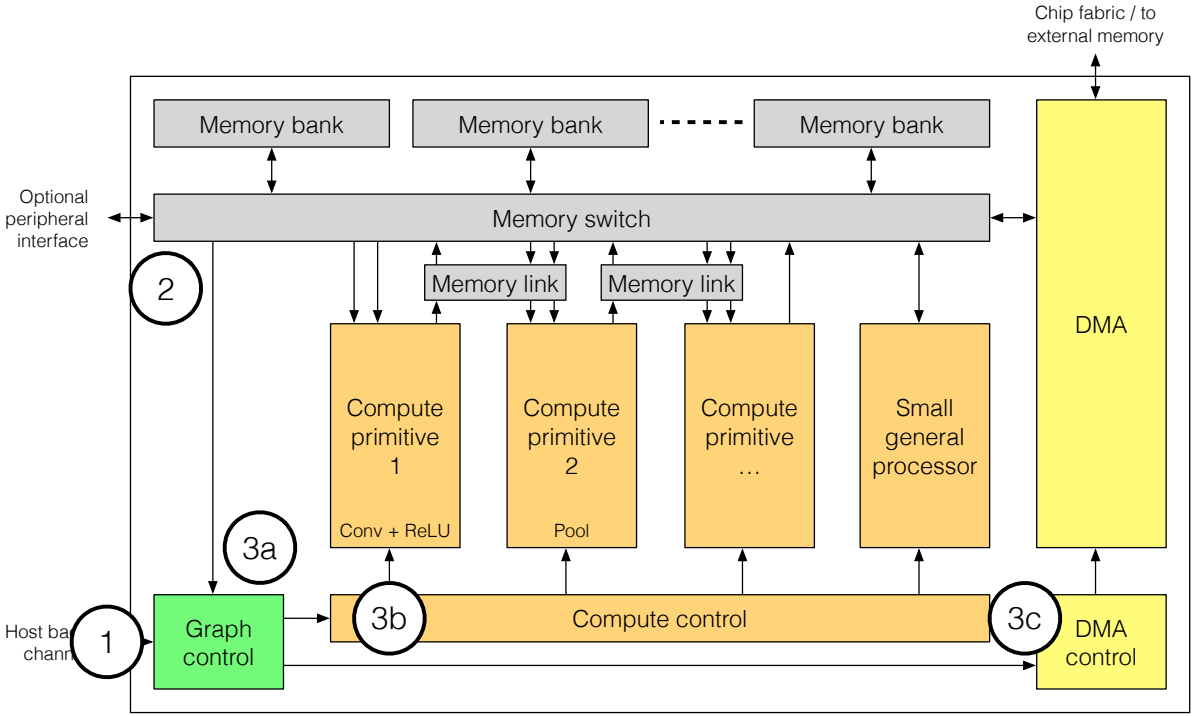
- Compute primitives
  - Use node instructions to setup and execute compute primitives
  - Compute primitives implement low level graph compute operations
  - Compute primitives include matrix, sort, … and a small general processor
- DMA primitive
  - Use node instructions to setup and execute DMA primitive
  - DMA primitive implements low level graph internal / external memory movement operations
  - DMA primitive includes multidimensional transformations, compression, …



Control

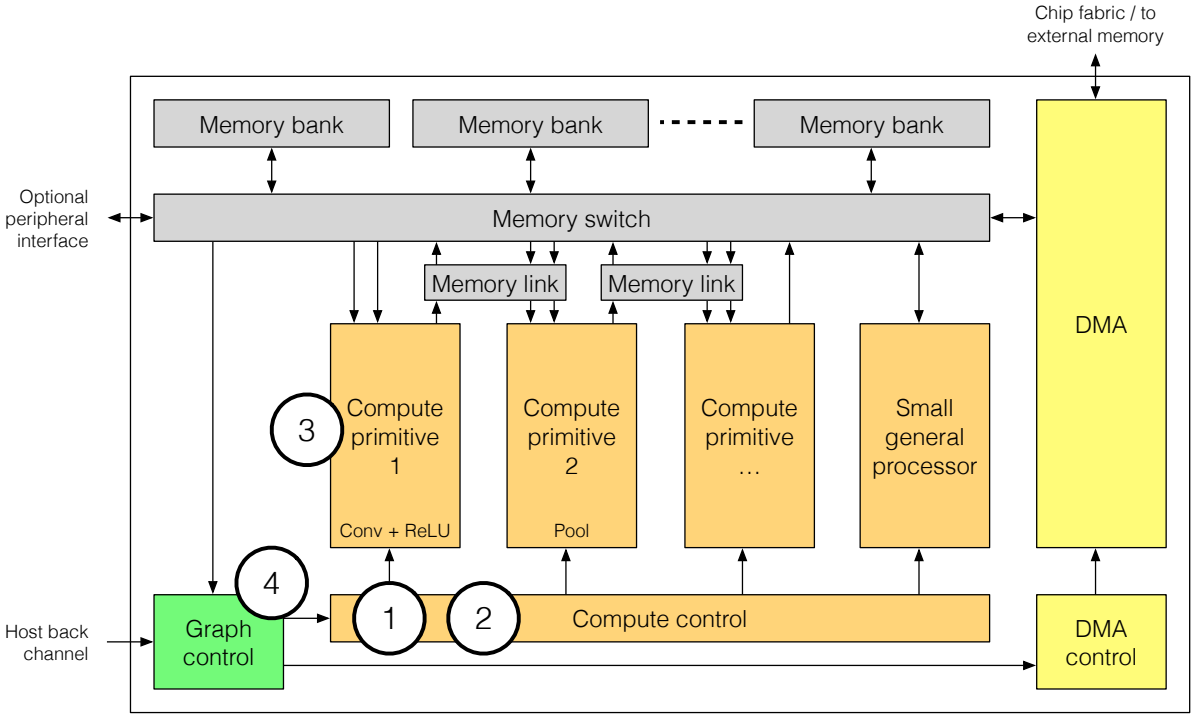
Graph Control

- 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



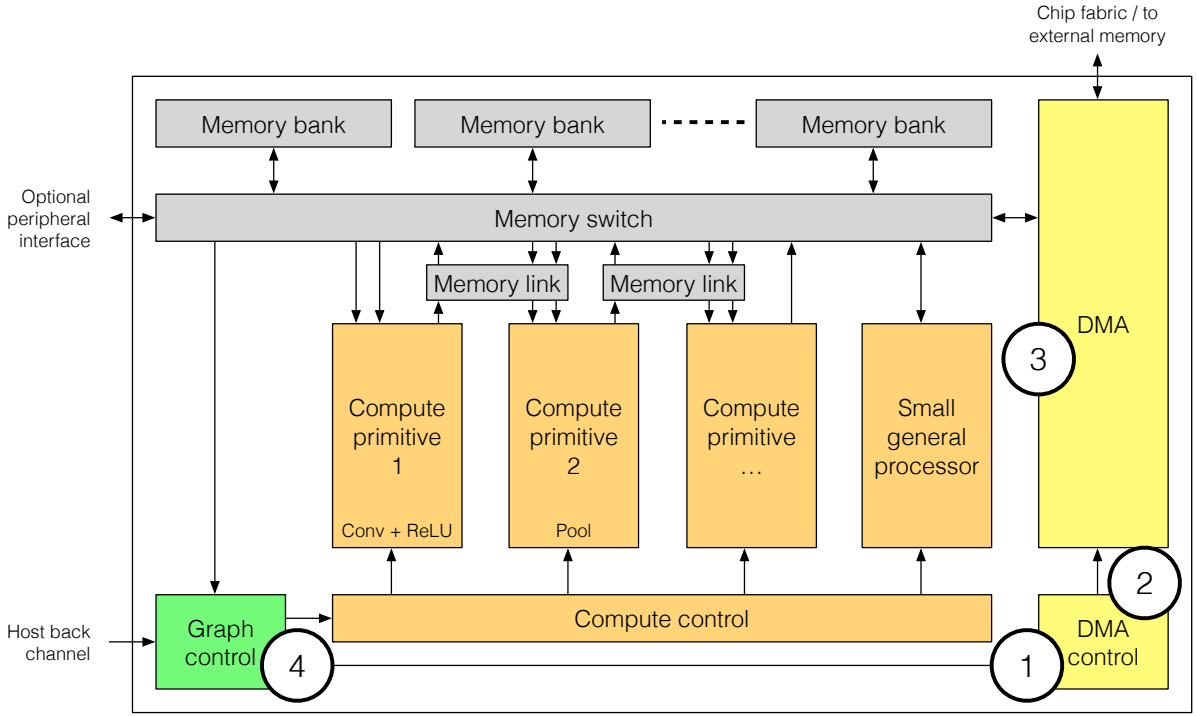
Compute Control

- 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

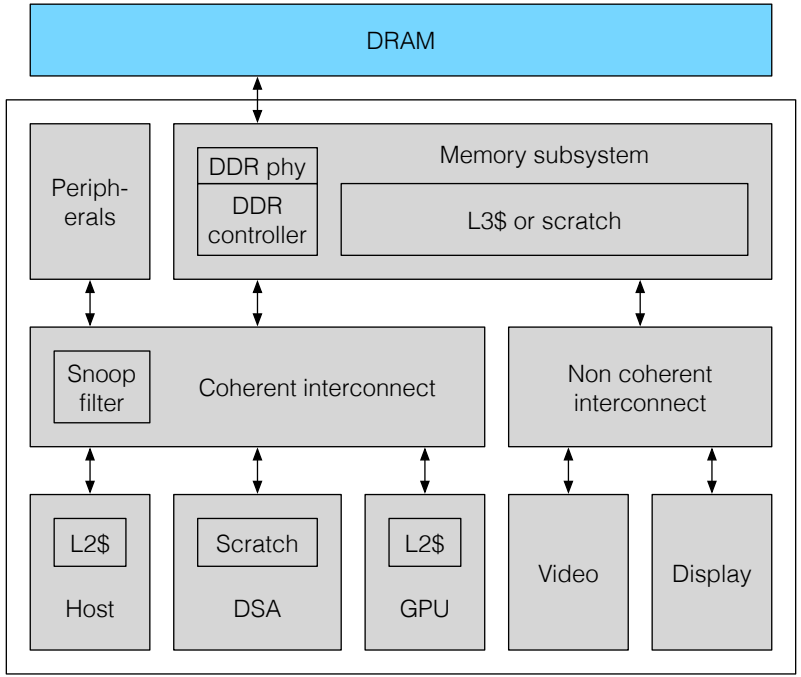
- DMA control does the same things that compute control does, just for the DMA (vs the compute primitives)



Memory

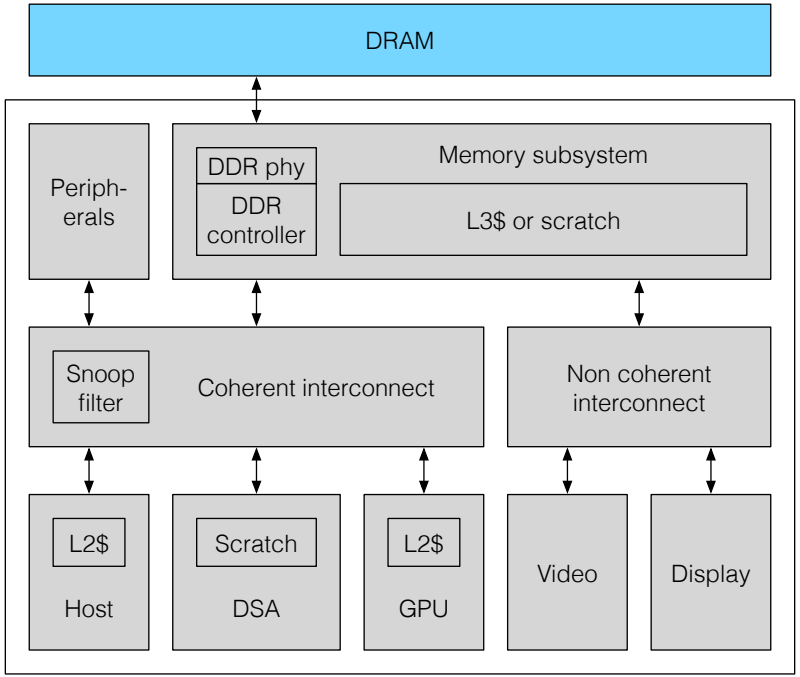
External Memory

- 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



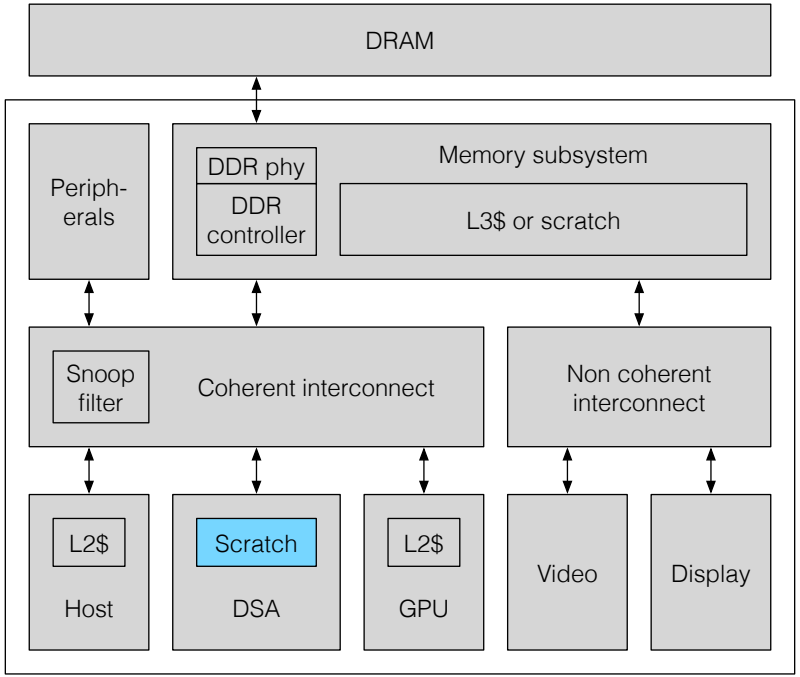
External Memory

- 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



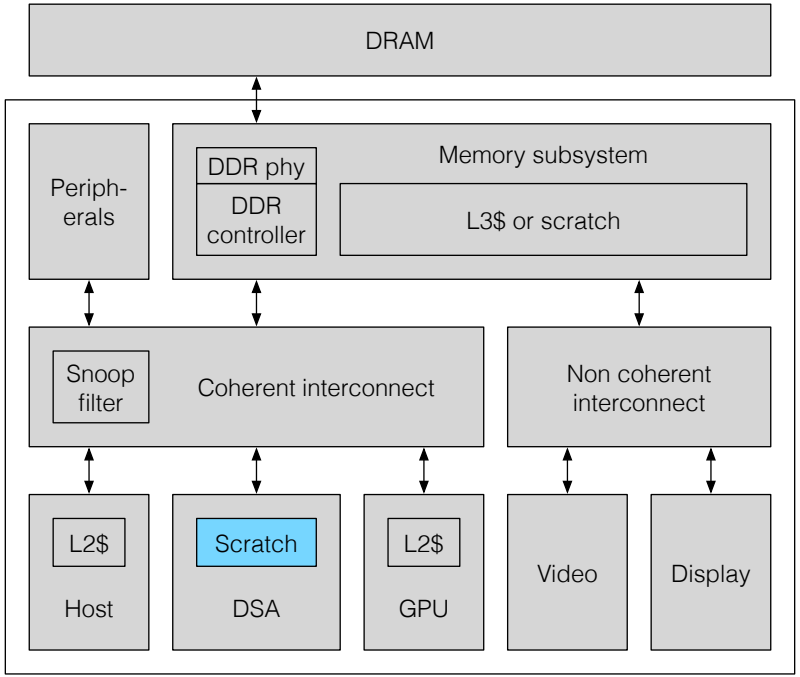
Internal Memory

- 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



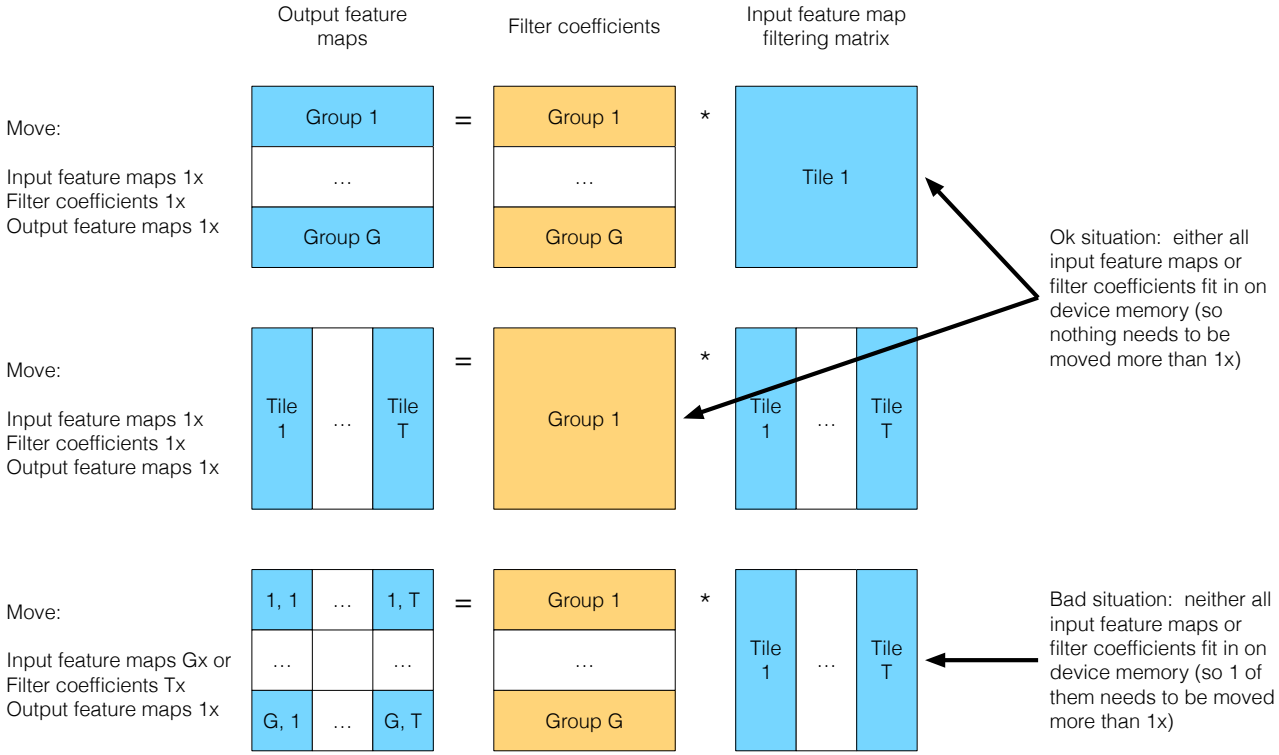
Internal Memory

- 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

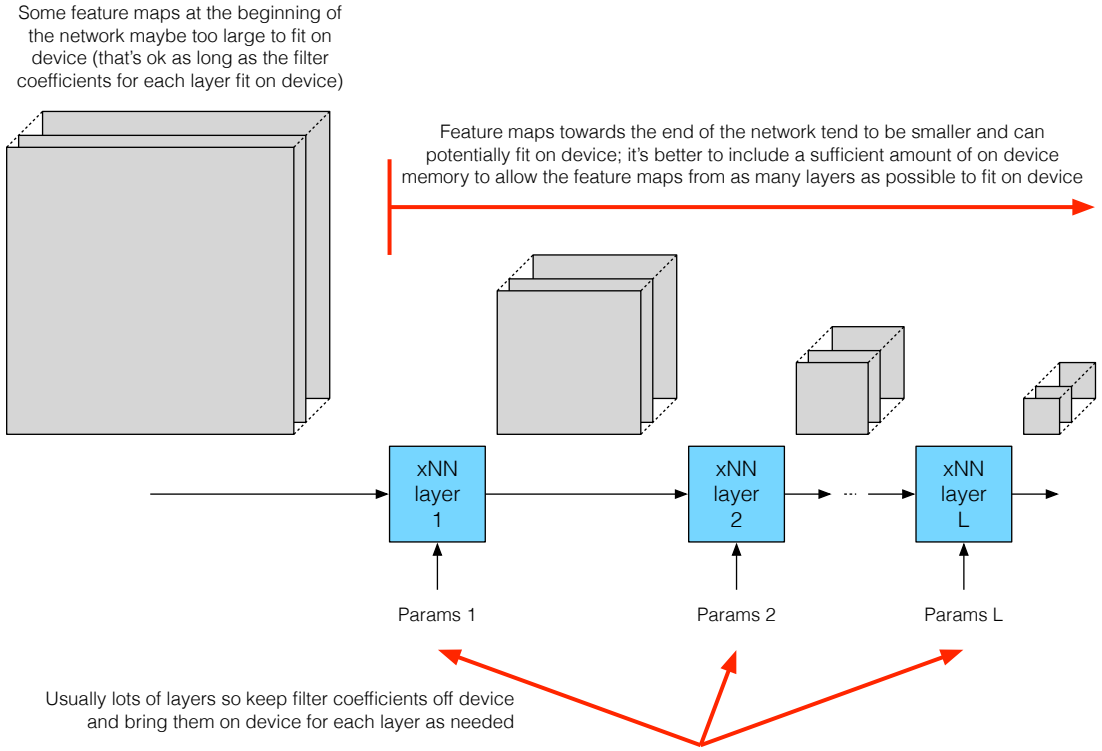


How Much Memory Is Ok

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



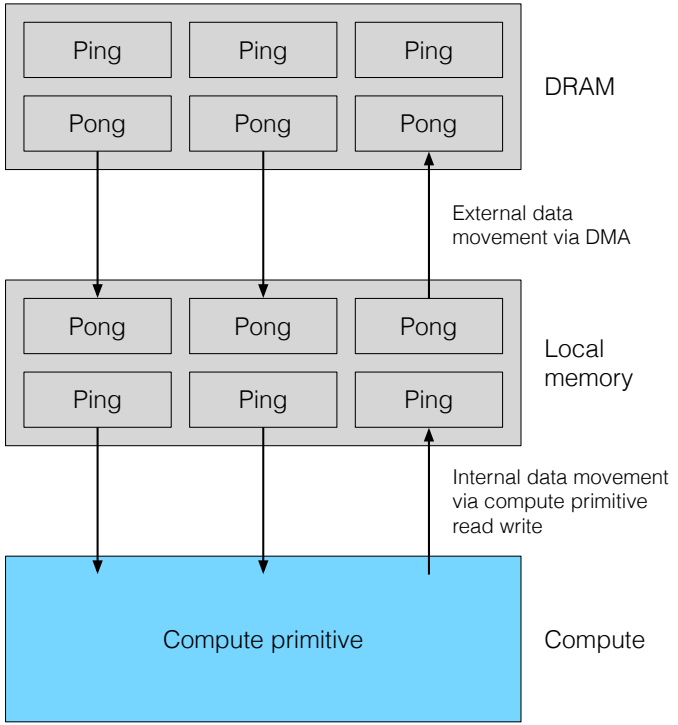
How Much Memory Is Better



External – Internal Data Movement

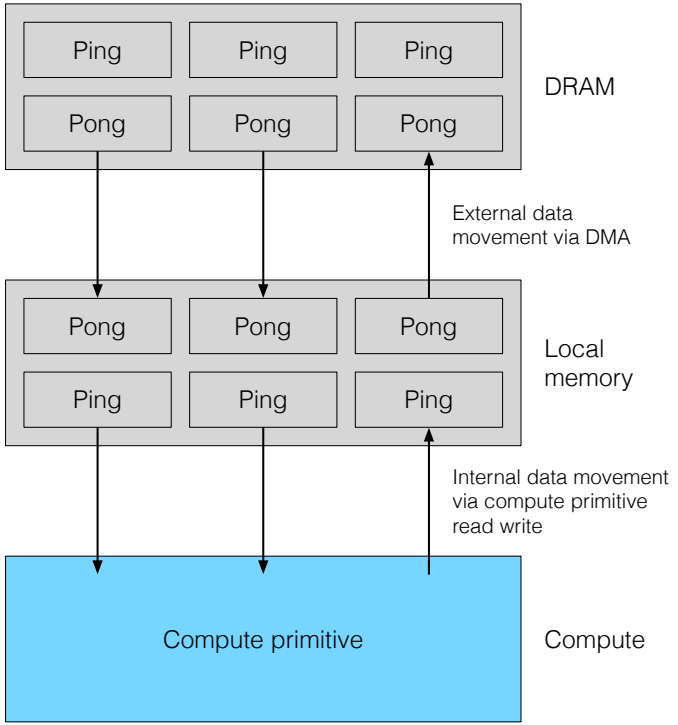
CNN Strategy

- 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



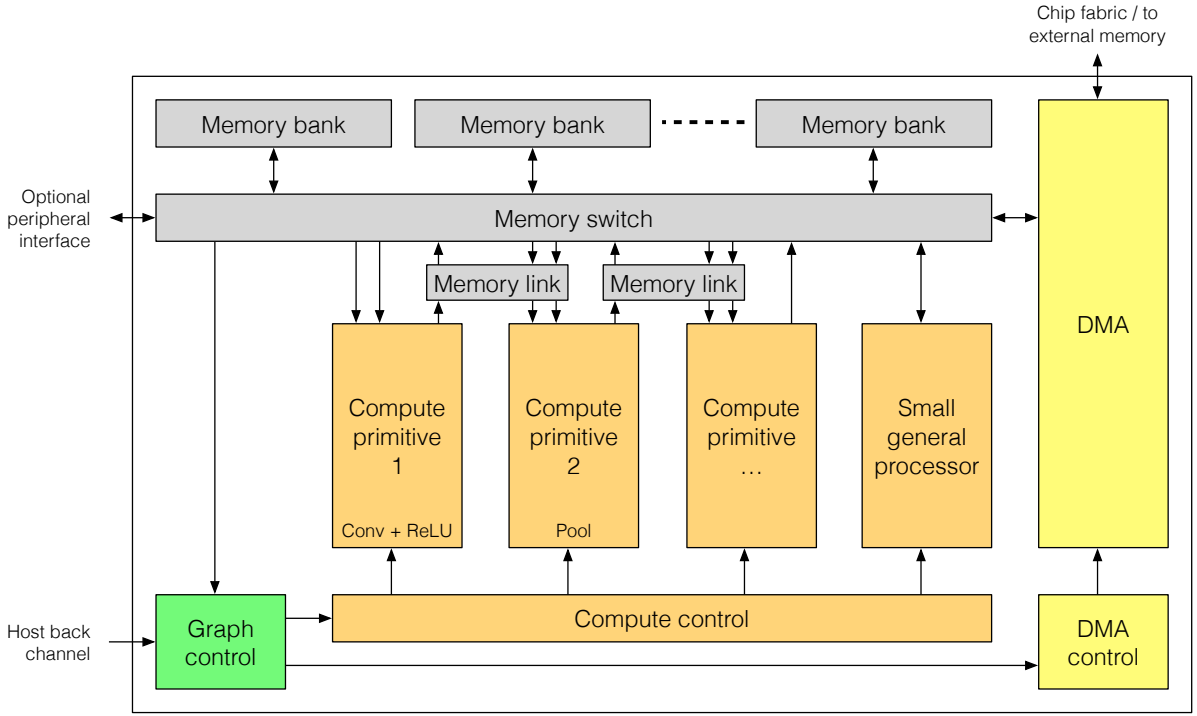
CNN Strategy

- 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



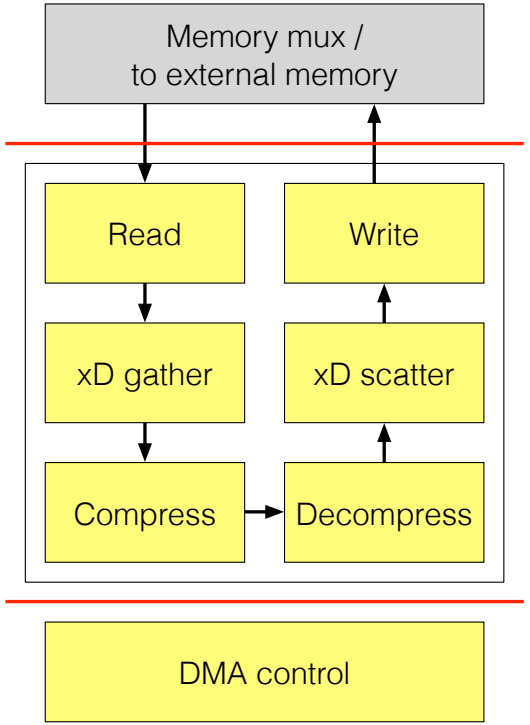
DMA

- 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



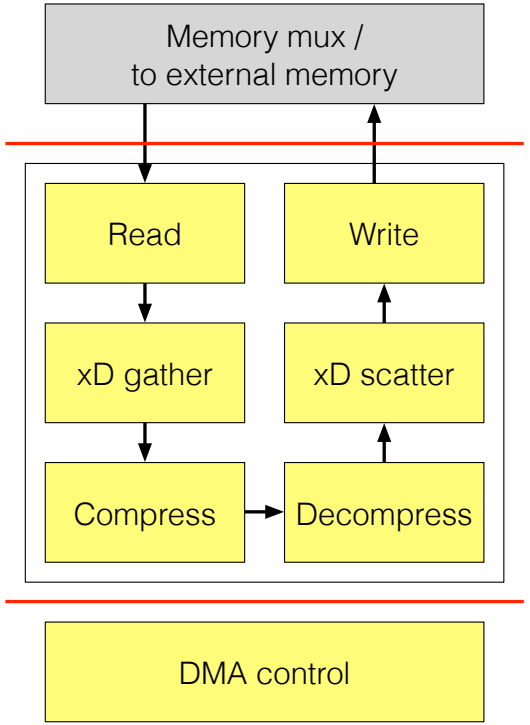
DMA

- 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



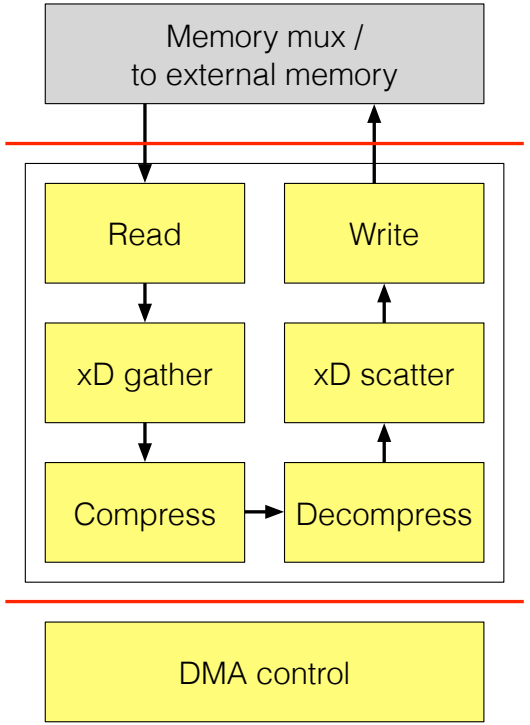
Reduce Data Movement Via Compression

- 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

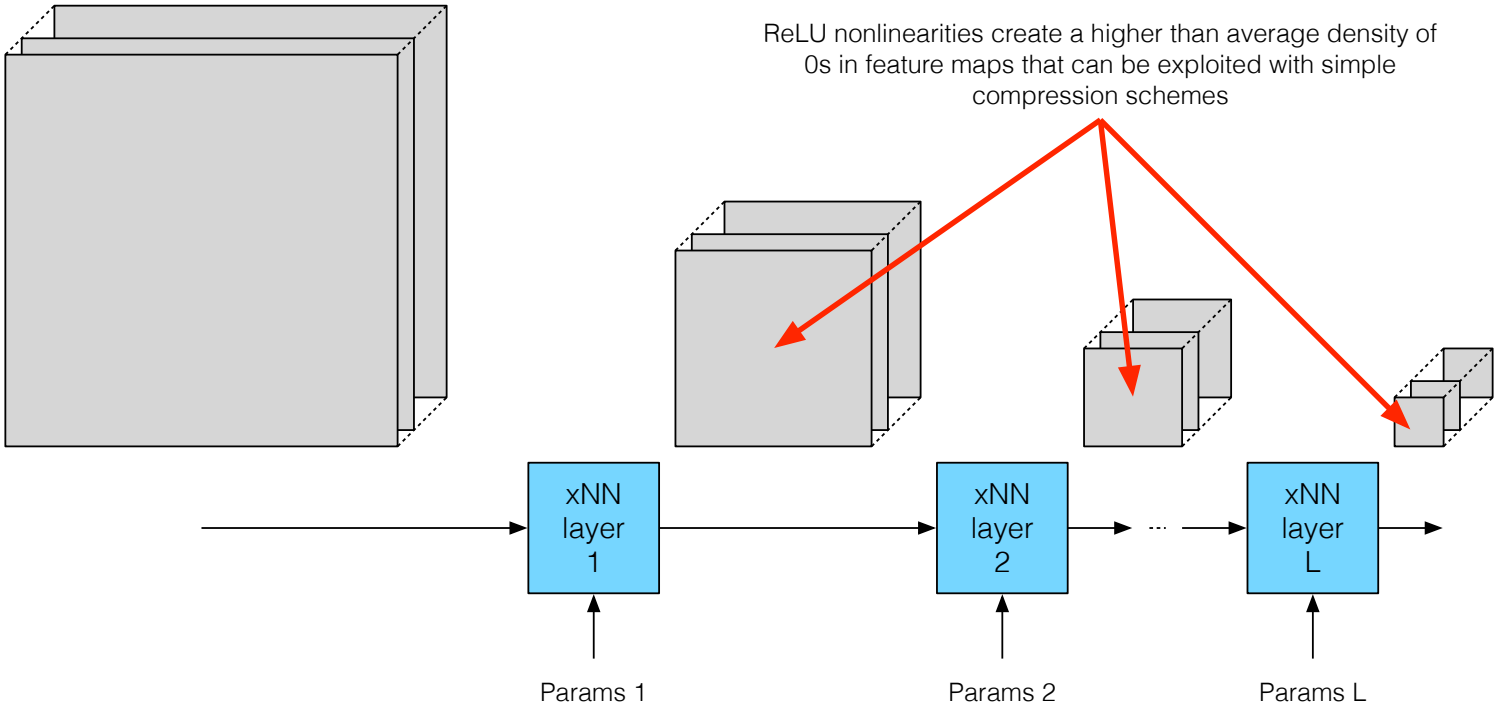


Feature Map Compression

- 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

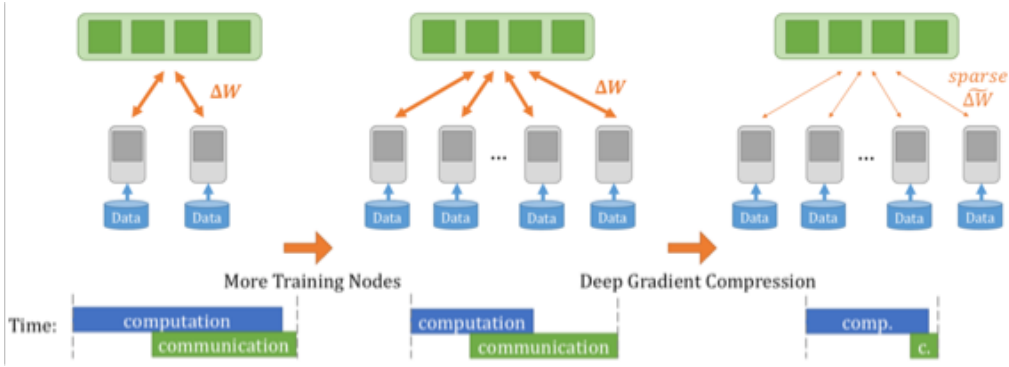


Figure from https://arxiv.org/abs/1712.01887

Compute

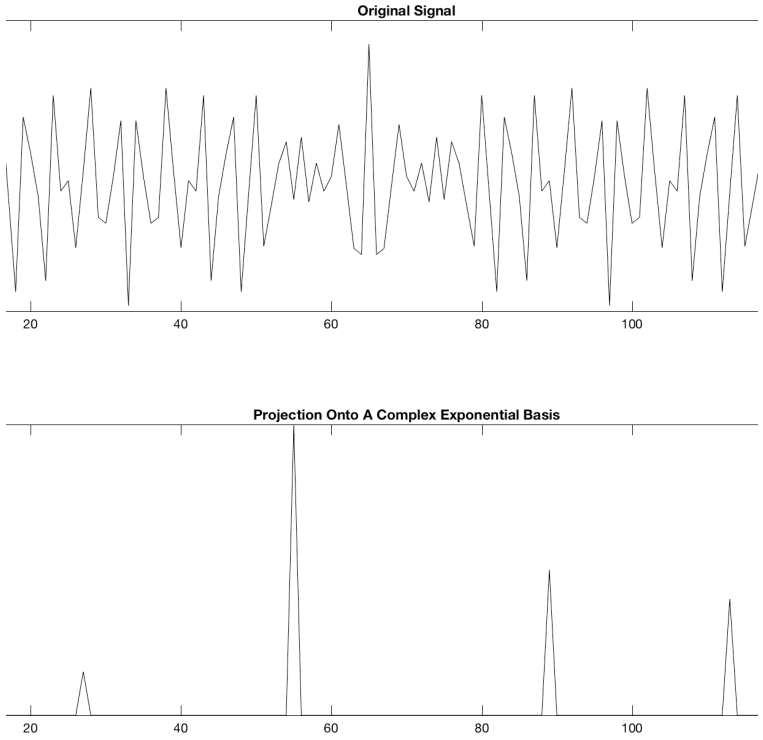
You Don't Put High Level Algorithms In Gates

- 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

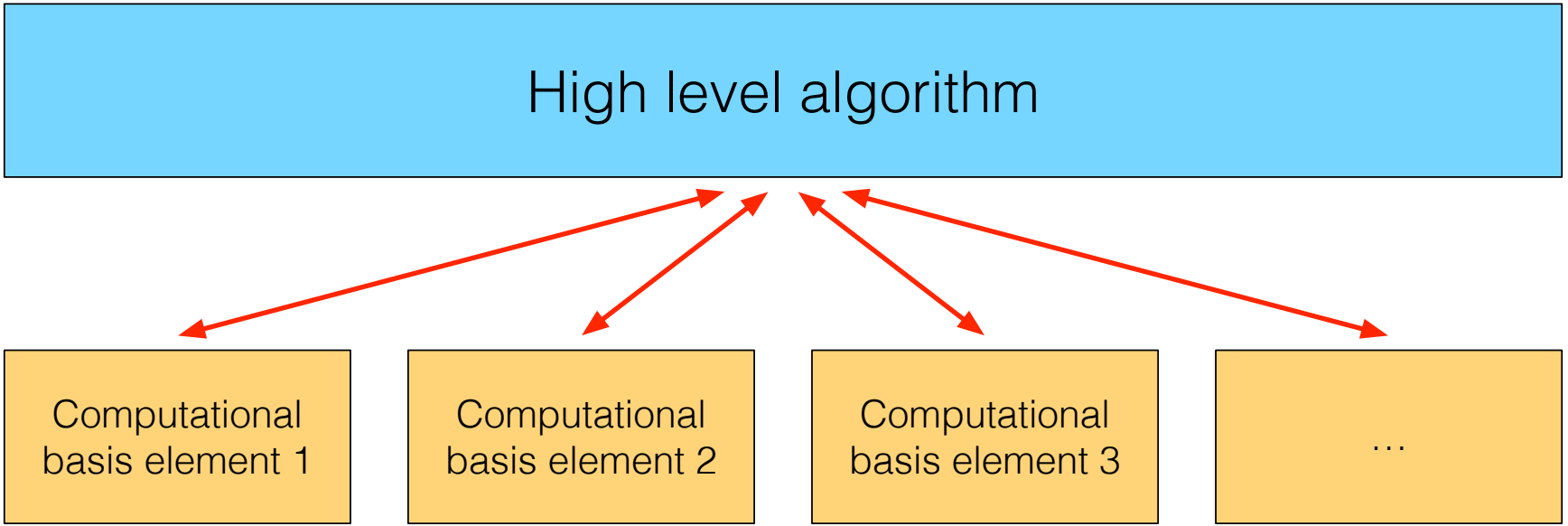
FFT



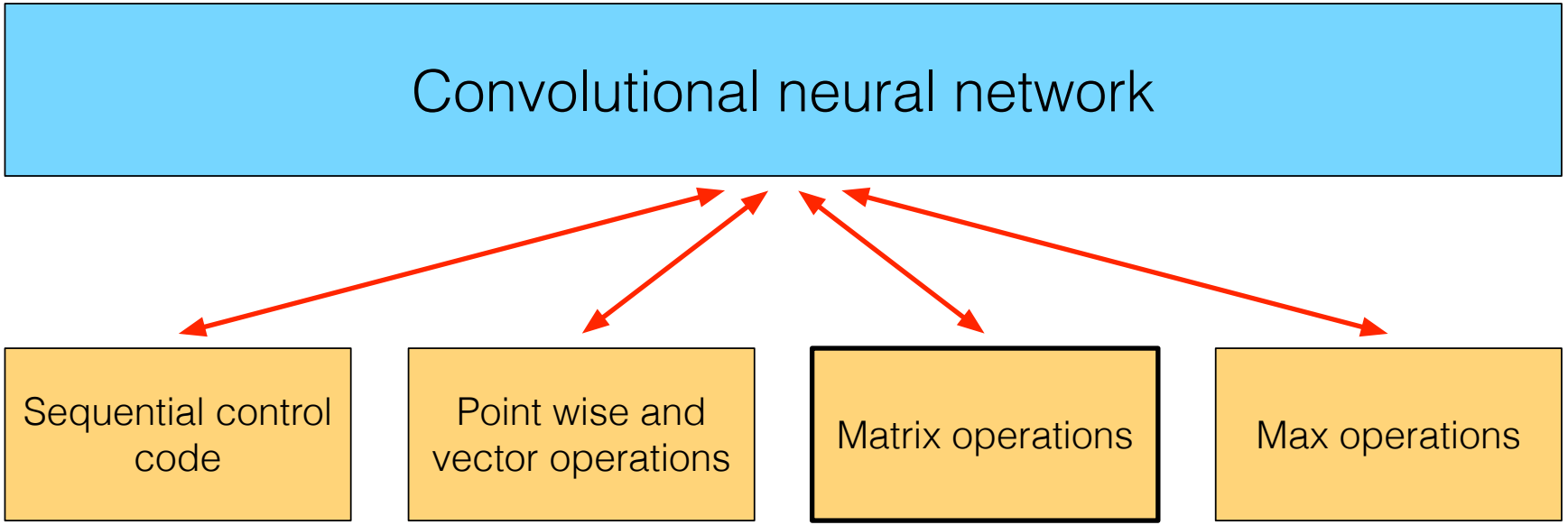
Dear observant class member: yes the top figure is a bit of a cheat (just the abs of the sequence) because I was too lazy when making figures to construct a real even signal; but the point of the slide is valid

A Computational Basis For Algorithms

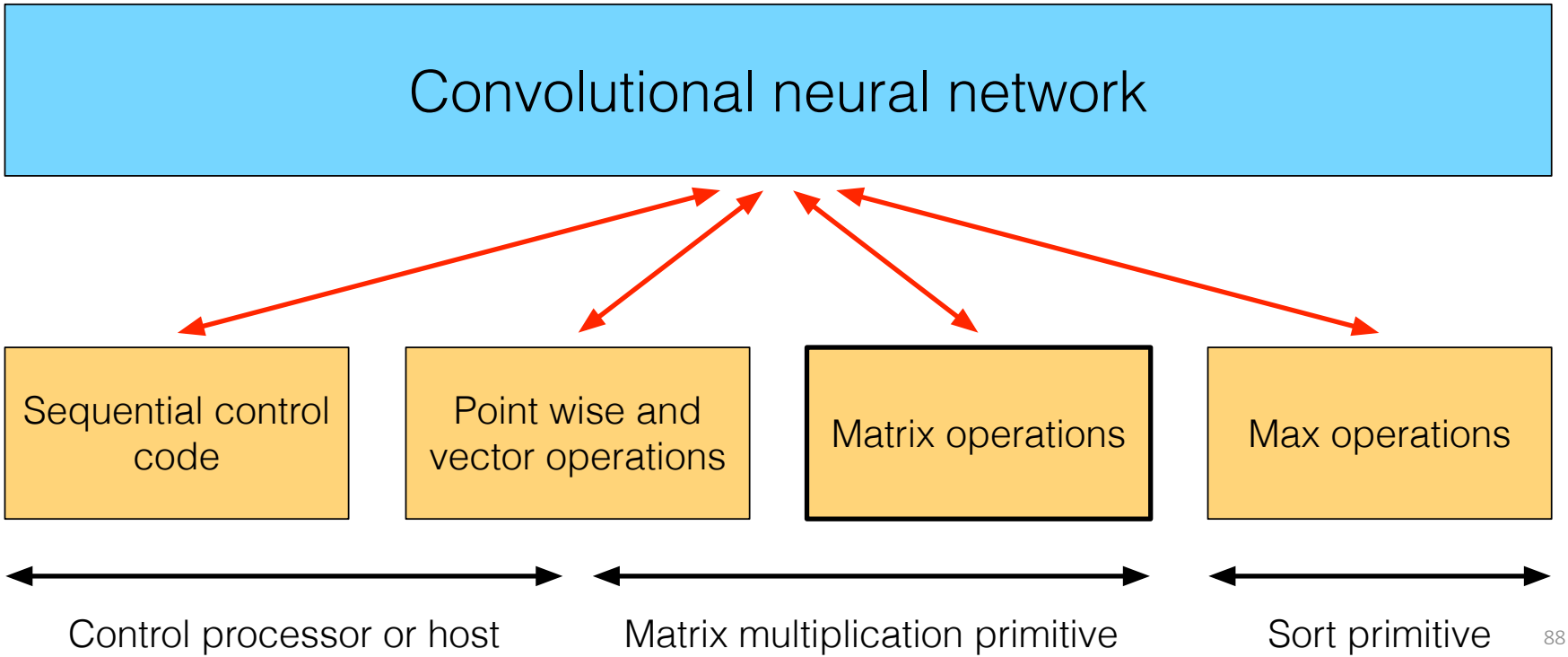
Answer to the question: use the same strategy for designing hardware, decompose high level algorithms onto a computational basis and provide optimized implementations of key computational basis elements (get ASIC efficiency while maintaining algorithmic generality)



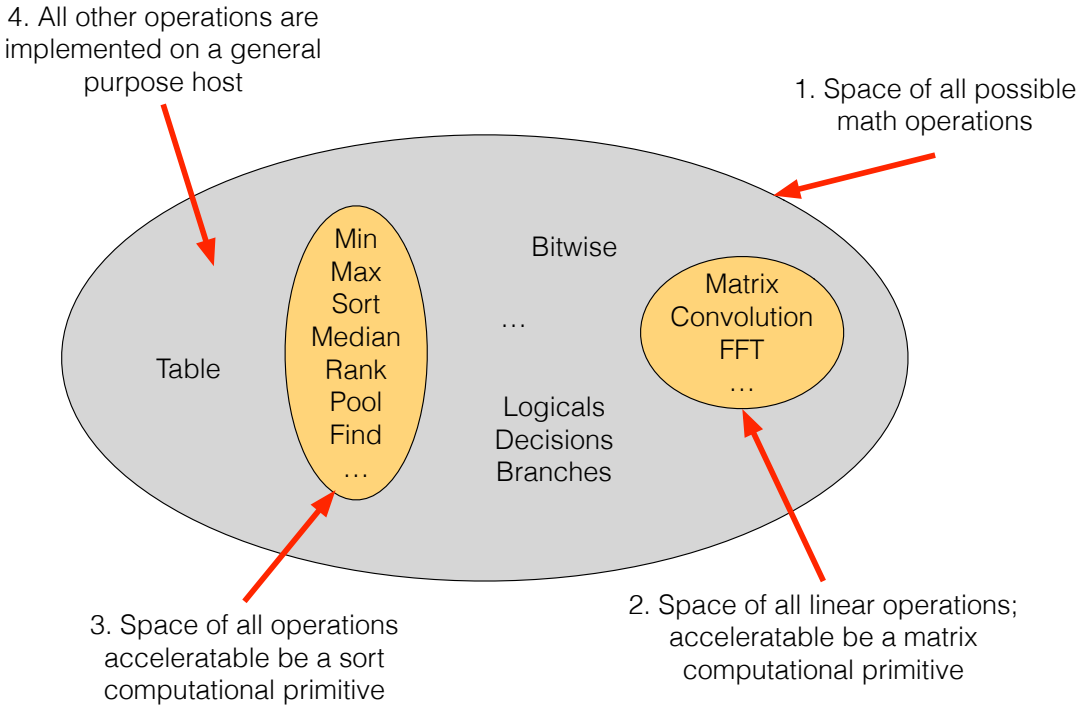
Projecting A xNN Onto A Computational Basis



Projecting A xNN Onto A Computational Basis



An Alternate View Of The Last Few Slides



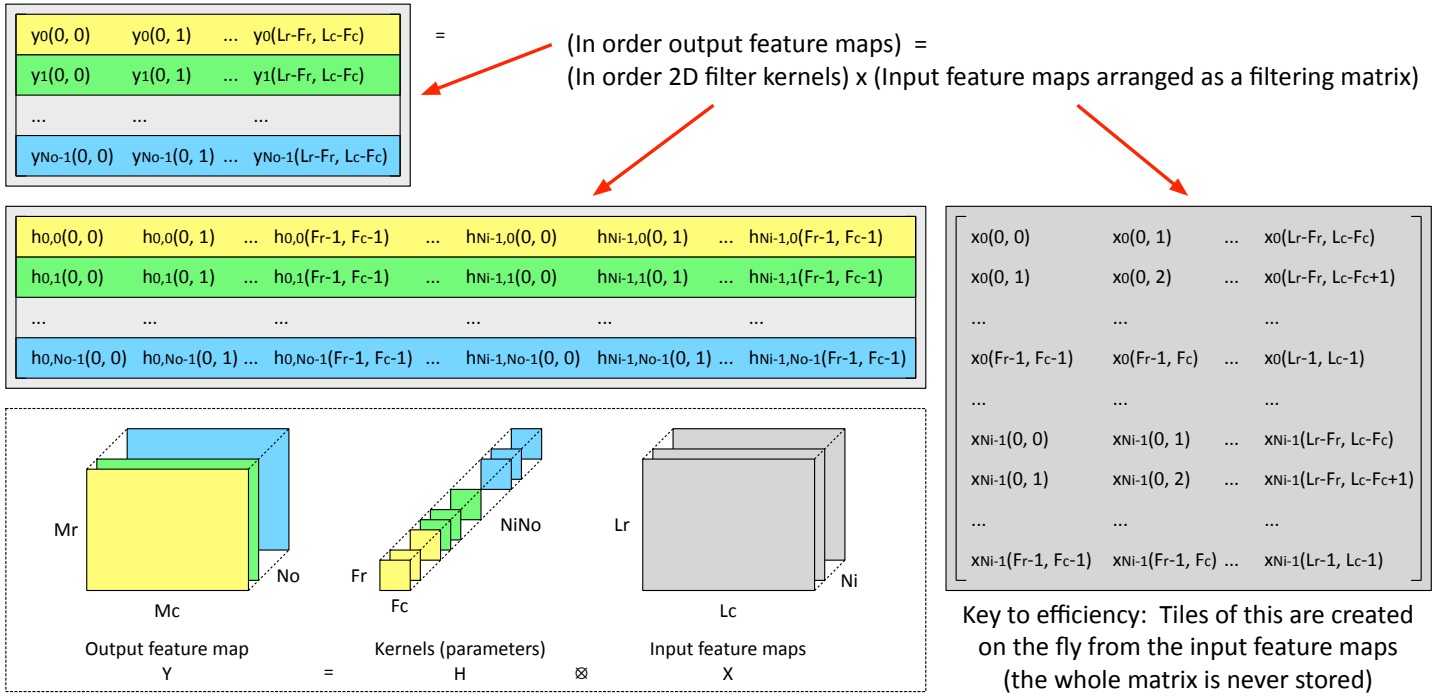
Reminder: Matrix Compute Is Special

- 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

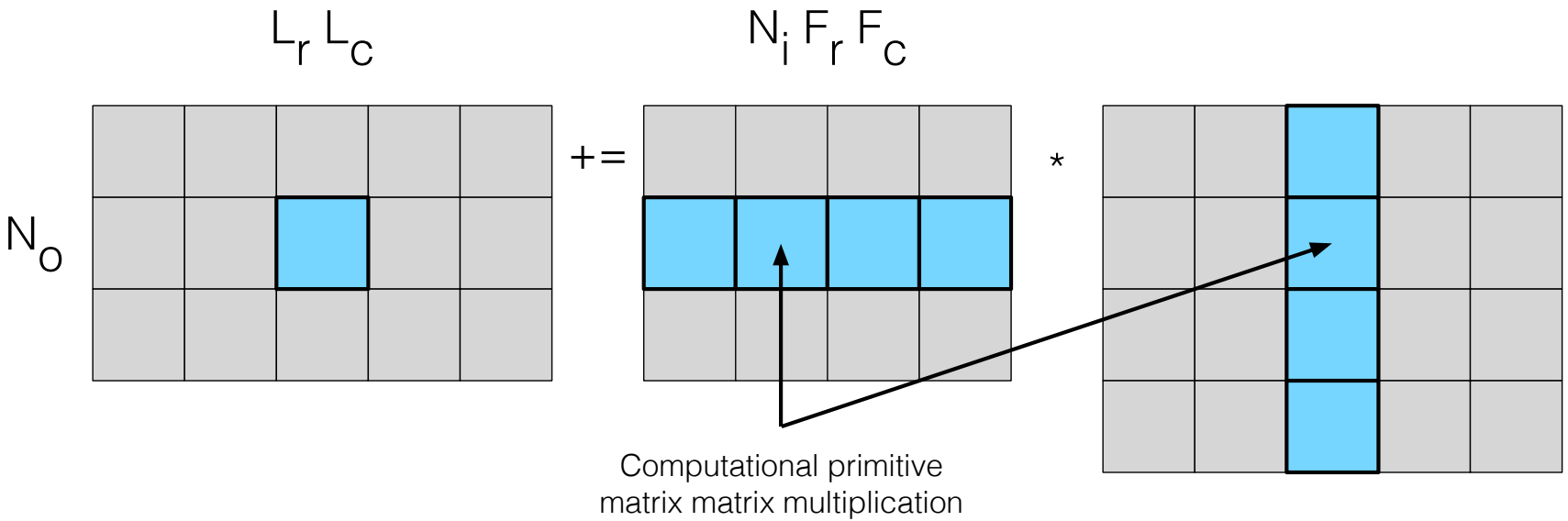
CNN Style 2D Conv Is Large Matrix Mult



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

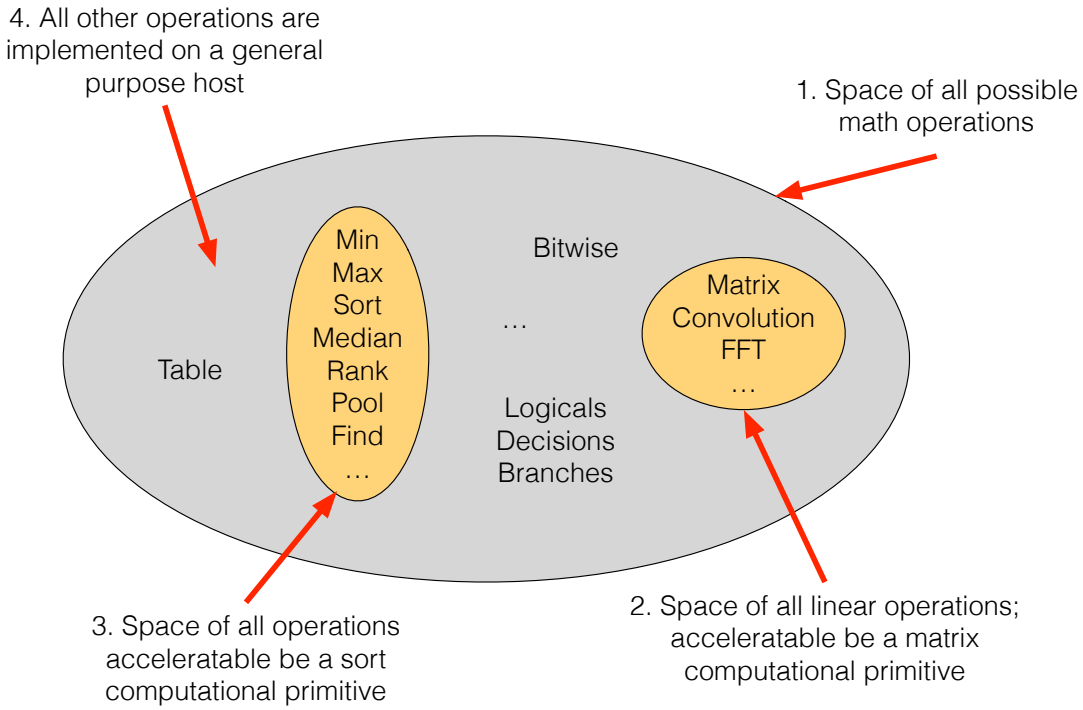
Consider the extremes
- If Fr = Fc = 1 this reduces to matrix matrix multiplication as would be used by a fully connected layer processing multiple regions or batches
- If Fr = Lr, Fc = Lc this reduces to matrix vector multiplication; not possible for any method to accelerate well because bandwidth limited

Large Matrix Mult Via Tiled Matrix Mult



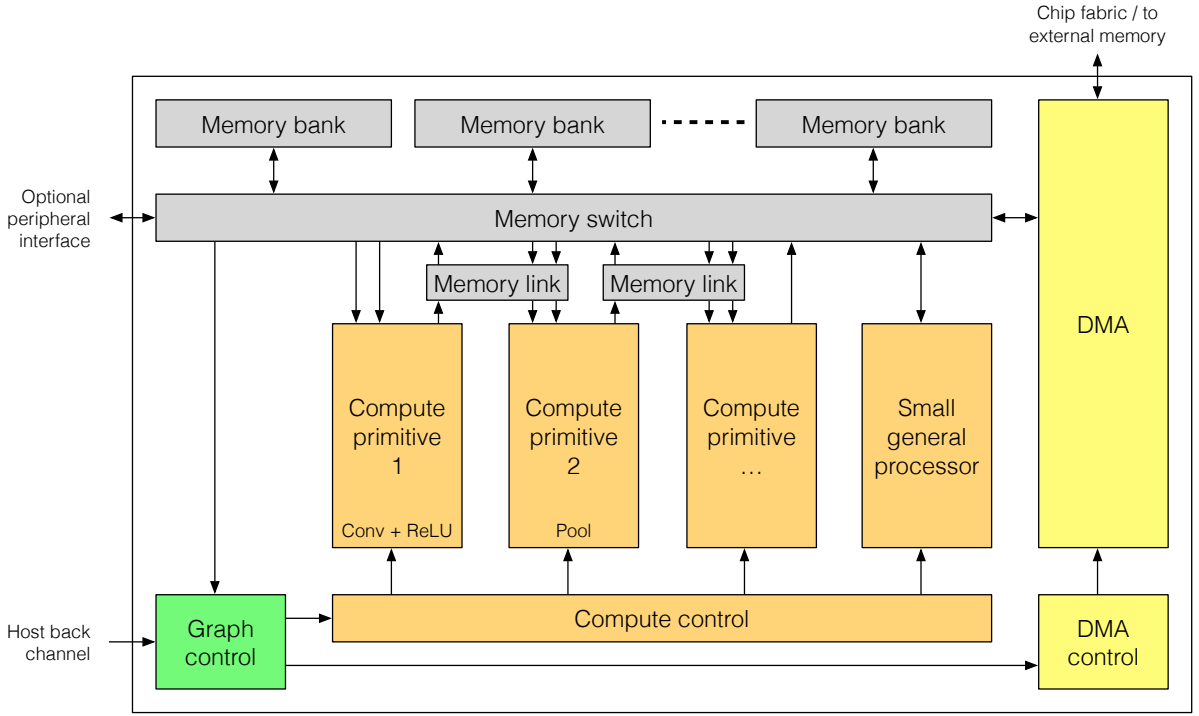
Computational Primitive Candidates

- 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
  - …



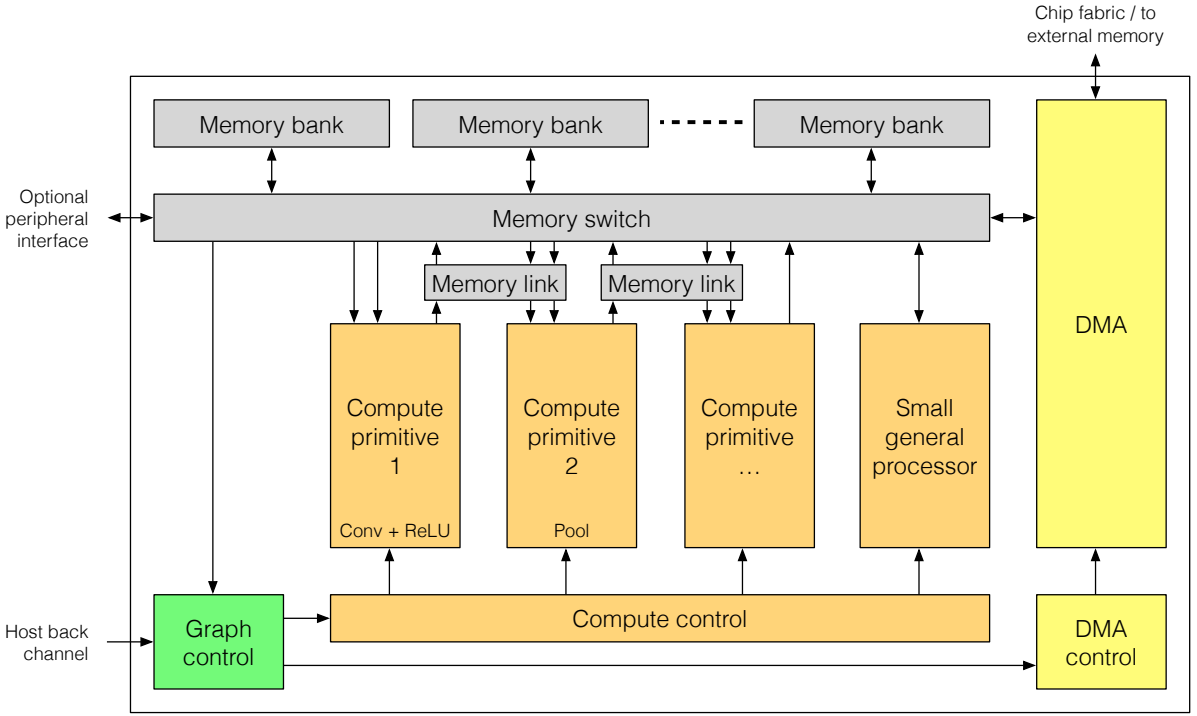
Serial Or Parallel Execution Of Compute

- 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

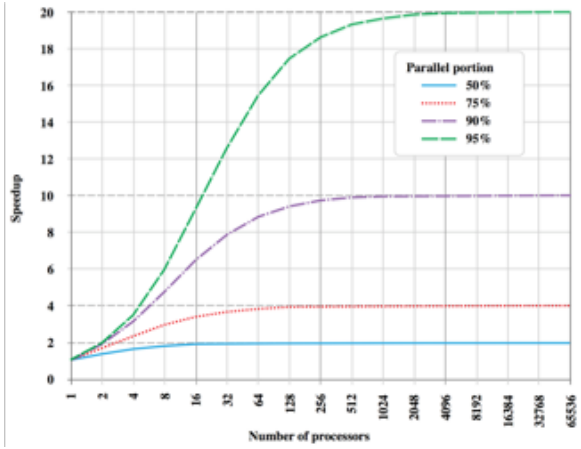
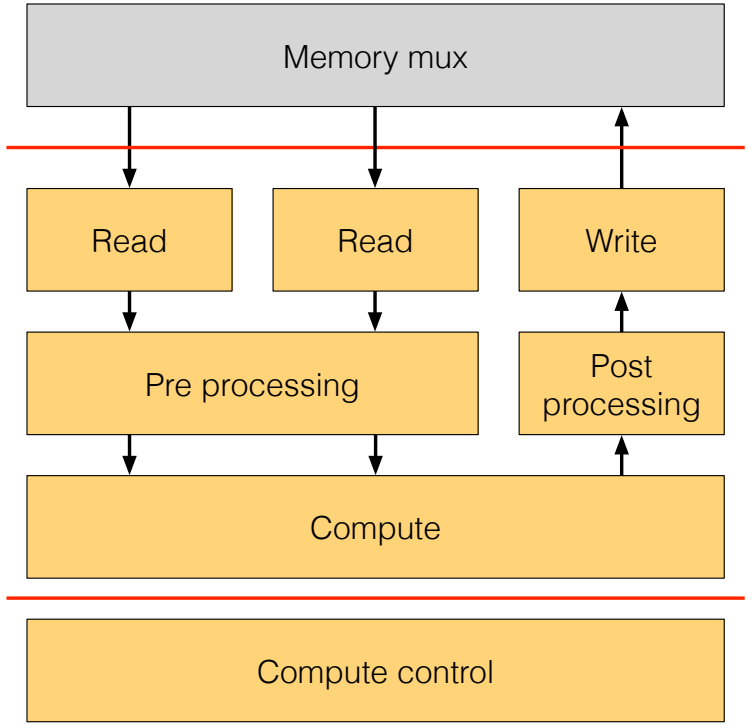


Figure from https://en.wikipedia.org/wiki/Amdahl%27s_law

Computational Primitive Template

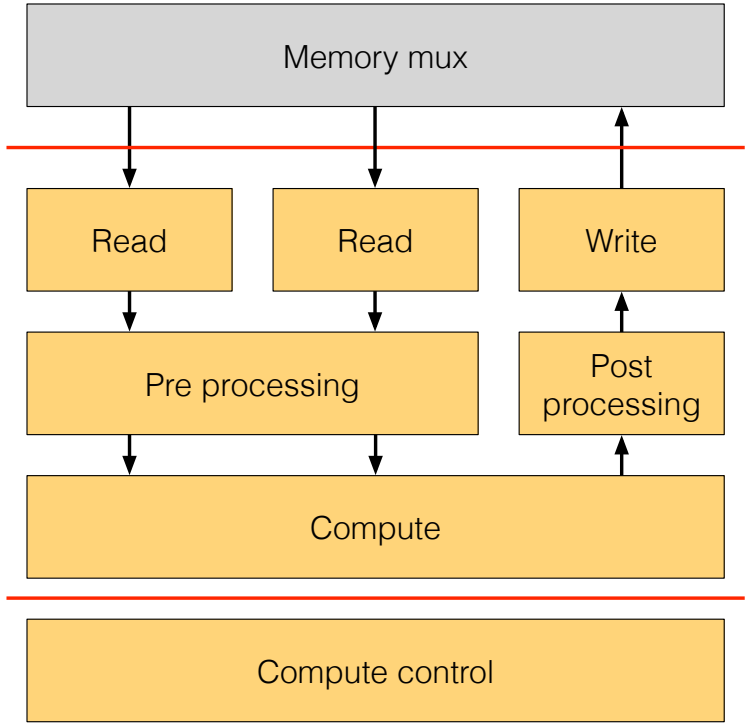
- Data flow
  - Read from local memory
    - Optimal alignment and length
  - Pre processing
    - Simple limited set of data transformations
    - Allows optimized regular compute structure
    - Enable generality within the primitive class
  - Compute
    - Computational primitive computation
  - Post 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

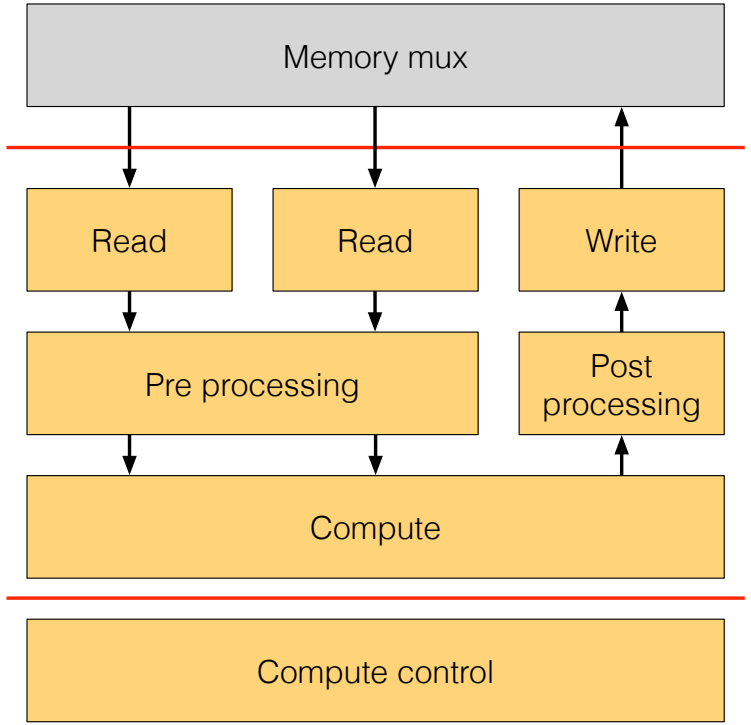
- Basics
  - Read two N x N matrices in N cycles
    - Ex: $\mathbf{H}_{tile}(m, k)$ and $\mathbf{X}_{tile}(k, n)$
  - Use pre processing to do formatting for the input
    - Ex: $\mathbf{H}_{tile}(m, k) \rightarrow \mathbf{A}$ and $\mathbf{X}_{tile}(k, n) \rightarrow \mathbf{X}_{filter}(k, n) \rightarrow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

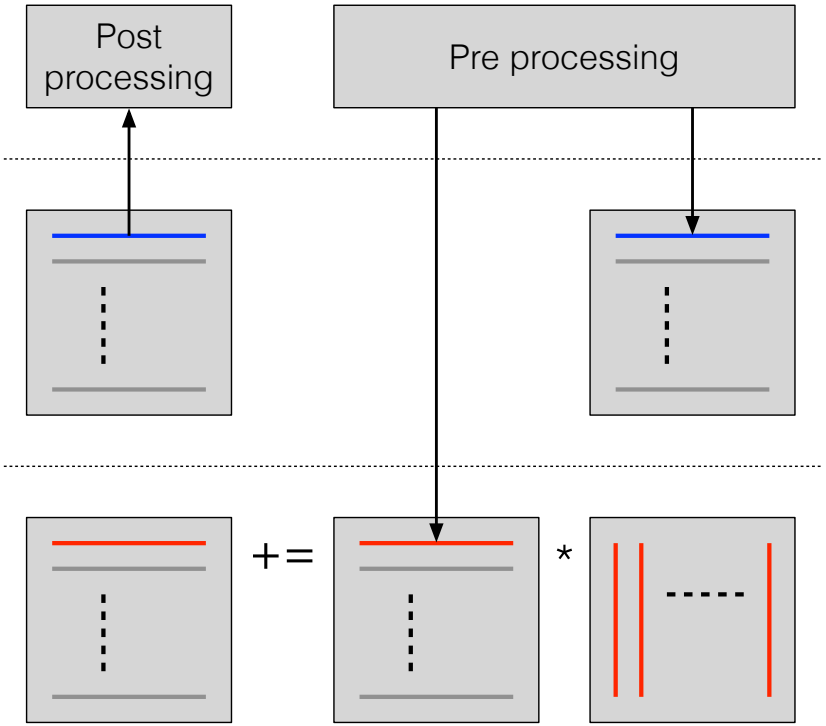


Inner Product Based Matrix Multiplication

- 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

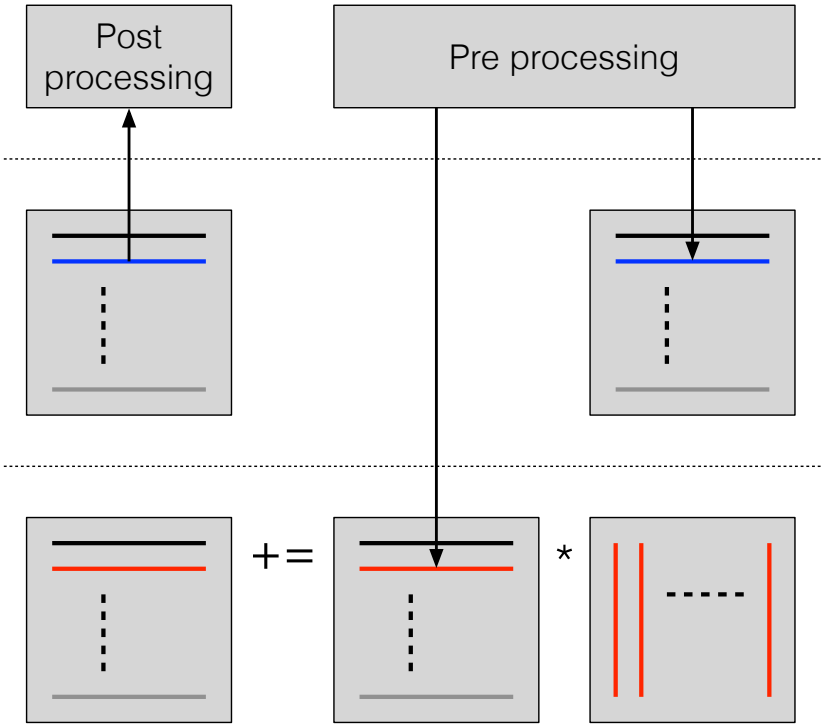


Inner Product Based Matrix Multiplication

- 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

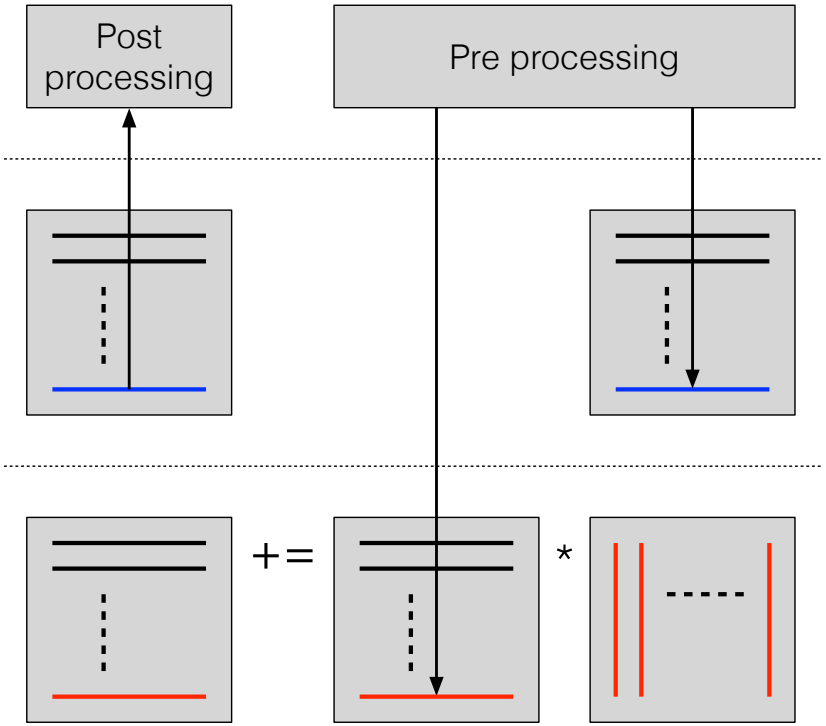


Inner Product Based Matrix Multiplication

- 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

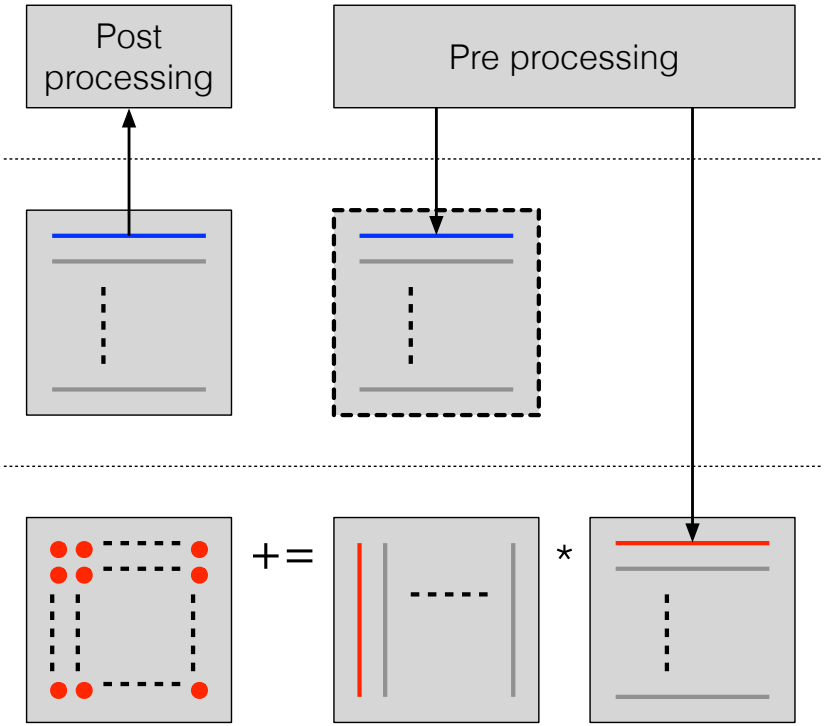


Outer Product Based Matrix Multiplication

- 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

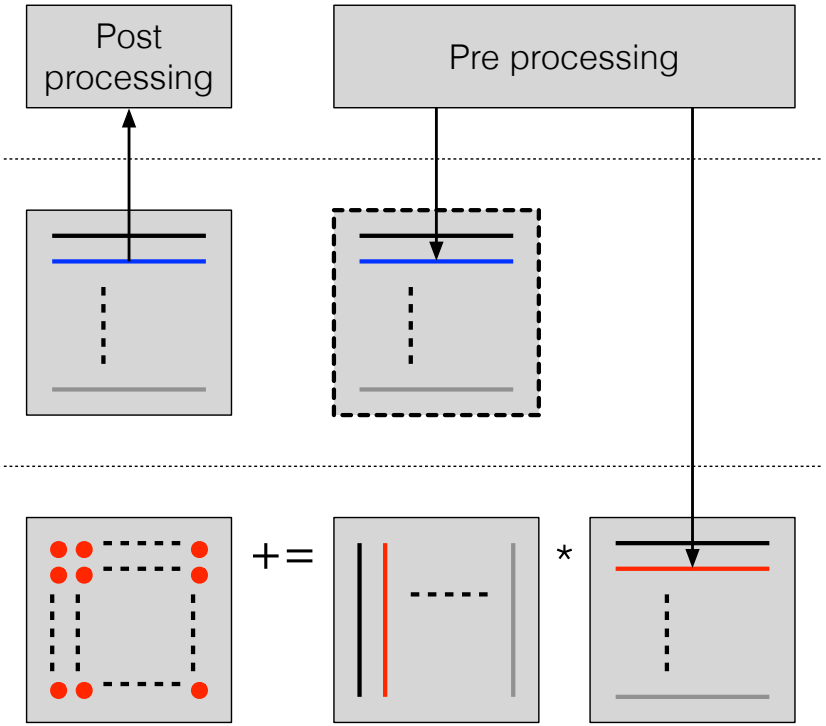


Outer Product Based Matrix Multiplication

- 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

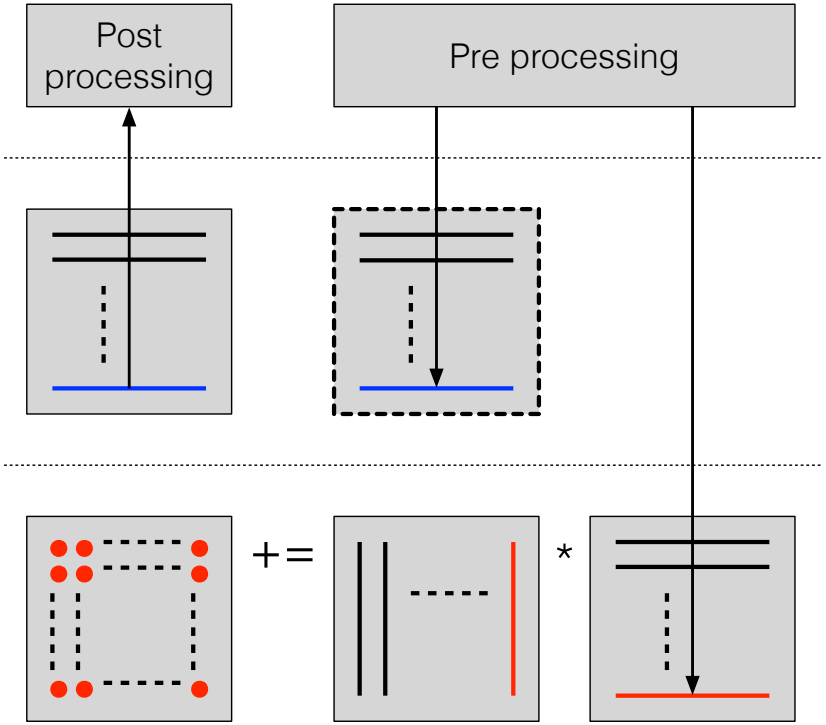


Outer Product Based Matrix Multiplication

- 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(:, :)

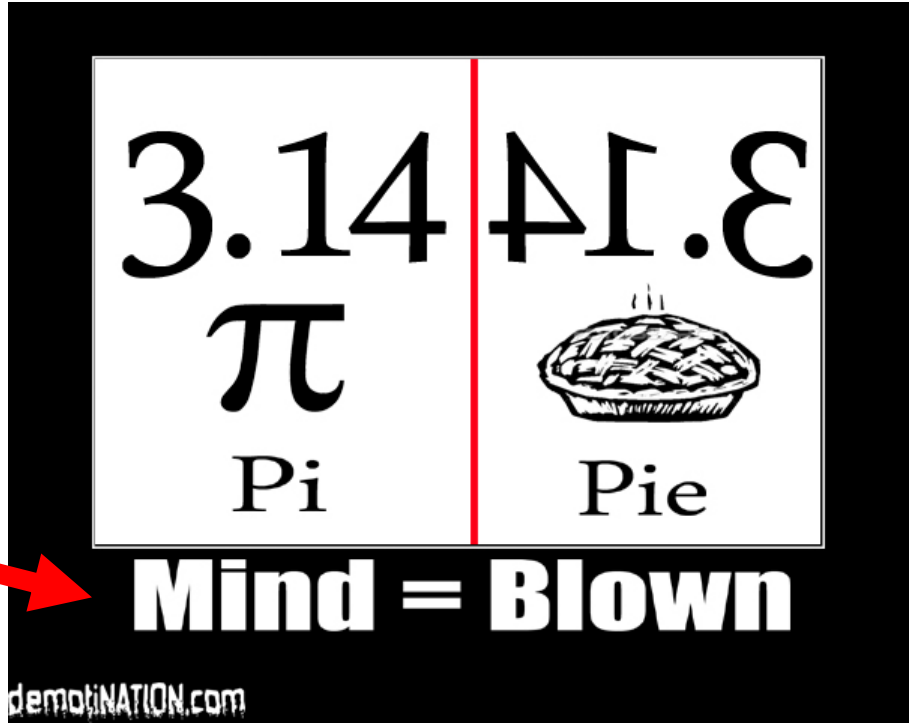
```
C = C0                    // e.g., bias matrix
For k = 0 to K - 1        // k = K - 1
   For m = 0 to M - 1
      For n = 0 to N - 1
         C(m, n) += A(m, k) B(k, n)
      End
   End
End
```

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

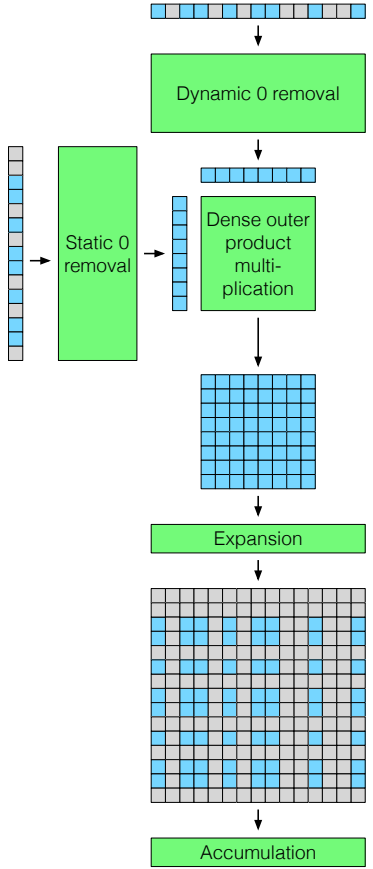
- Related side note
- Similar to fast algorithms for multiplying complex numbers, FFTs, matrix multiplication, ... Winograd figured out a fast algorithm for convolution
- A modified variant has been applied to CNNs, is used in some libraries and is appropriate for some architectures
- Fast algorithms 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

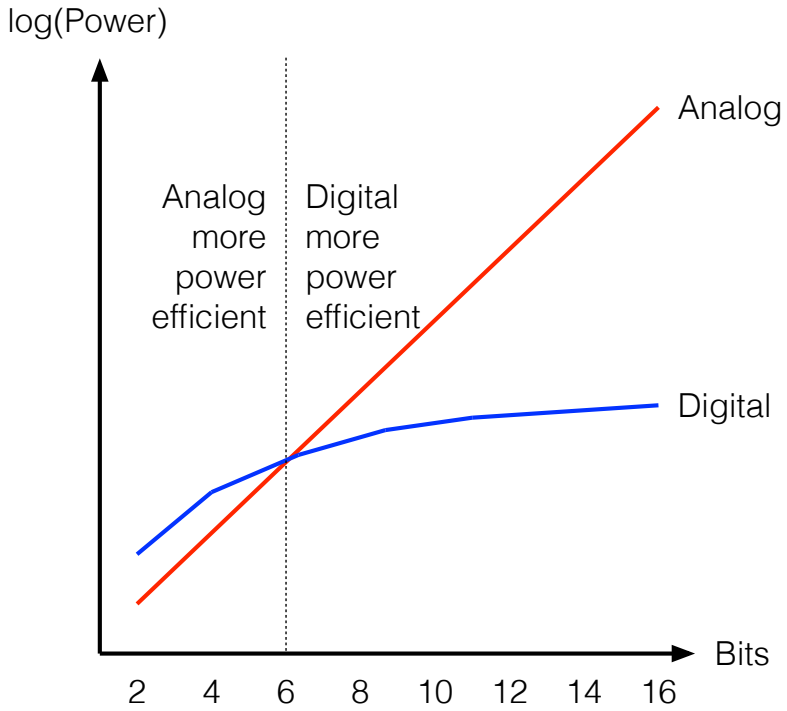


Analog Matrix Multiplication

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

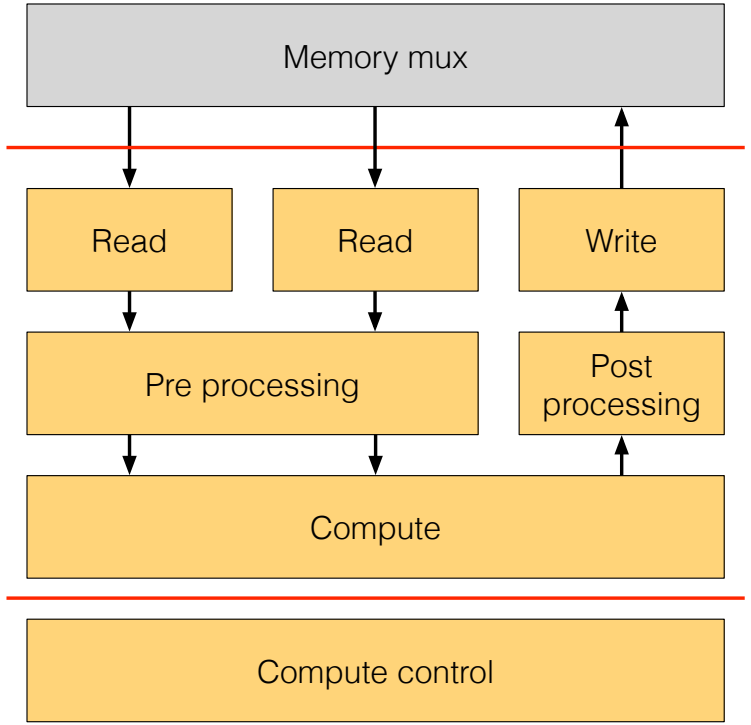
Matrix Multiplication Primitive Uses

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)

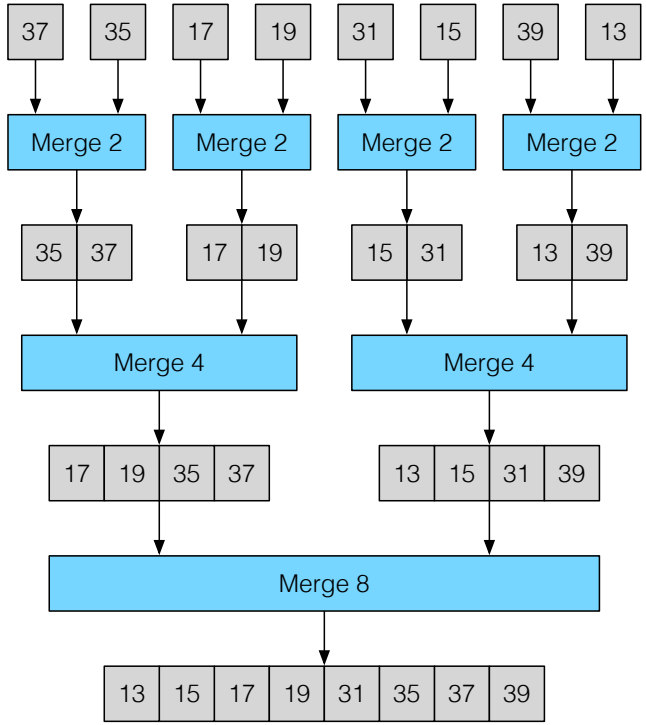
Sort Primitive

- 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

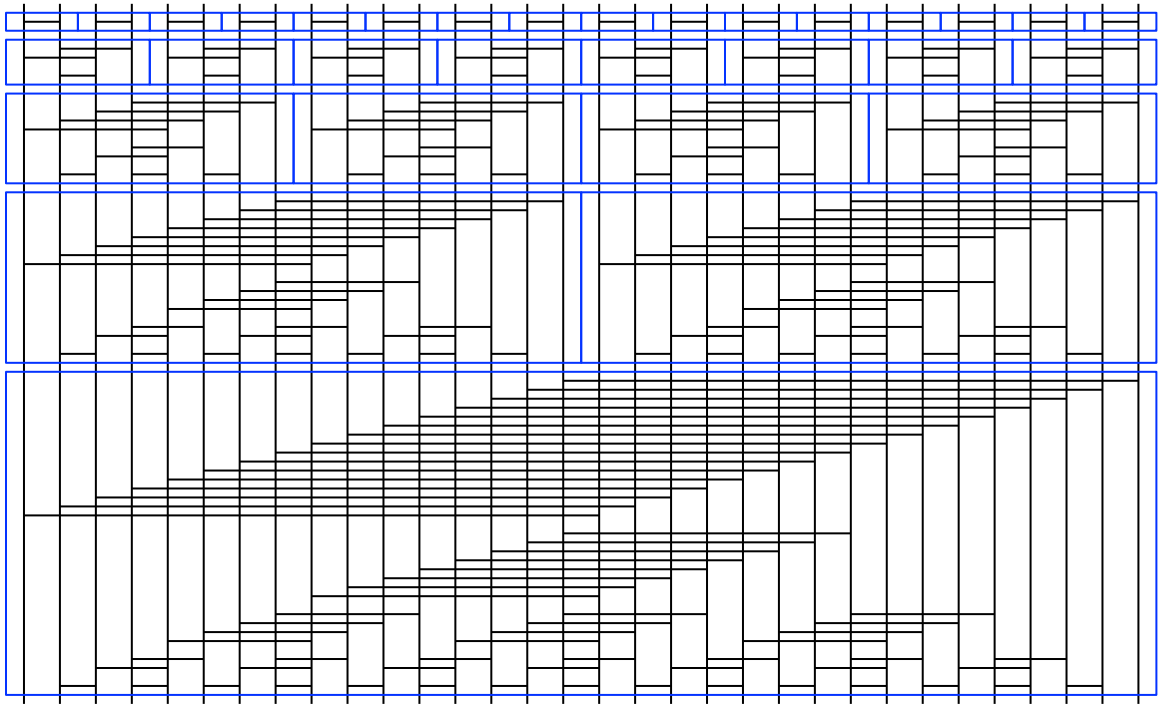


Sort Primitive

- 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)

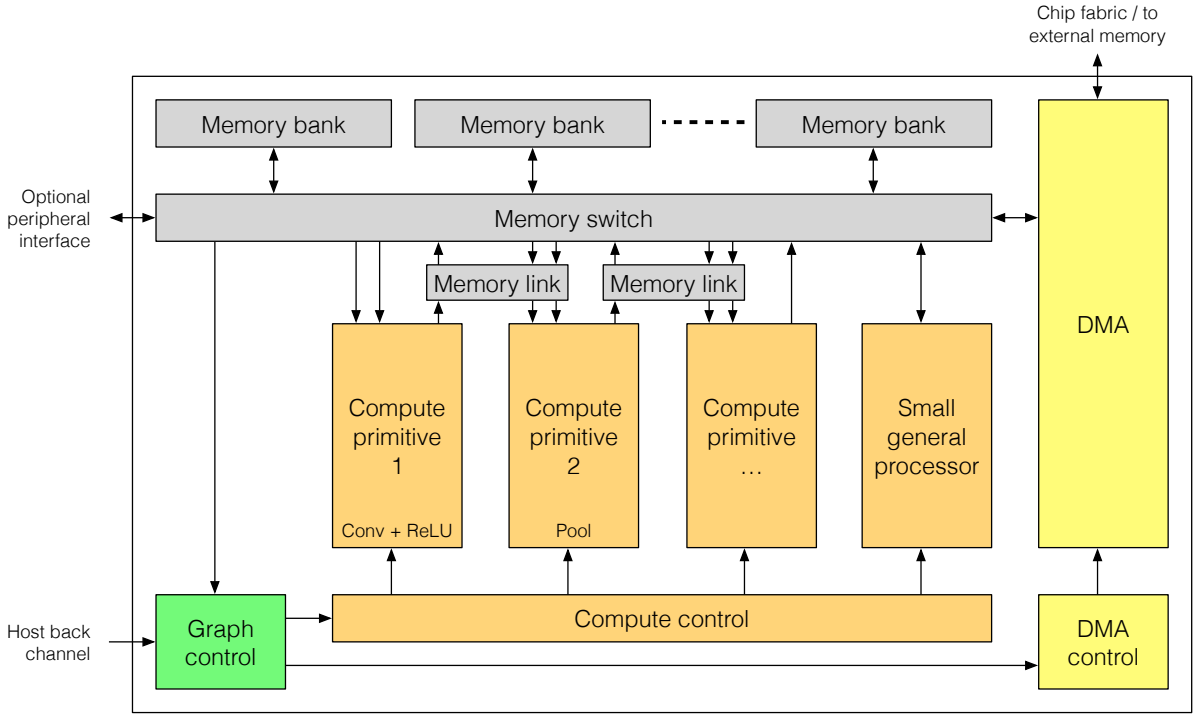
Sort Primitive Uses

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

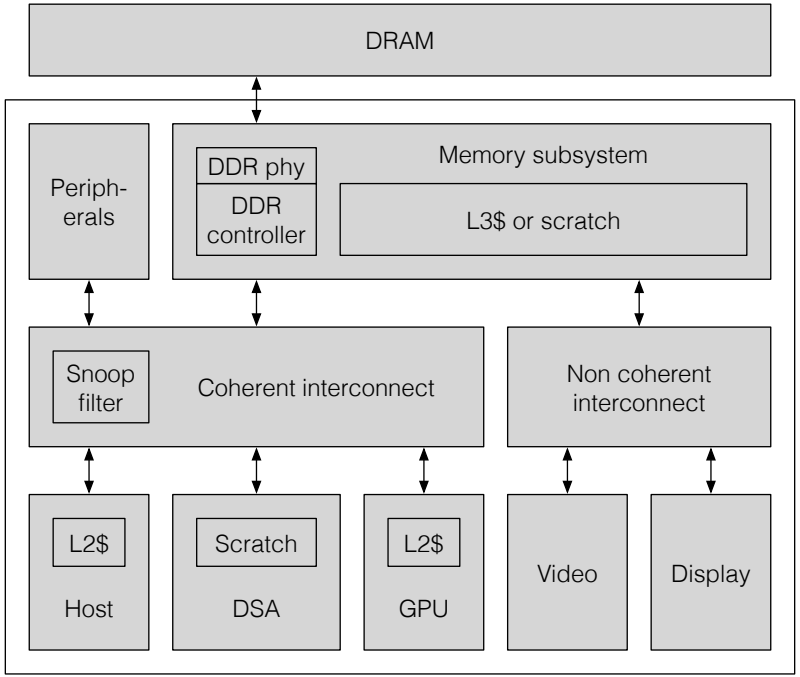
- Cleanup for anything that can't be mapped to 1 of the current computational primitives
- Use for generality
- Use for future proofing



Configurations

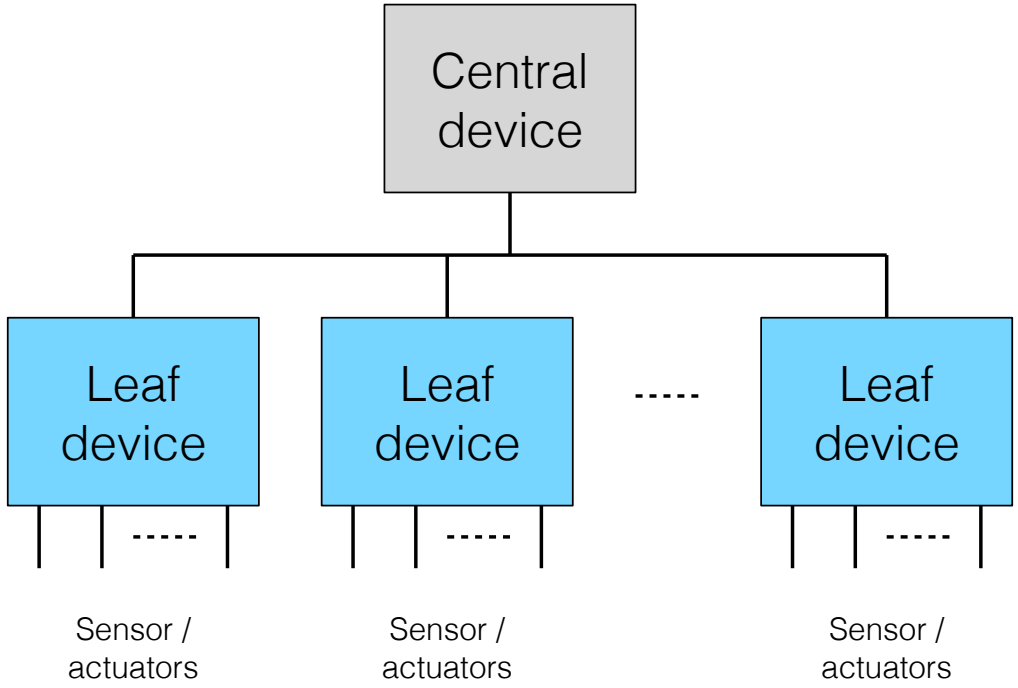
Individual Device

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



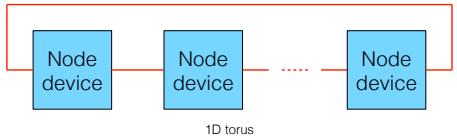
Multiple Devices In A Tree Configuration

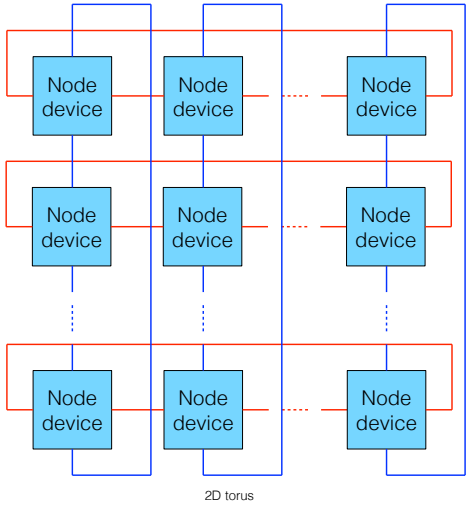
- Example sensor / actuator use
  - Disjoint sensor processing via xNNs in edge devices
  - Higher level fusion in a central device leading to decisions; possibly xNN based, possibly shallow combination
  - Disjoint actuation in edge devices based on central device decisions
- Example training use
 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





Examples

ARM Reference SoC Diagram

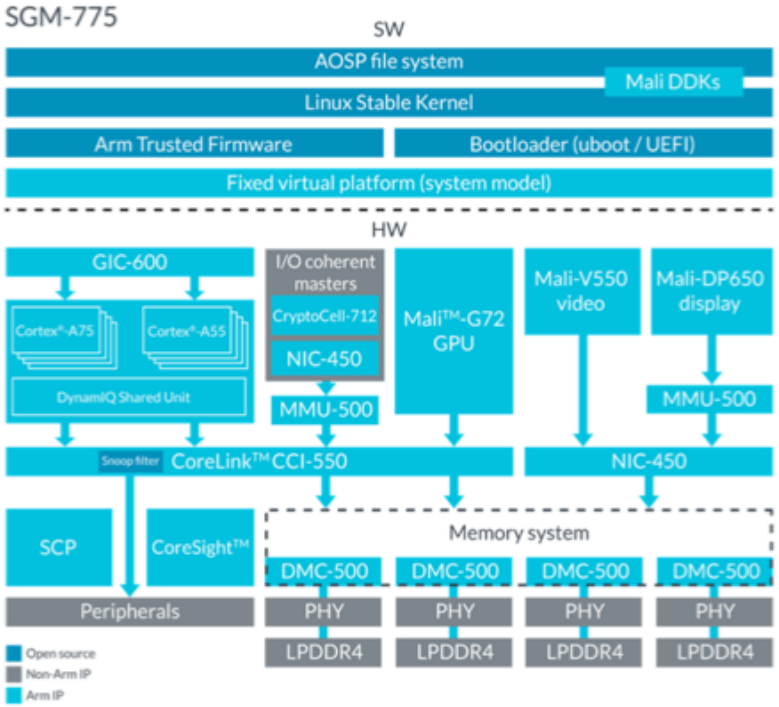
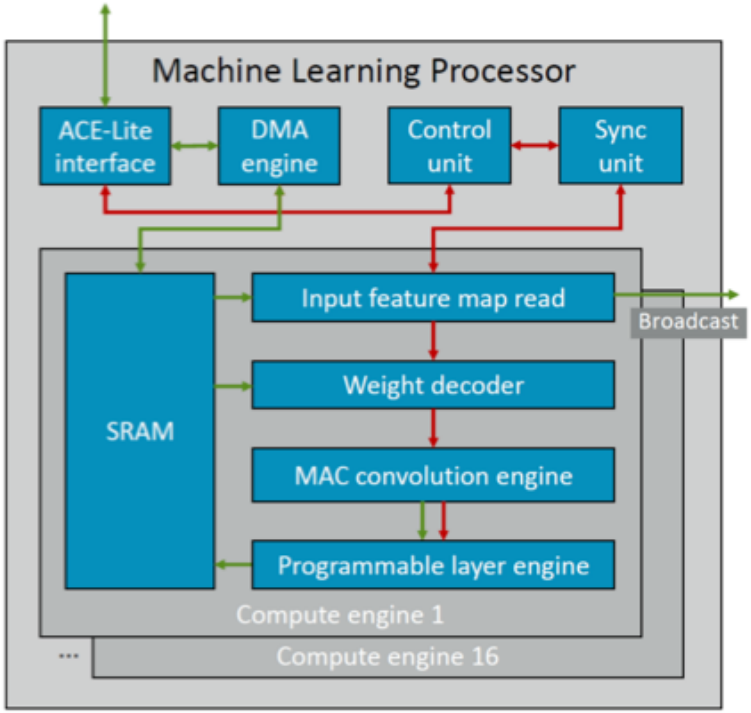


Figure from https://developer.arm.com/products/system-ip

ARM ML Processor



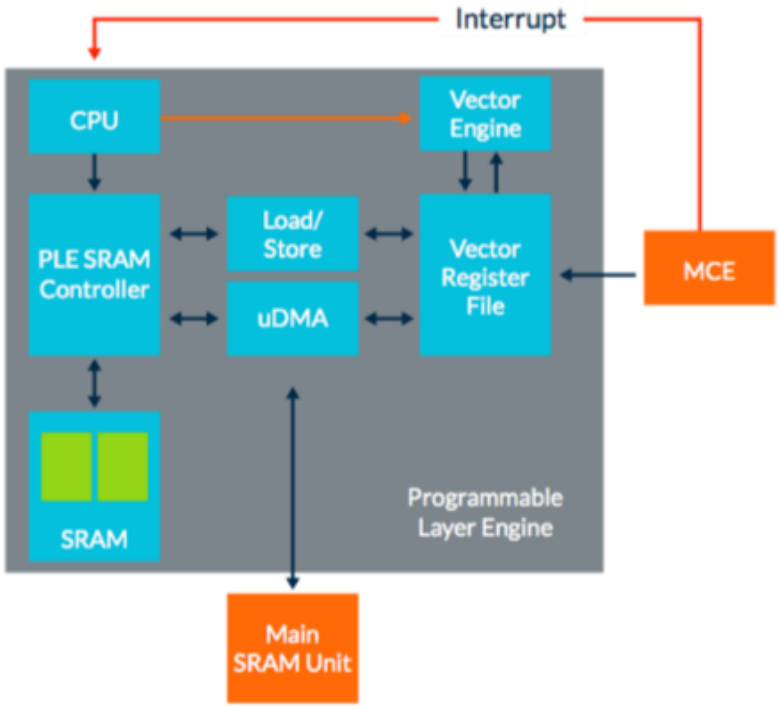


Figure from https://www.anandtech.com/show/12791/arm-details-project-trillium-mlp-architecture

Cadence Tensilica DNA 100

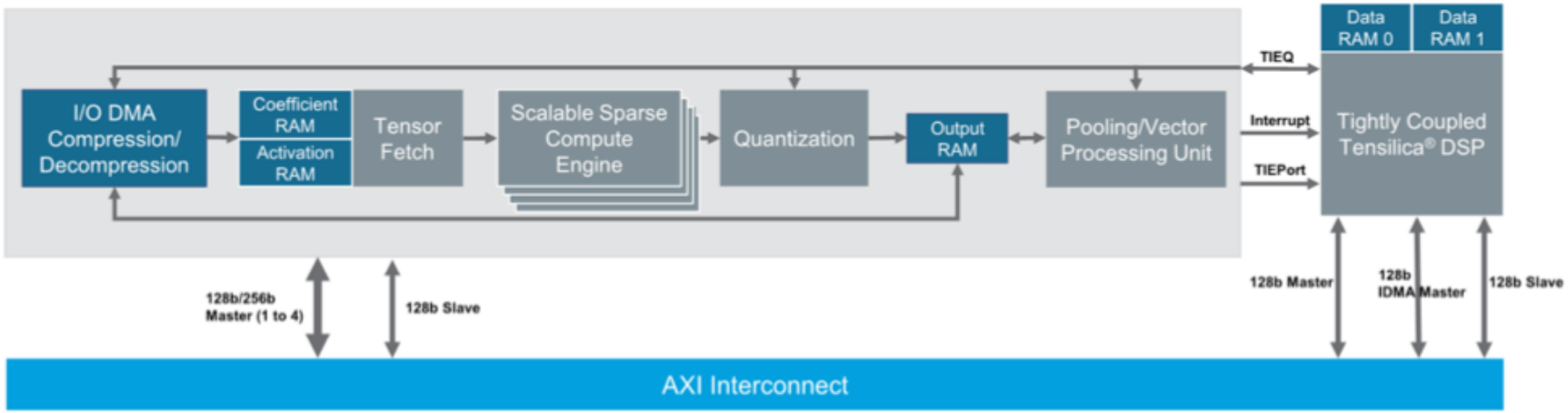
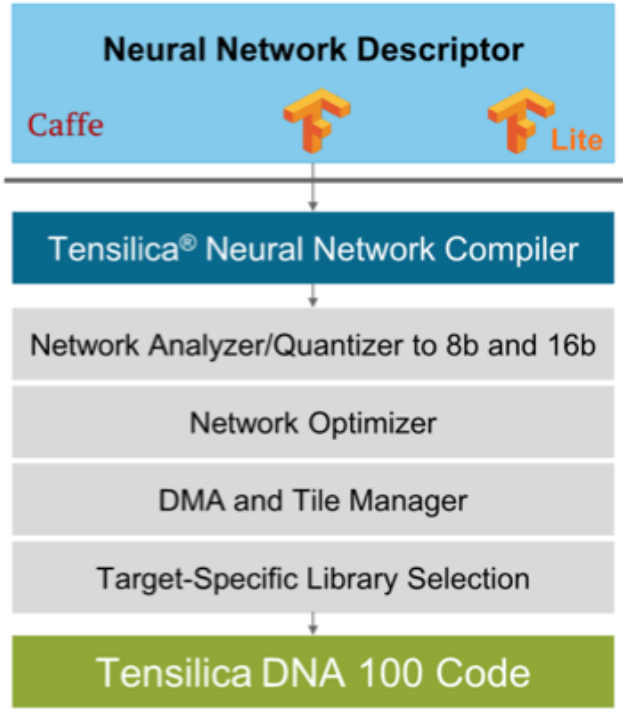


Figure from https://www.anandtech.com/show/13377/cadence-announces-tensilica-dna-100-a-bigger-nn-ip

Cadence Tensilica DNA 100



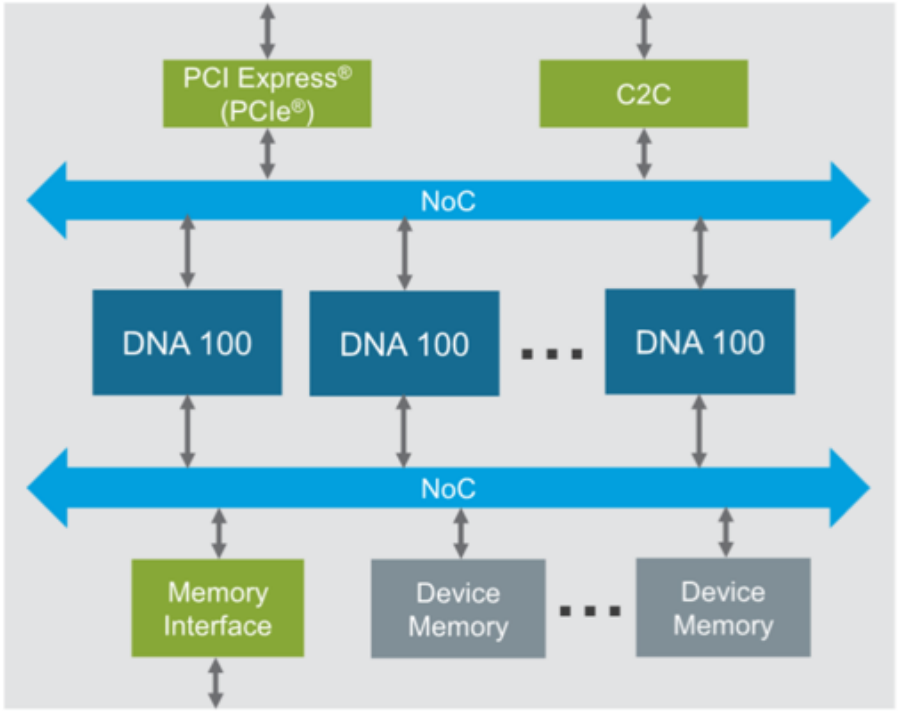


Figure from https://www.anandtech.com/show/13377/cadence-announces-tensilica-dna-100-a-bigger-nn-ip

CEVA CDNN And NeuPro

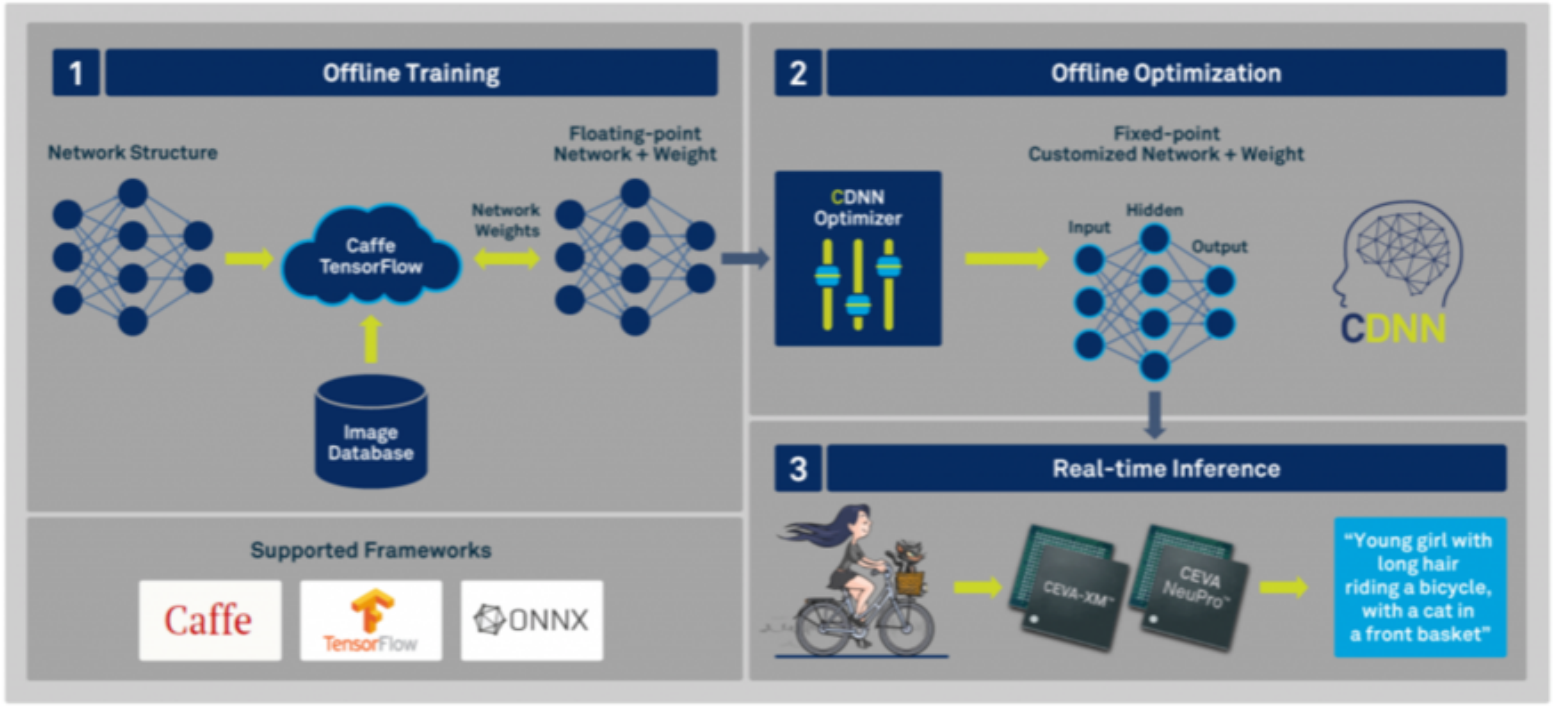


Figure from https://www.ceva-dsp.com/product/ceva-deep-neural-network-cdnn/

CEVA CDNN And NeuPro

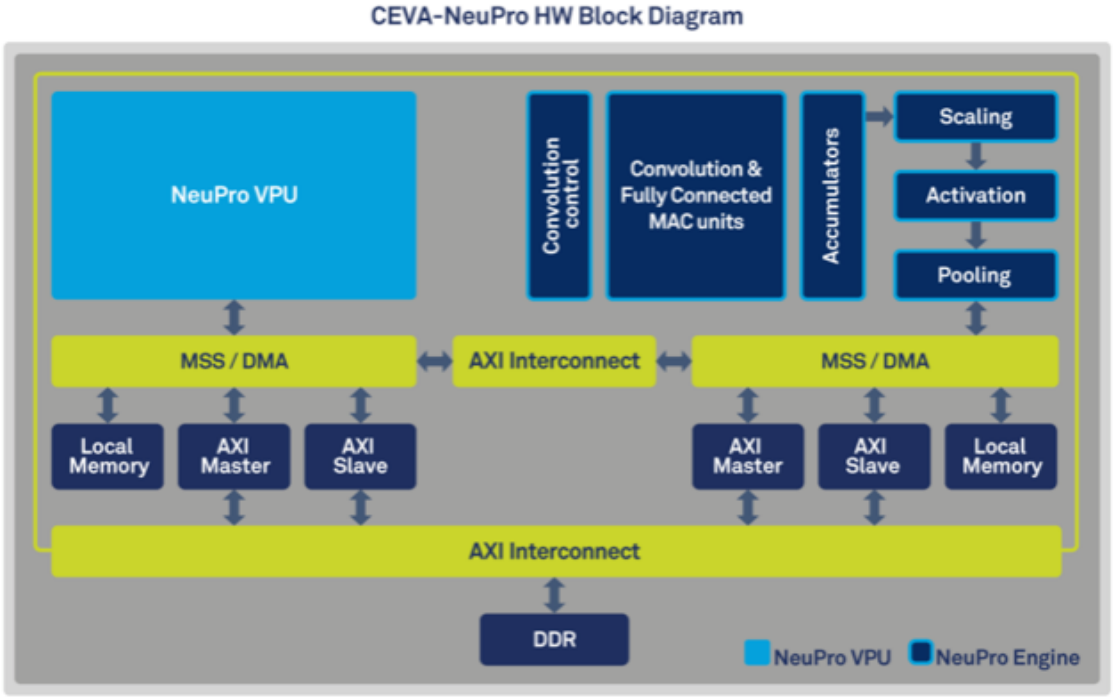


Figure from https://www.ceva-dsp.com/product/ceva-neupro/

Google TPU

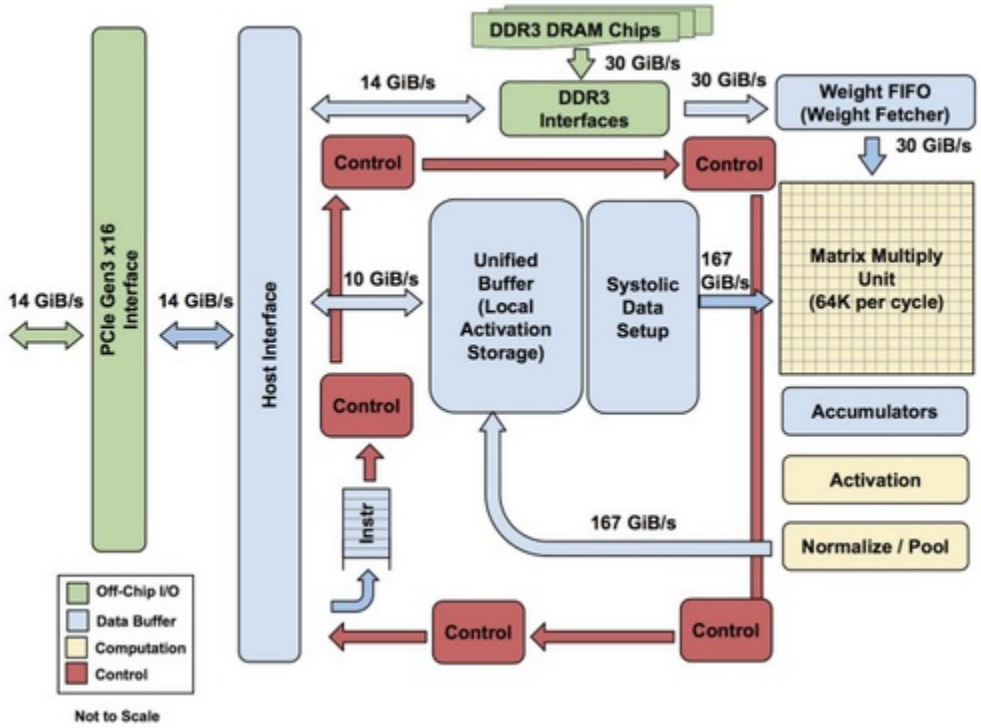
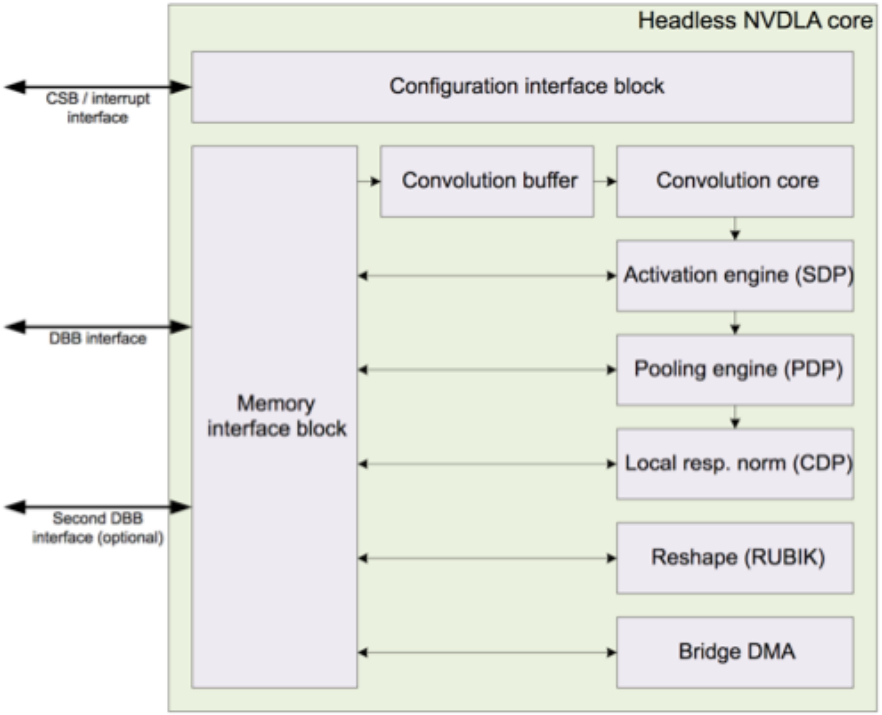


Figure from https://www.nextplatform.com/2017/04/05/first-depth-look-googles-tpu-architecture/

Nvidia NVDLA



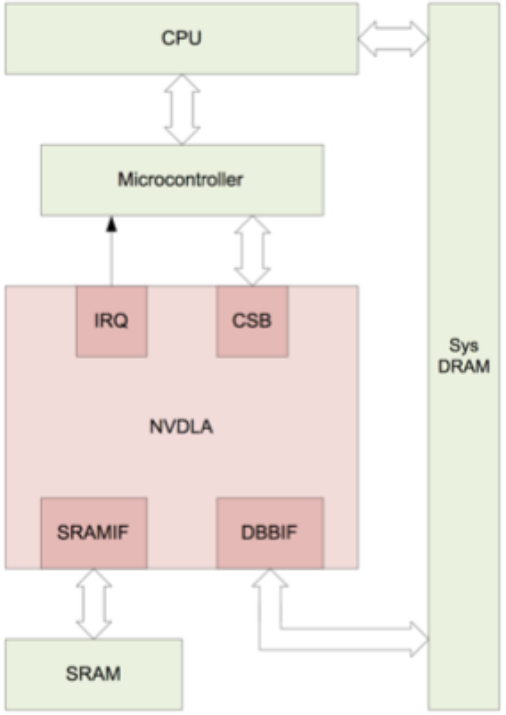


Figure from http://nvdla.org/primer.html

Nvidia Turing

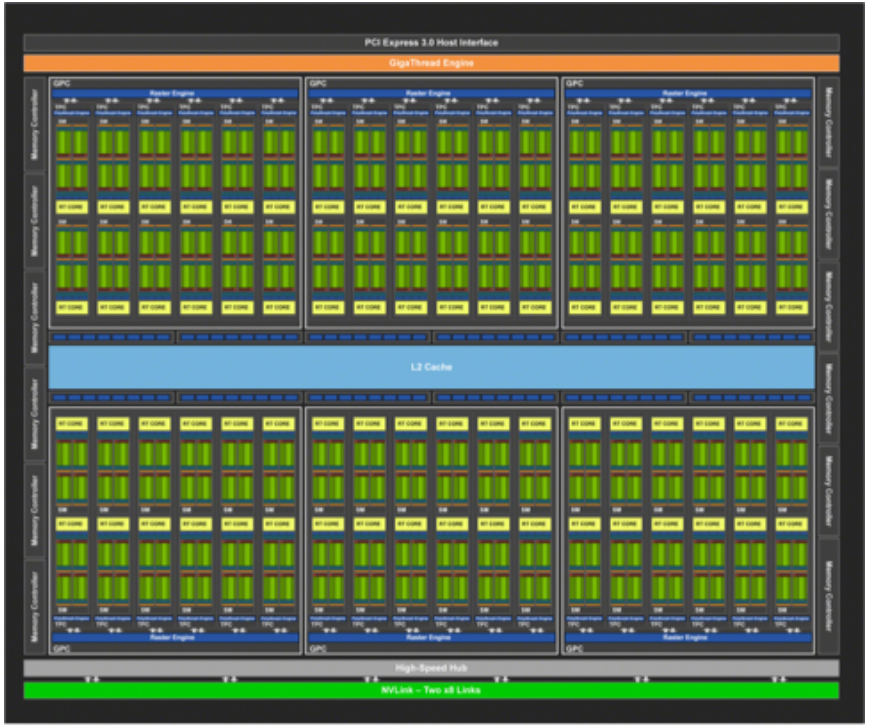


Figure from https://www.anandtech.com/print/13282/nvidia-turing-architecture-deep-dive

Nvidia Turing

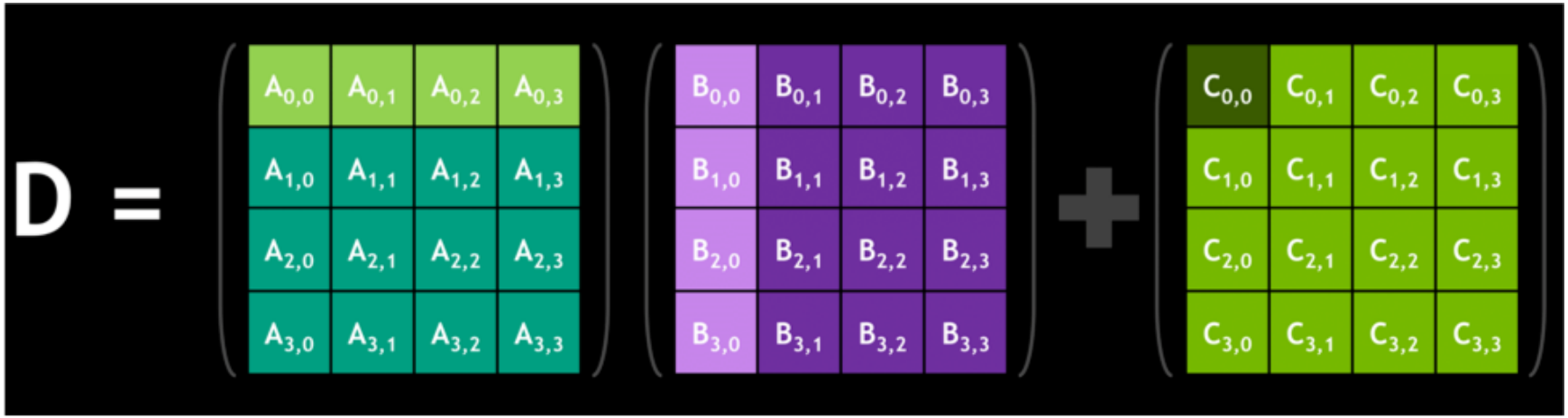


Figure from https://www.anandtech.com/print/13282/nvidia-turing-architecture-deep-dive

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

Xilinx

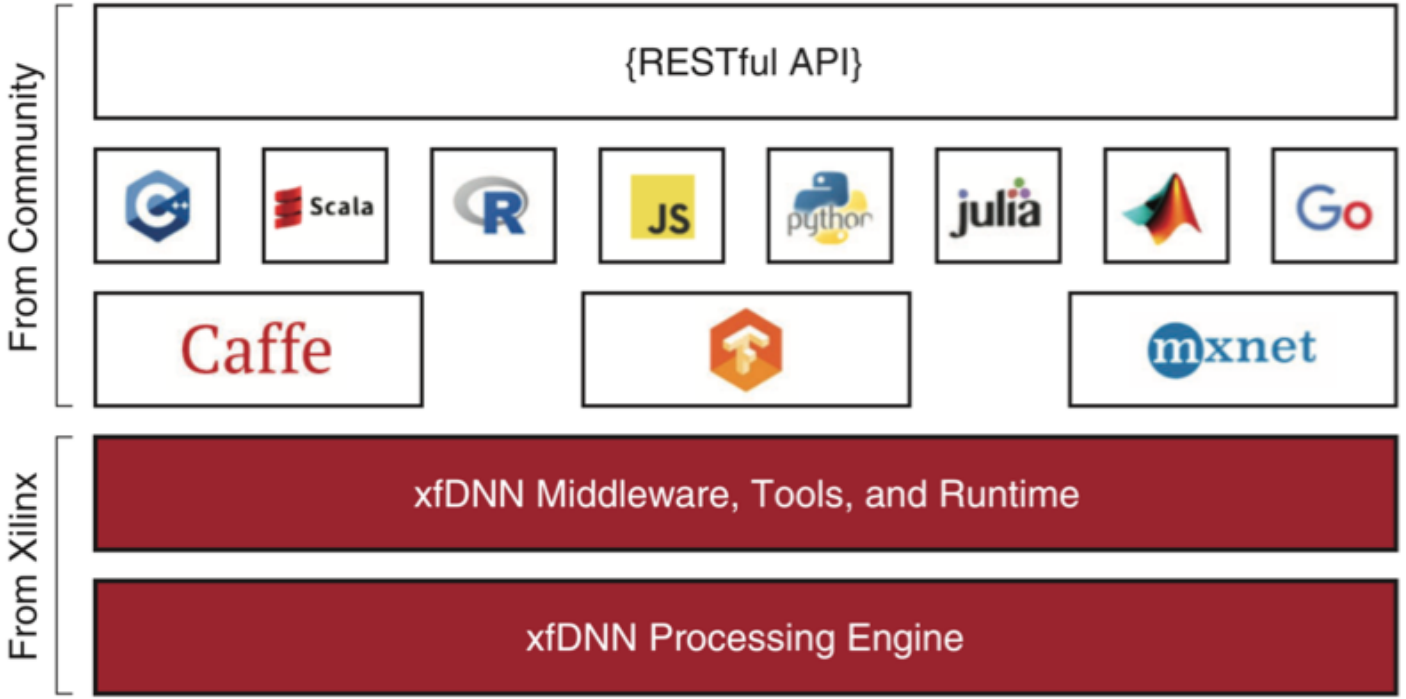
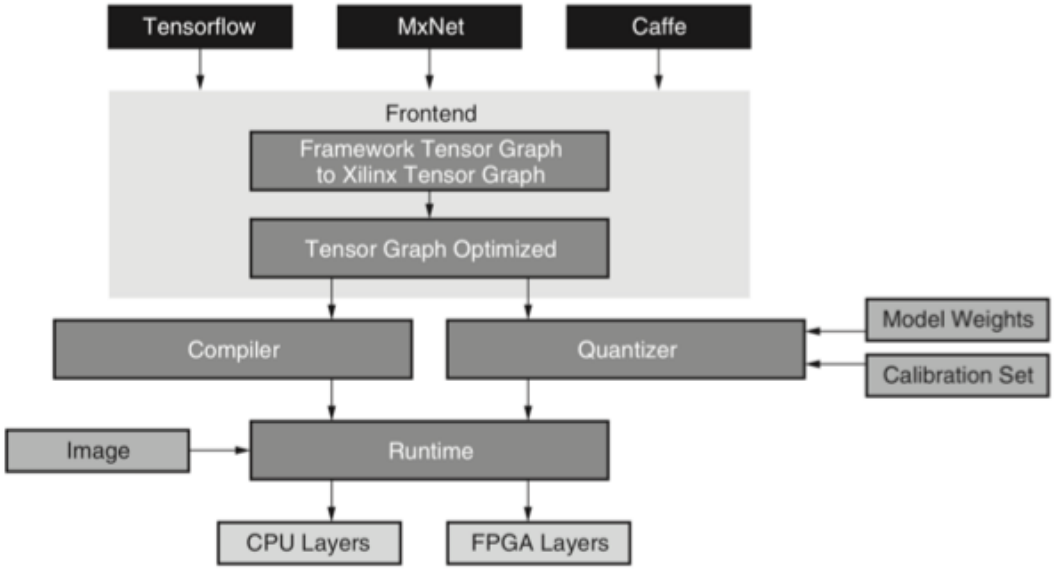


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx



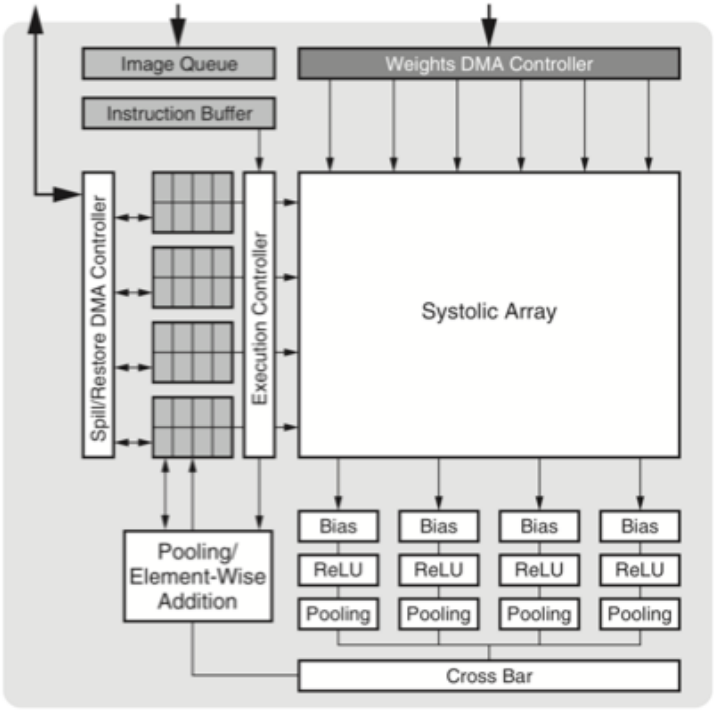


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx

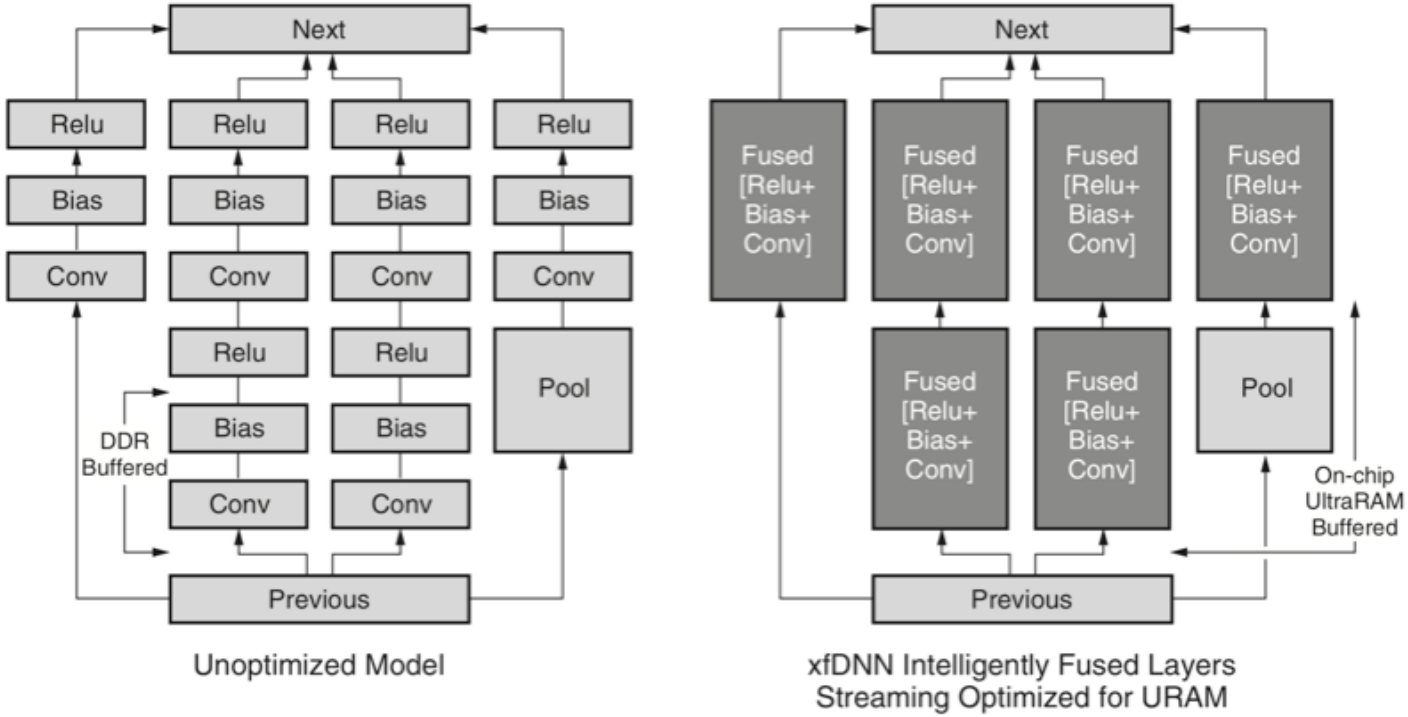
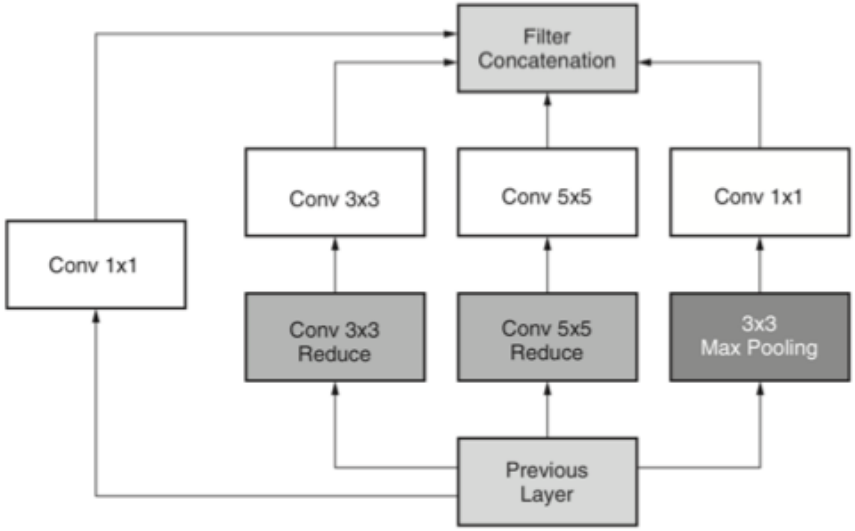
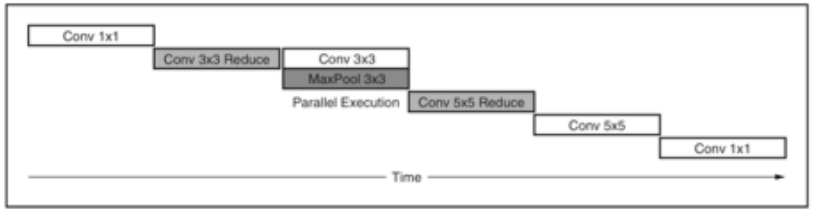


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx





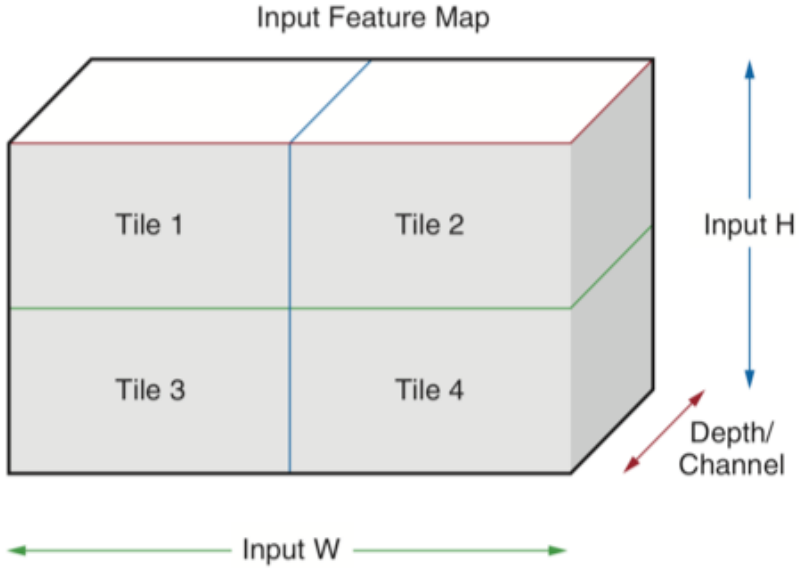


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Xilinx

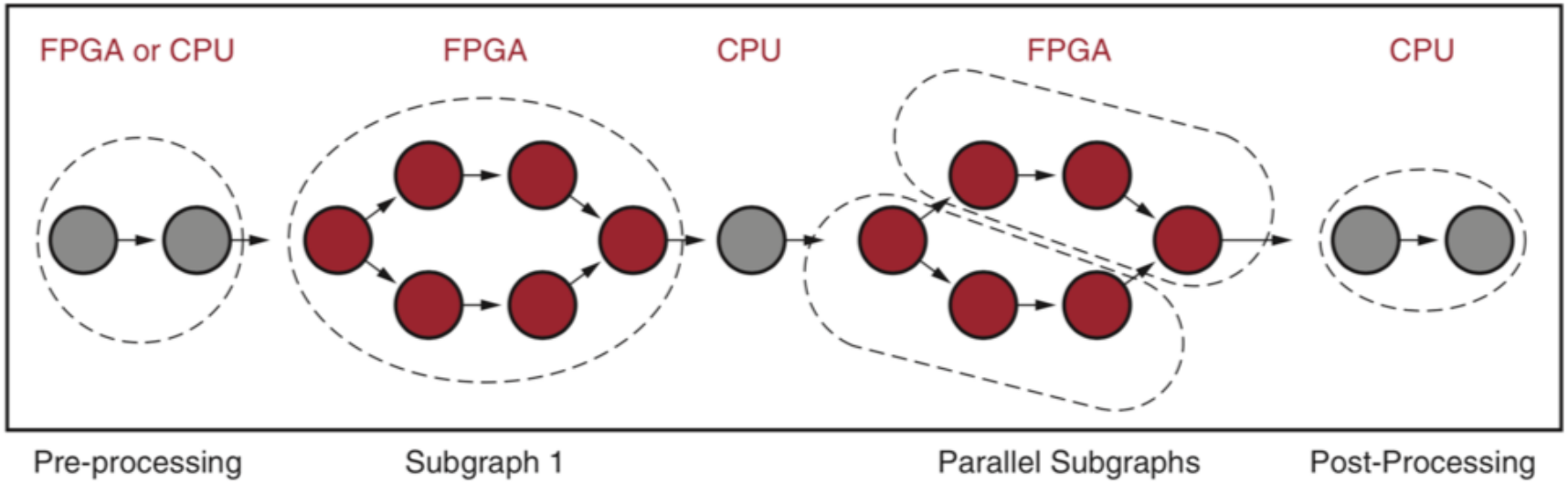
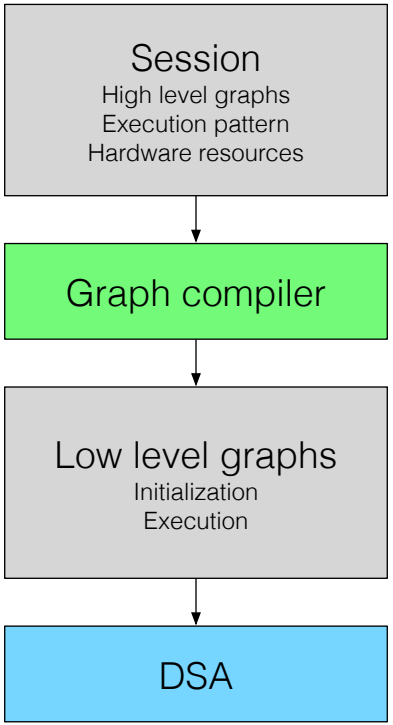


Figure from https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

Software

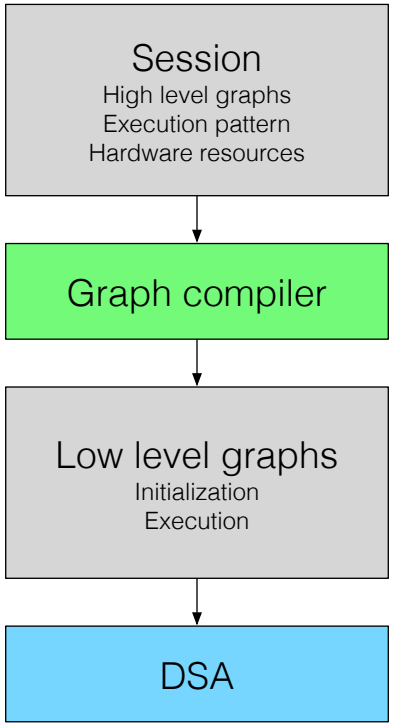
Flow

- High level graphs
  - Network specification
- Session
  - The graph compiler input
  - High level graphs + execution pattern
  - Hardware resources
- Graph compiler
  - Compile high level graphs to low level graphs
  - Edges are memory and nodes are operators
  - Separate initialization and execution graphs
  - Exploits compile time information
  - Everything is on the graph to simplify hardware
  - Fully deterministic



Flow

- Low level graphs
  - Map 1 to 1 to hardware
  - Node descriptors encode structure
  - Node instructions control operations
  - Operations include instruction movement, data movement and compute
- Runtime
  - Initialization
    - Called 1x
    - Link static data
    - Run initialization graph
  - Execution
    - Called for each new input
    - Link dynamic data
    - Run execution graph



High Level Graph Specification

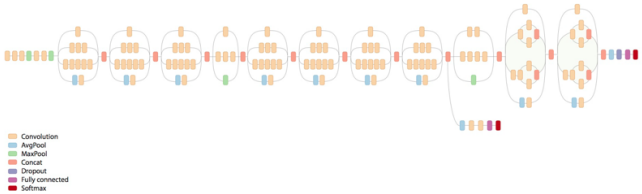
- Specification
  - (Relatively) hardware agnostic directed acyclic graphs
  - Edges represent memory, nodes represent operators
    - Variables are dynamic memory
    - Constants are static memory
    - Functions are operators
  - TensorFlow, PyTorch, Caffe, OpenVX …

```
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

Transformation

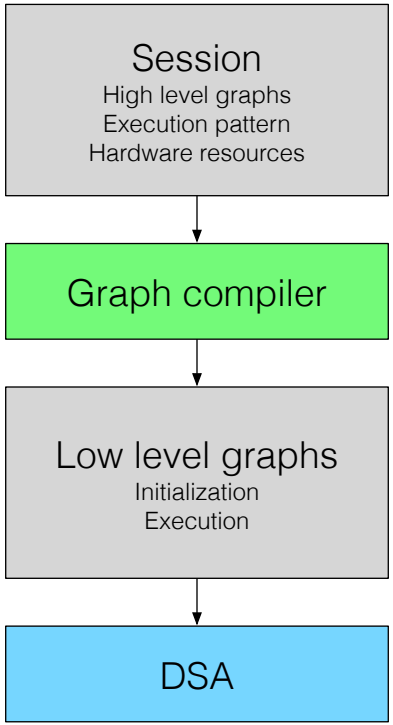
- Application and hardware agnostic transformations
  - Ex: unused edge / node pruning
- Application specific transformations
  - Ex: absorbing batch norm into convolution for inference
  - Ex: removing dropout and normalizing fully connected layer weights
  - Ex: converting from floating to fixed point
- Hardware specific transformations
  - Ex: merging ReLU(convolution + bias) into 1 node
- Transformation to a common representation
  - Multiple high level graph specific environments are allowed
  - Transforming them all to a common format allows for a single graph compiler
  - LLVM compiler framework style; ONNX is a step in this direction

Static Vs Dynamic

- Note
  - You will hear a lot of discussion on static vs dynamic graphs and eager vs graph execution
  - Dynamic eager execution offers the most convenience for development
  - Static graph execution offers the potential for the highest level of performance
  - For some additional discussion on this see https://www.tensorflow.org/guide/eager

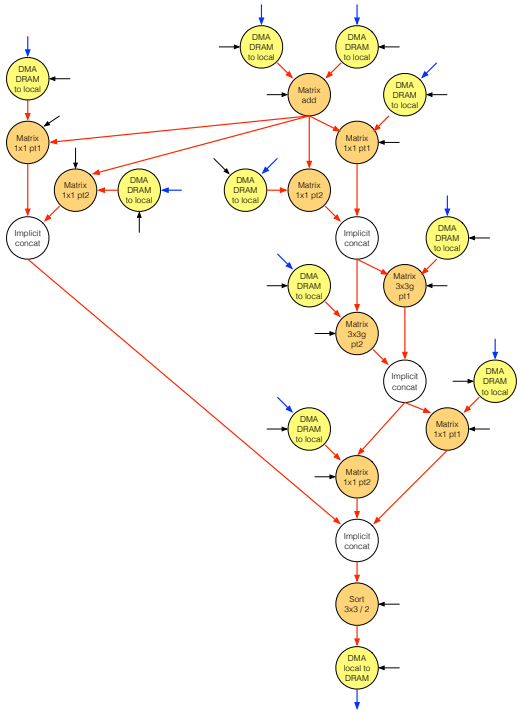
Session

- What is the likely high level graph execution pattern for the case of multiple graphs? What hardware resources are available for mapping the graphs onto?
  - The session encapsulates this information
- The session defines
  - High level graphs and their likely execution pattern (if known)
  - Available hardware resources (accelerator framework, external memory pools, external memory interfaces)
  - Available compute and DMA libraries (with performance)
- This information is used by the graph compiler to create low level graphs
  - Remember that xNNs typical require large amounts of memory, data movement and compute
  - Having a full problem in a compact graph format along with available resources gives the graph compiler a full view of the problem



Graph Compiler

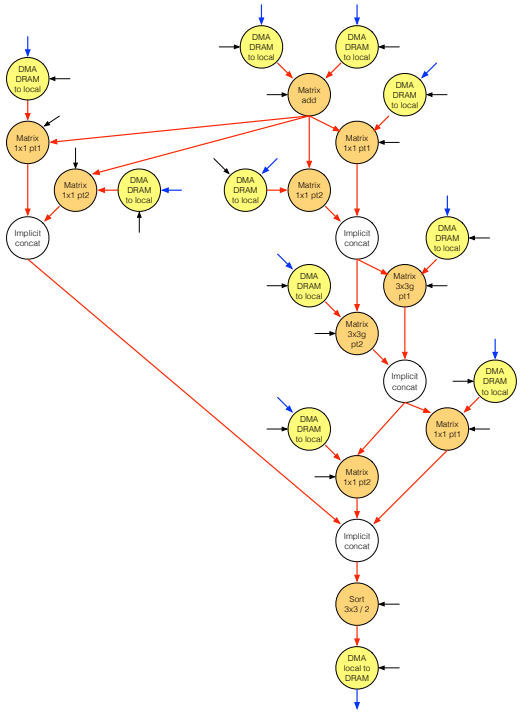
- The graph compiler transforms high level graphs suitable for application development to low level graphs suitable for controlling hardware using information specified in the session
  - Offline / compile time optimization and the session information provide the potential for optimality and simplicity (e.g., parameter dependent code optimization)
- Everything (everything) is specified on the low level graph
- Low level graphs are designed such that there's a 1 to 1 mapping between
  - Edges on the graph and memory
  - Nodes on the graph and components of the DSA
- To accomplish this, high level graph nodes are typically decomposed into multiple low level nodes and augmented with additional nodes to make instruction and data movement explicit



Low Level Graphs

- Low level graphs generated by the graph compiler
  - Initialization - graph portion associated with static data
  - Execution - graph portion associated with dynamic data

- Low level graphs are composed of edges and nodes
  - Edges represent memory
    - Node descriptors and node instructions
    - Data (feature maps, filter coefficients, ...)
    - 1 to 1 mapping to DRAM and local memory locations
  - Nodes represent an operation
    - Control
    - Compute
    - Memory movement
    - 1 to 1 mapping to DSA components



Runtime Initialization

1x

- Data is either static or dynamic with respect to location in memory
  - External edges
    - Input and output data is likely dynamic (though with the same alignment and in the same place in the memory hierarchy)
    - **Filter coefficients are likely static**
    - **Node descriptors and instructions are likely static**
  - Internal edges
    - **Are likely static**

- Link static data
  - Node instructions include memory locations
  - The purpose of linking static data is to set these static memory locations

- Run initialization graph
  - Process (an optimized subset of) nodes with static locations and contents

Runtime Execution

1x for every new input

- Data is either static or dynamic with respect to location in memory
  - External edges
    - **Input and output data is likely dynamic (though with the same alignment and in the same place in the memory hierarchy)**
    - Filter coefficients are likely static
    - Node descriptors and instructions are likely static
  - Internal edges
    - Are likely static

- Link dynamic data
  - Input and output memory locations
  - The purpose of linking dynamic data is to set these dynamic memory locations

- Run execution graph
  - Process all remaining nodes

Android NDK

Similar in style / design to the previously mentioned software flow

- Applications run in a high level library such as TensorFlow Lite or Caffe2
- The high level library calls the Android NN API (a C API)
- The Android Neural Network Runtime distributes the computation
  - Hardware accelerators when available via HAL
  - The host as a fallback

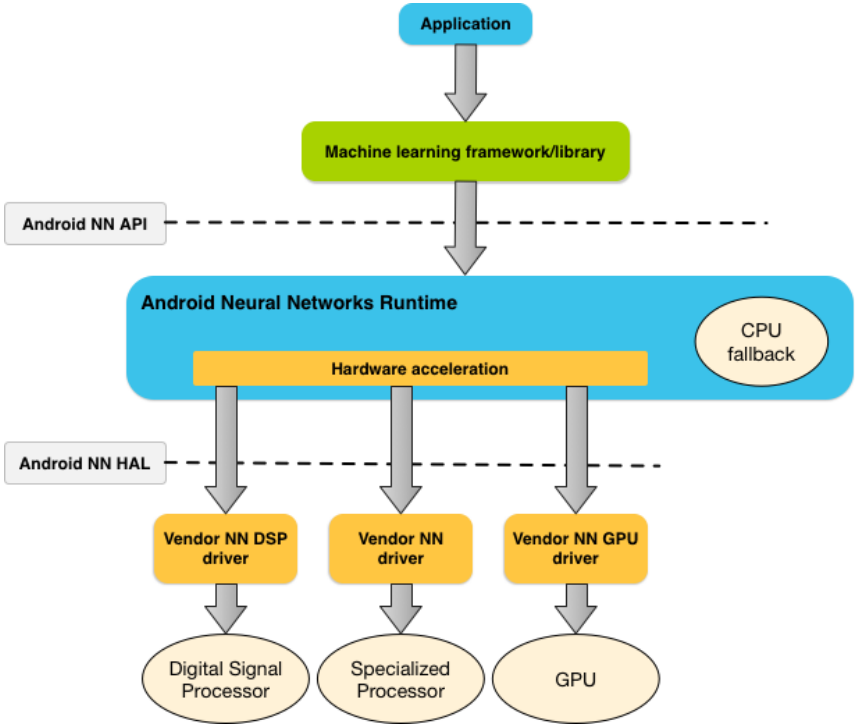


Figure from https://developer.android.com/ndk/guides/neuralnetworks/

Android NDK

Similar in style / design to the previously mentioned software flow

- Memory
  - Buffers for tensors
  - Buffers for inputs and outputs
- Model
  - A computational graph
  - Operands are edges, operations are nodes
- Compilation
  - Create low level code representing the model
- Execution
  - Apply the compiled model to a specific set of inputs to generate a specific set of outputs
  - Repeat for each input / output pair

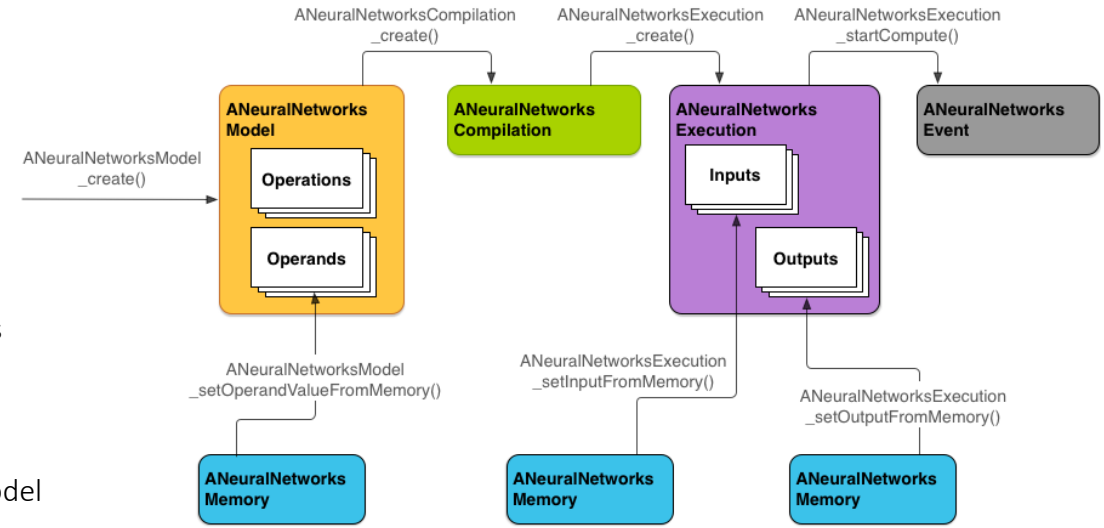
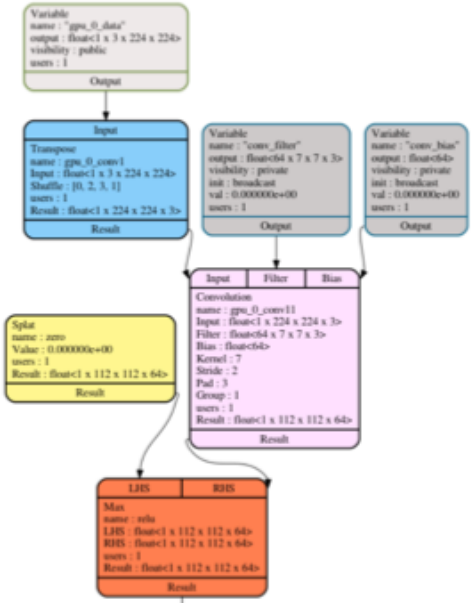


Figure from https://developer.android.com/ndk/guides/neuralnetworks/

GLOW

Similar in style / design to the previously mentioned software flow



High-Level Graph

```
declare {
 %input = weight float<8 x 28 x 28 x 1>, broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>, xavier, 25.0
 %filter0 = weight float<16>, broadcast, 0.100
 %weights = weight float<10 x 144>, xavier, 144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo, @in %input,
     @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in %allo0,
     @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

Low-Level IR

```
LBB14_1:
  vmovaps 3211264(%rcx,%rax,4), %ymm1
  vmovaps 3211296(%rcx,%rax,4), %ymm2
  vmovaps 3211328(%rcx,%rax,4), %ymm3
  vaddps 6422528(%rcx,%rax,4), %ymm1, %ymm1
  vaddps 6422560(%rcx,%rax,4), %ymm2, %ymm2
  vmovaps 3211360(%rcx,%rax,4), %ymm4
  vaddps 6422592(%rcx,%rax,4), %ymm3, %ymm3
  vaddps 6422624(%rcx,%rax,4), %ymm4, %ymm4
  vmaxps %ymm0, %ymm1, %ymm1
  vmaxps %ymm0, %ymm2, %ymm2
  vmaxps %ymm0, %ymm3, %ymm3
  vmovaps %ymm1, 6422528(%rcx,%rax,4)
  vmovaps %ymm2, 6422560(%rcx,%rax,4)
  vmaxps %ymm0, %ymm4, %ymm1
  vmovaps %ymm3, 6422592(%rcx,%rax,4)
  vmovaps %ymm1, 6422624(%rcx,%rax,4)
  addq $32, %rax
```

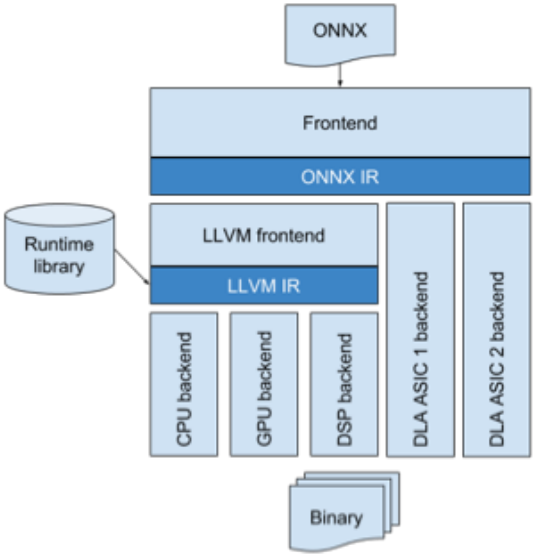
Machine Code

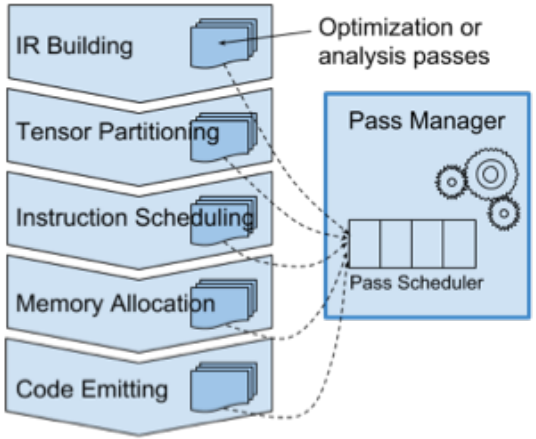
Figure from https://github.com/pytorch/glow

Open Neural Network Compiler

- 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

TVM

Similar in style / design to the previously mentioned software flow

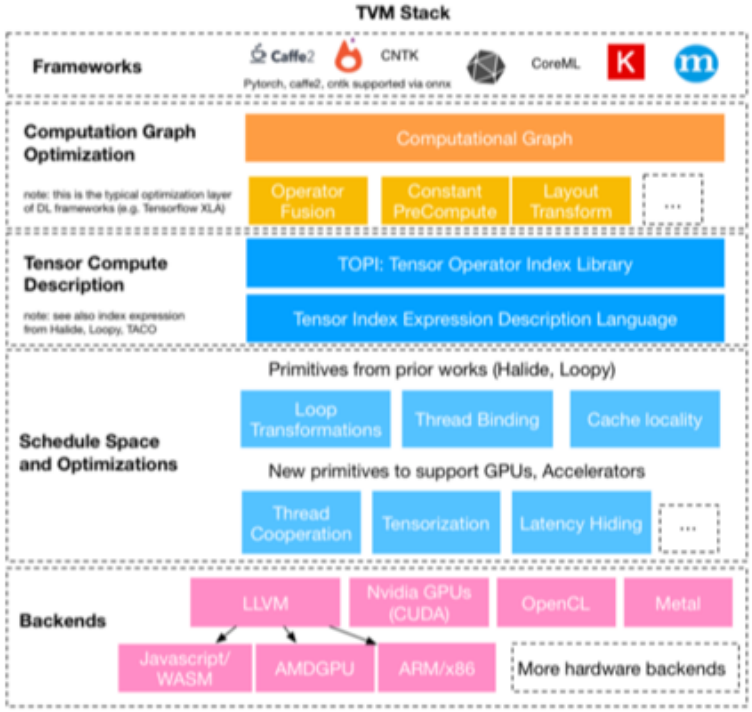


Figure from https://tvm.ai/about

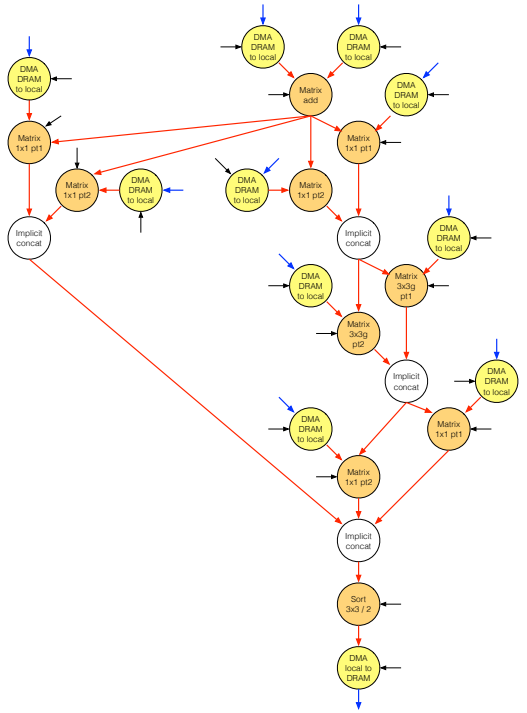
Performance

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

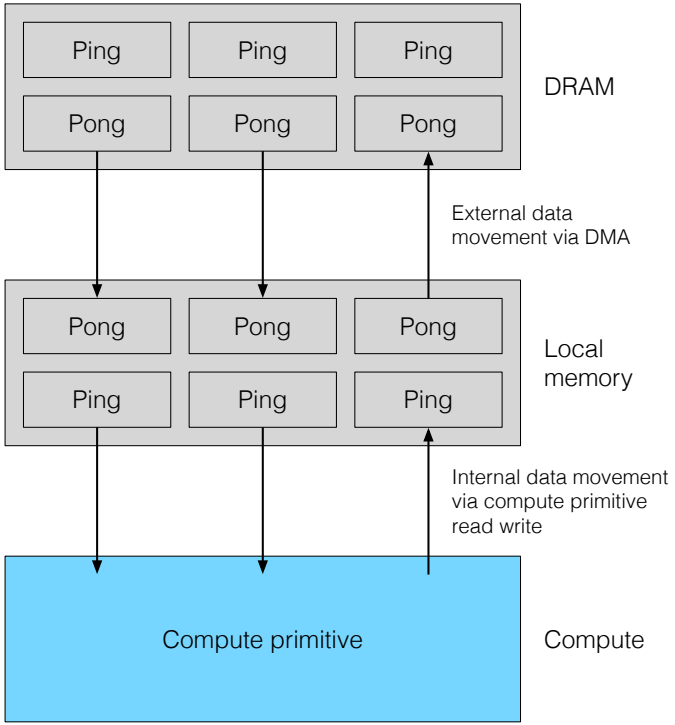
Predicting Performance

- 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

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

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/

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