Architectures and Platforms for Artificial Intelligence Notes

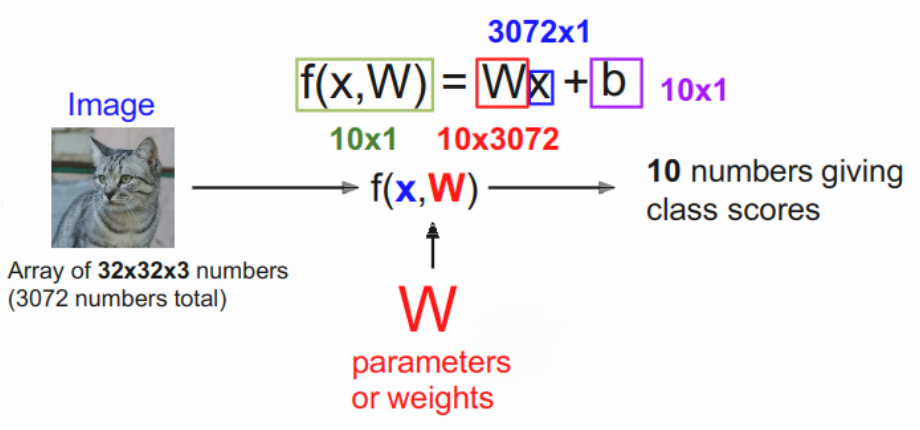
Author: Mattia Orlandi

1. Introduction

Artificial Intelligence: "The science and engineering of creating intelligent machines"

Machine Learning: "Field of study that gives computers the ability to learn without being explicitly programmed"

1.1. Linear Classifier

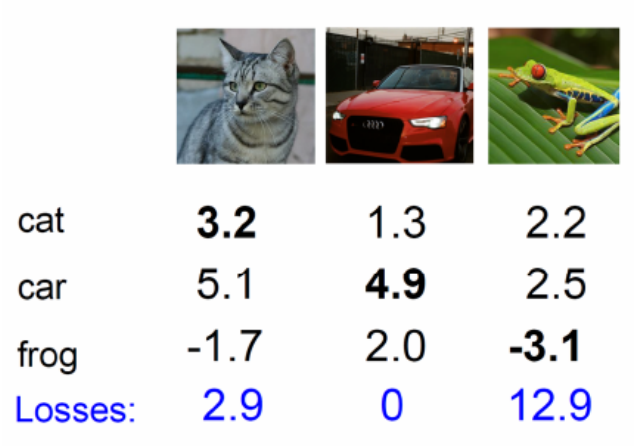


Each row of the weight matrix could be rearranged in an image, representing the pattern corresponding to the class the row is referring to.

Given N classes, if we project the weights matrix and the bias vector into an N-dimensional space, each row (together with the corresponding bias) represents the coordinates of an hyperplane. Such hyperplanes mark the boundaries between classes.

Evaluation *Loss function*: it tells how good our current classifier is; given a dataset of examples $\{(x_i, y_i)\}^N$, where x_i is the image and y_i the label, the loss over the dataset is the sum of the loss over the single examples $L = \frac{1}{N}\sum_i L_i\left(f(x_i, W), y_i\right)$.

Example: Multiclass SVM loss Having defined the scores vector of the i-th example as $s_i = f(x_i, W)$, the SVM loss has the form $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$ (hinge loss).



Loss for each example:

1. True label: *cat*

$$L_1 = \max(0, 5.1 - 3.2 + 1) + \max(0, -1.7 - 3.2 + 1) = 2.9$$

2. True label: *car*

$$L_2 = \max(0, 1.3 - 4.9 + 1) + \max(0, 2.0 - 4.9 + 1) = 0$$

3. True label: *frog*

$$L_3 = \max(0, 2.2 - (-3.1) + 1) + \max(0, 2.5 - (-3.1) + 1) = 6.3 + 6.6 = 12.9$$

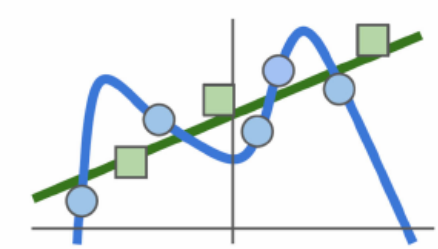
Loss over the full dataset:

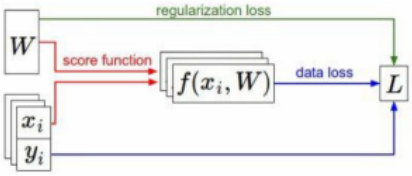
$$L = \frac{1}{3}(L_1 + L_2 + L_3) = \frac{(2.9 + 0 + 12.9)}{3} = 5.27$$

Regularization

- Data loss: model's predictions should match training data.
- Regularization: model should be as simple as possible (*Occam's Razor*).

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$





Training To find the best W, **gradient-based optimization**, in particular **Stochastic Gradient Descent** (SGD), is employed.

GD pseudo-code:

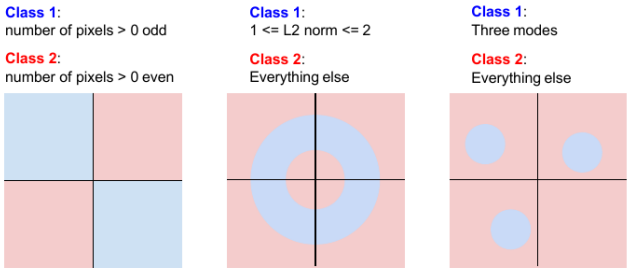
```
while True:
    weights_grad = evaluate_gradient(loss_fn, data, weights)
    weights += -step_size * weights_grad  # parameter update
```

Since computing the loss over the full dataset is expensive, in SGD such sum is approximated using a minibatch of examples.

SGD pseudo-code:

```
while True:
    data_batch = sample_training_data(data, 256)  # sample 256 examples
    weights_grad = evaluate_gradient(loss_fn, data_batch, weights)
    weights += -step_size * weights_grad  # parameter update
```

Limits of Linear Classifiers They cannot handle cases in which data is **not linearly-separable**.



1.2. From Linear to Non-Linear Classifiers

- **Kernel trick**: apply a feature transform s.t. in the new space data becomes linearly separable.
- **Feature extraction**: analyze the image and extract features with a given algorithm:
  - color histogram;

– histogram of gradients (HoG);
– **convolutional networks**.

1.3. Brain-Inspired Machine Learning

Brain-Inspired: an algorithm that takes its basic functionality from our understanding of how the brain operates.

The basic computational unit of the brain is the neuron:

- 86 billion neurons;
- from 10^{14} to 10^{15} *synapses*.

Neurons receive input signals from *dendrites* and produce output signals along *axons*, which interacts with other neurons' dendrites via **synaptic weights**, which are learnable and control the influence strength.

Spiking-based Machine Learning Each artificial neuron integrates incoming signals and fires an output signal towards the following neurons (*e.g.* IBM TrueNorth).

Neural Networks Simplified model of human brain: layered structure in which each neuron of a given layer computes a weighted sum of the values received from the neurons of the previous layer, applies a non-linear activation function and sends the output value to the neurons of the next layer.

Deep learning is the field of study of deep neural networks (*i.e.* networks with many layers).

1.4. Deep Neural Networks (DNNs)

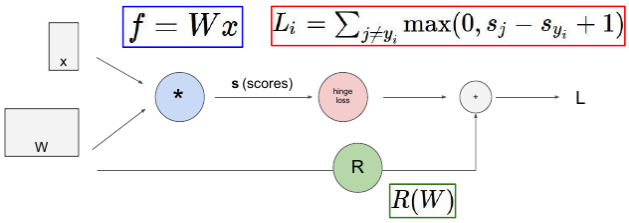
Terminology:

- *Weight sharing*: multiple synapses use the same weight value.
- *Activation*: neuron's outputs.
- *Fully connected*: all input neurons are connected to all output neurons.
- *Sparsely connected*: some input neurons are connected only to some output neurons.
- *Feed-forward*: connections between neurons do not form cycles.
- *Feedback*: connections between neurons form a directed graph along a temporal sequence.

Training Relies on gradient descent. The gradient can be computed with the following techniques:

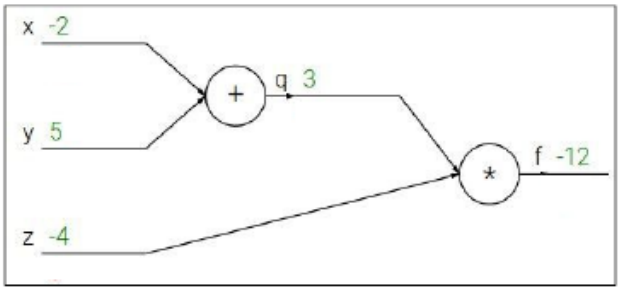
- *Numerical gradient*: slow, approximate but easy to write.
- *Analytical gradient*: fast, exact but error-prone.
- **Automatic differentiation**: fast, exact and easy to implement.

The **backpropagation** algorithm employs automatic differentiation, which relies on the *chain rule* and on *computational graphs*, namely graphs tracking the sequence of operations between input and output.



Example Given the loss function $f(x, y, z) = (x + y)z$, it can be decomposed in:

$$q = x + y, \quad \frac{\partial q}{\partial x} = 1, \ \frac{\partial q}{\partial y} = 1$$
$$f = qz, \quad \frac{\partial f}{\partial q} = z, \ \frac{\partial f}{\partial z} = q$$



By applying the chain rule, we obtain:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x} = z \cdot 1 = z$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y} = z \cdot 1 = z$$
$$\frac{\partial f}{\partial z} = q$$

For vectorized data, the derivatives is no longer a scalar but a matrix: the **Jacobian**.

Popular types of DNNs:

- *MultiLayer Perceptron (MLP)*: feed-forward, fully-connected network;
- **Convolutional NN (CNN)**: feed-forward, sparsely-connected networks with weight sharing;
- *Recurrent NN (RNN)*: feedback network;
- *Long-Short-Term Memory (LSTM)*: feedback + storage.

2. Convolutional Networks

Modern Deep CNNs for image classification comprises 5 to 1000 convolutional (CONV) layers, and only 1 to 3 fully connected (FC) layers (classifier); each CONV layer extract features, from low-level (initial layers) to high-level (final layers).

Each CONV layer comprises a convolution operation and an activation function; optionally, between CONV layers there can be normalization (NORM) and pooling (POOL) layers.

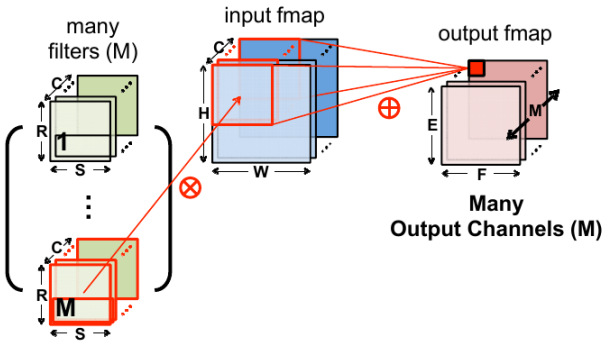
Convolutions accounts for more than 90% of overall computation, dominating runtime and energy consumption.

2.1. Convolutional Layer

The CONV layer processes an $H \times W$ input feature map (fmap) by applying on it a $R \times S$ sliding window filter, which performs an element-wise multiplication between filter weights and their corresponding input fmap values; then, a partial sum (psum) accumulation is performed, obtaining one single value (out of the $E \times F$) of the output fmap.

If there are C input channels, $C \times R \times S$ filters are used in order to obtain one single value of the output fmap.

If there are M output channels, $M \times C \times R \times S$ filters are used to obtain M values of the output fmap (one per channel).



In case of N batches, $M \times C \times R \times S$ filters are used on $N \times C \times H \times W$ input fmaps to produce $N \times M \times E \times F$ output fmaps.

Tensor computation

$$\mathbf{O}[n][m][x][y] = \text{Activation}(\mathbf{B}[m] + \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} \sum_{k=0}^{C-1} \mathbf{I}[n][k][Ux+i][Uy+j] \cdot \mathbf{W}[m][k][i][j])$$

$$0 \leq n < N,\ 0 \leq m < M,\ 0 \leq y < E,\ 0 \leq x < F$$

$$E = (H - R + U)/U,\ F = (W - S + U)/U$$

where:

- N fmap batch size;
- M number of filters/output fmap's channels;
- C number of input fmap/filter's channels;
- H, W input fmap's height and width;
- R, S filter's height and width;
- E, F output fmap's height and width;
- U convolution stride.

2.2. Activation functions

Traditional:

- Sigmoid: $y = 1/(1 + e^{-x})$;
- Hyberbolic tangent: $(e^x - e^{-x})/(e^x + e^{-x})$.

Modern:

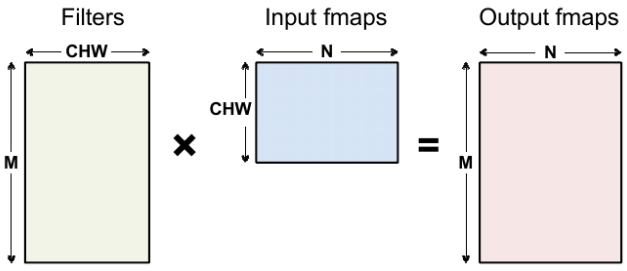
- Rectified Linear Unit (ReLU): $y = \max(0, x)$;
- Leaky ReLU: $y = \max(\alpha x, x)$, α;
- Exponential LU (ELU): x if $x \geq 0$, $\alpha(e^x - 1)$ otherwise.

2.3. Fully-Connected Layer

Equivalent to a CONV layer in which:

- height and width of output fmap are 1: $E = F = 1$;
- filters are as large as input fmaps: $R = H,\ S = W$.

It's implemented as a matrix multiplication between $M \times CHW$ filters and $CHW \times N$ input fmaps, resulting in a $M \times N$ output fmap.



2.4. Pooling Layer

- Reduces the resolution of each channel independently.
- Can be overlapping or non-overlapping, depending on stride.
- Increases translation invariance and noise resilience.

2.5. Normalization Layer

- Batch Normalization (BN) normalizes activations towards $\mu = 0$ and $\sigma^2 = 1$ along batch dimension.
- Put in between CONV/FC and Activation function, in order to avoid exploding/vanishing gradients:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

- Believed to be key to getting high accuracy and fast training on very deep NN.
- Other NORM variants normalize along channels (Layer Normalization) or along both batch and channels (Instance Normalization).

2.6. Popular DNNs

- *LeNet* (1998)
- **AlexNet** (2012): first below 20% top-5-error
- *OverFeat* (2013)
- **VGGNet** (2014)
- **GoogleNet** (2014)
- **ResNet** (2015): first below 5% top-5-error (better than humans)

AlexNet

- 5 CONV + 3 FC
- 61M weights
- 724M MACs
- ReLU non-linearity
- Local Response Normalization (LRN)

VGG-16 (also 19)

- 13 CONV + 3 FC
- 138M weights
- 15.5G MACs

GoogLeNet v1 (also v2, v3 and v4)

- 21 depth/57 total CONV + 1 FC
- 7M weights
- 1.43G MACs

- Inception modules: parallel filters to process images at different scales (with final 1x1 convolution to reduce the number of weights)

ResNet-50 (also 34, 152 and 1202)

- 49 CONV + 1 FC
- 25.5M weights
- 3.9G MACs
- ShortCut module: helps addessing vanishing gradients by letting the network learn residuals $F(x) = H(x) - x$

As it can be seen, the increase in top-5-error accuracy is correlated to the increase of the number of CONV layers, meaning that good feature extraction is crucial. However, CONV requires far more MACs than FC, so they need to be optimized in order to reduce inference cost.

2.7. Image Classification Datasets

Given an image, the goal is to select 1 out of N classes, without localization.

MNIST Toy dataset for digit classification (current SOTA: 0.21% error rate).

ImageNet Dataset for object classification:

- 256×256 color images;
- 1000 classes;
- 1.3M training + 100000 validation + test samples.

2.8. Deep Learning Frameworks

- TensorFlow (Google)
- PyTorch (Facebook)

They compute automatically gradients, and can efficiently run on GPUs.

3. Computational Transforms

3.1. Kernel computation

Fully-Connected Layer

- Matrix-Vector multiply: all inputs in all channels are multiplied by a weight and summed together, as below

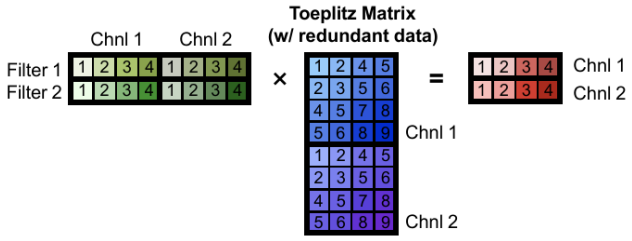
$$(M \times CHW) \cdot (CHW \times 1) = M \times 1$$

- Matrix-Matrix multply: extra dimension N due to batching, as below

$$(M \times CHW) \cdot (CHW \times N) = M \times N$$

- Implemented as **General Matrix Multiplication** (*GeMM*):
  - CPU: OpenBLAS, Intel MKL, etc.
  - GPU: cuBLAS, cuDNN, etc.
- Optimized by tiling to storage hierarchy.

Convolutional Layer Convolutions cannot be implemented with a sliding window approach due to its inefficiency (it cannot make use of CPU cache and locality of reference). Thus, it is usually converted into matrix multiplication using the Toeplitz Matrix: the filter is unrolled into a row vector, while the patches of input fmap are unrolled and transformed into columns, that are horizontally stacked together. As a result, convolution can be computed with GeMM (whose output fmap will be a row vector), but as a downside data is replicated (redundancy and memory bandwidth problems). In the multichannel case, the unrolled channels are vertically stacked together.

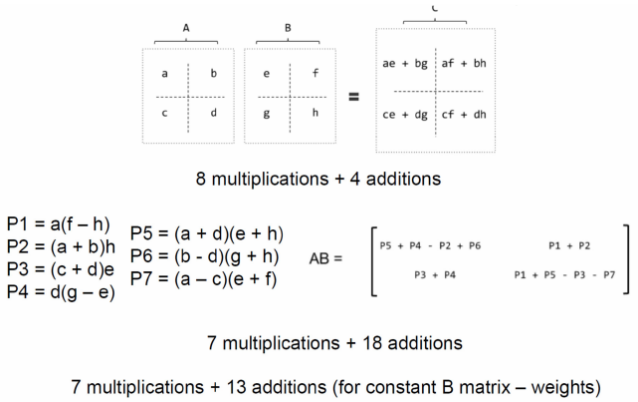


3.1. Computation transfomations

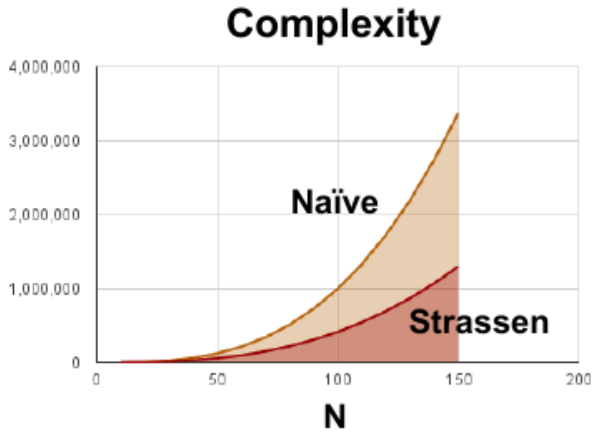
- Goal: achieve the same result but with less operations.
- Focuses mostly on computation.

Strassen

- In case of 2×2 matrices, the standard multiplication requires 8 multiplications and 4 additions.
- Strassen transforms requires 7 multiplications and 18 additions instead.



- CPUs are faster performing additions than multiplications, thus Strassen leads to a speed-up.
- In general, Strassen reduces the complexity of matrix multiplication from $\mathcal{O}(N^3)$ to $\mathcal{O}(N^{2.807})$, at the price of reduced numerical stability and increased memory footprint (you have to precompute and store intermediate data).



Winograd

- It target convolutions instead of matrix multiply.
- **1D - F(2, 3)** case:
  - output vector with size 2;
  - filter vector with size 3:
  - it scales well when applied to matrix with more dimensions.

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix}$$

(input: d_0, d_1, d_2, d_3; filter: g_0, g_1, g_2)

6 multiplications + 4 additions

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

4 multiplications + 12 additions + 2 shifts

4 multiplications + 8 additions (for constant weights)

- **2D - F**$(2 \times 2, 3 \times 3)$ case:
  - output matrix with size 2×2;
  - filter matrix with size 3×3;
  - obtained by nesting 1D Winograd transforms.

Filter

g_{00}	g_{01}	g_{02}
g_{10}	g_{11}	g_{12}
g_{20}	g_{21}	g_{22}

$*$ Input Fmap

d_{00}	d_{01}	d_{02}	d_{03}
d_{10}	d_{11}	d_{12}	d_{13}
d_{20}	d_{21}	d_{22}	d_{23}
d_{30}	d_{31}	d_{32}	d_{33}

$=$ Output Fmap

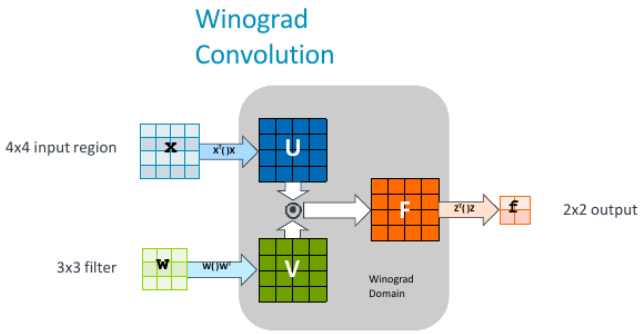
y_{00}	y_{01}
y_{10}	y_{11}

Original: 36 multiplications

Winograd: 16 multiplications → 2.25 times reduction

- It works on a small region of output at a time, and therefore it uses inputs repeatedly (Winograd halos), making it less memory-hungry than Strassen.
- Input data is transformed into the **Winograd domain**, **element-wise product** is performed (instead of convolution), and then the result is transformed back into the standard domain:

$$f = \mathbf{Z}^T[(\mathbf{W}w\mathbf{W}^T) \odot (\mathbf{X}x\mathbf{X}^T)]\mathbf{Z}$$



$$f = Z^T \left[(WwW^T) \odot (X^T xX) \right] Z$$

Input region transform: Input region x is transformed into a congruent matrix U in Winograd domain by means of a matrix $\mathbf{X}$, which is filled only with 1, 0 and -1, resulting in fast computations (either identity, zeroing or sign flipping).

Filter transform: Filter w is transformed into a congruent matrix V (which has the same size as U) in Winograd domain by means of a matrix $\mathbf{W}$, which is filled only with 1, 0, 1/2, -1/2, -1, resulting again in fast computations (1/2 can be implemented with a shift).

Output channel transform: Output F is transformed into a congruent matrix f in standard domain by means of a matrix $\mathbf{Z}$, which is filled only with 1, 0 and -1, resulting again in fast computations.

In particular, for input region of size 4×4, filter of size 3×3 and output of size 2×2 we have:

- 16 element-wise multiplications instead of 36 MACs for standard convolutions (36/16=2.25x reduction), making it effective also on small matrices (as opposed to Strassen, which is effective on larger matrices);
- 24 additions and 36 multiplications to transform filters, done only once at the beginning (offline cost);
- 32 additions to transform input, done only once for each inference pass (one-time cost, highly reusable);
- 24 additions to back-transform output, computed for several output channels at the same time.

Winograd's performance vaires according to convolution layer's dimensions: in fact, in cuDNN there's the possibility to choose Winograd depending on the tensor's size and shape. Transformation cost is ammortized only in case of data reuse.

Fast Fourier Transform Similar to Winograd: filters and input fmaps are transformed into the frequency domain, in which convolution becomes a simple multiply, and then the result is anti-transformed to the standard domain.

It reduces complexity from quadratic ($\mathbf{O}(N_{o_2} N_{f_2})$) to log-linear ($\mathbf{O}(N_{o_2} \log_2 N_o)$).

The transformation, which requires a multiplication by twiddle factors, is more expensive than Winograd; in fact FFT's computational benefits decreases with decreasing filter's size (not heavily used for this reason).

It also requires more memory bandwidth to store pre-computed frequency-domain filters (which have the same dimension of input, but they're complex).

4. Reducing Inference Cost

Two approaches:

- **Reduce the size** of operands for storage/computation:
  - from floating point to fixed point data;
  - bit-width reduction;
  - non-linear quantization.
- **Reduce the number** of operations for storage/computation:
  - exploit activation statistics (compression);
  - network pruning;
  - compact network architectures.

4.1. Reduce size of operands

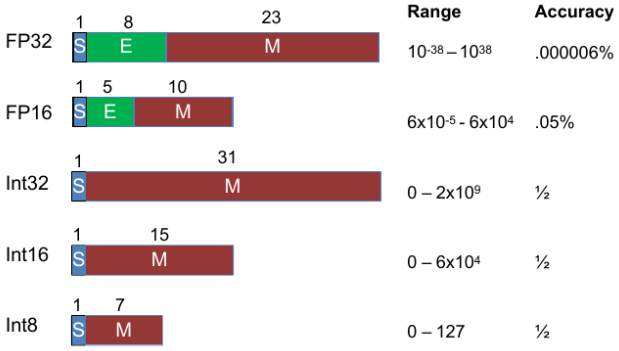
Precision: refers to the number of levels (*i.e.* values); the number of bits necessary to represent a certain number of levels is $N_{\text{bits}} = \log_2(\#\text{ levels})$

Quantization: mapping data to a smaller set of levels; it can be *linear* (*e.g.* **fixed-point**) or *non-linear* (*e.g.* computed, or via table lookup).

The objective of such strategies is to reduce the size of operands in order to increase speed and/or reduce energy cost while preserving accuracy.

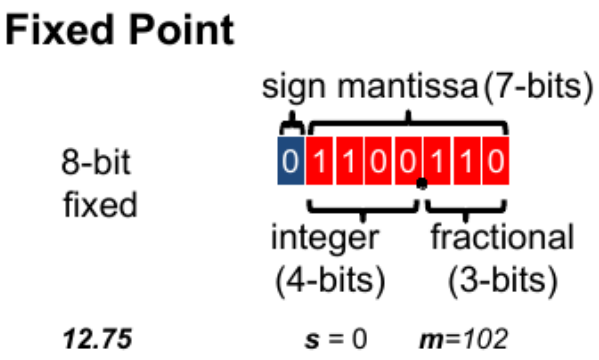
The energy cost is really high when communicating with memory (CPU and memory are quite far from each other, even if they're into the same chip), so working with smaller number helps reducing this cost (if data is small enough, it can be stored directly on the chip).

Number representation Despite using the same number of bits, FP32 are much more flexible than Int32: in fact, having an 8-bits-exponent, the former can represent $2^{2^8} \sim 10^{77}$ numbers in the range of $[10^{-38}, 10^{38}]$; on the other hand, the latter can only represent $2^{31} \sim 2 \cdot 10^9$.

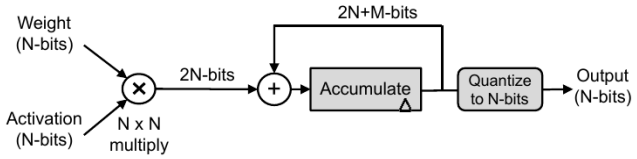


Similarly, FP16, having a 5-bit-exponent, can represent $2^{2^5} \sim 10^9$ numbers in the range of $[10^{-5}, 10^4]$, whereas Int16 can only represent $2^{31} \sim 10^4$ numbers.

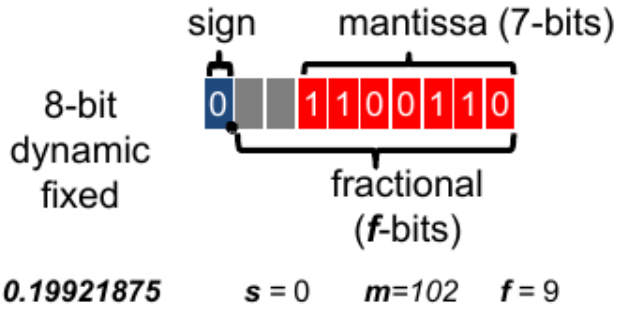
Conversely, **fixed point** numbers do not have an exponent: some bits are reserved for the integer part, while the others are reserved for the fractional part.



In neural networks, the most common operation is multiply-accumulate (MAC). Weights and activations (both represented with N bits) are multiplied: the result, in the worst case, will require N^2 bits to be represented (due to multiplication being a shift). Then, such result gets accumulated in an accumulation register: at each accumulation, since we sum two N-bit-numbers, the size increases at most by 1 bit. Therefore, if the accumulation is repeated M times, the result will require $2N + M$ bits of space in the accumulation register (M is based on the largest filter size). Finally, the result is quantized back into N bits, since such output will be the input of the following layer.



Since data's order of magnitude may vary between layers, we could use **dynamic fixed point** numbers, in which the number of bits f reserved for the fractional part varies based on data type and layer.



Impact on accuracy More bits means more precision, but also more expensive computation.

In particular, dynamic fixed point preserve almost entirely the original accuracy.

Dynamic fixed point can be avoided by centering dynamic range in each layer, which can be done with batch normalization.

Hardware accelerators **Nvidia Pascal**: unprecedented training performance with both 16-bit floating point and 8-bit integer instructions.

Google's TPU: focused on delivering data quickly by cutting down on precision, it relies on 8-bit integer rather than on floating points (supported only from v2).

Microsoft Brainwave: provides custom 8-bit floating point format.

Precision varies from layer to layer, so we can choose the number of quantization bits according to the specific layer's precision, without affecting too much the network's accuracy.

Tolerance	Bits per layer (I+F)
AlexNet (F=0)	
1%	10-8-8-8-8-8-6-4
2%	10-8-8-8-8-8-5-4
5%	10-8-8-8-7-7-5-3
10%	9-8-8-8-7-7-5-3

In bit-serial hardware, which computes additions and multiplications one bit at a time, reducing the bitwidth speeds-up computations by skipping CPU cycles.

Binary Networks

- **Binary Connect** (BC):
  - weights $= \{-1, 1\}$;
  - 32-bit float activations;
  - MAC reduces to addition/subtraction;
  - 19% accuracy loss on AlexNet.
- **Binarized Neural Network** (BNN):
  - weights $= \{-1, 1\}$;
  - activations $= \{-1, 1\}$;
  - MAC reduces to XNOR;
  - 29.8% accuracy loss on AlexNet.
- **Binary Weigth Nets** (BWN):
  - weights $= \{-\alpha, \alpha\}$, except first and last layers (32-bit float);
  - 32-bit float activations;
  - α determined by the L_1-norm of all weights in a filter;
  - 0.8% accuracy loss on AlexNet.
- **XNOR-Net**:
  - weights $= \{-\alpha, \alpha\}$;
  - activations $= \{-\beta_i, \beta_i\}$, except first and last layers (32-bit float);
  - β_i determined by the L_1-norm of all activations across channels for a given position i of the input fmap;
  - 11% accuracy loss on AlexNet.

Scale factors (α, β_i) can change according to filter/position in filter; they can occupy more bits, so the computation is more expensive w.r.t. BC.

In the case of XNOR-Nets, hardware needs to support both activation precisions.

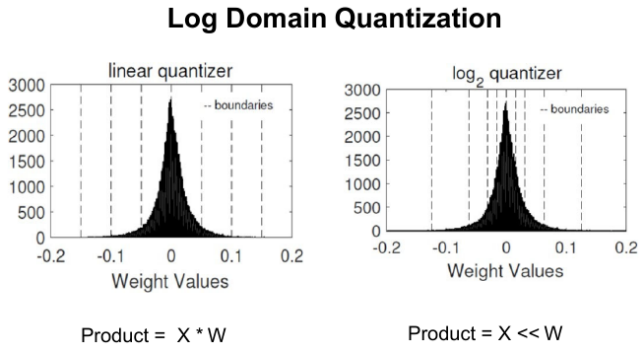
Ternary Networks They allow weights to be zero, increasing sparsity but also the number of bits (2-bits).

- **Ternary Weights Nets** (TWN):

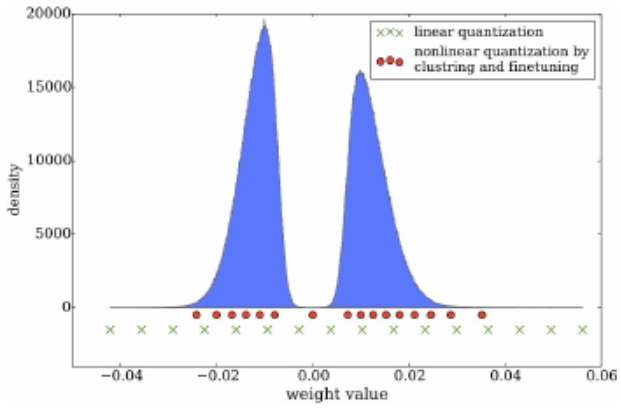
  - weights $= \{-w, 0, w\}$, except first and last layers (32-bit float);
  - 32-bit float activation;
  - 3.7% accuracy loss on AlexNet.
- **Trained Ternary Quantization** (TTQ):
  - weights $= \{-w_1, 0, w_2\}$, except first and last layers (32-bit float);
  - 32-bit float activation;
  - 0.6% accuracy loss on AlexNet.

Computed Non-linear Quantization

- In linear quantization, floats are evenly spaced resulting in uniform (low) precision; moreover, the product is computed as a standard multiplication.
- In $\log_2$ quantization, floats are more dense near zero and more sparse far from it, resulting in a higher precision for small values (which is more common, since the mean of weights is zero); moreover, the product in log-domain becomes a cheap shift operation.



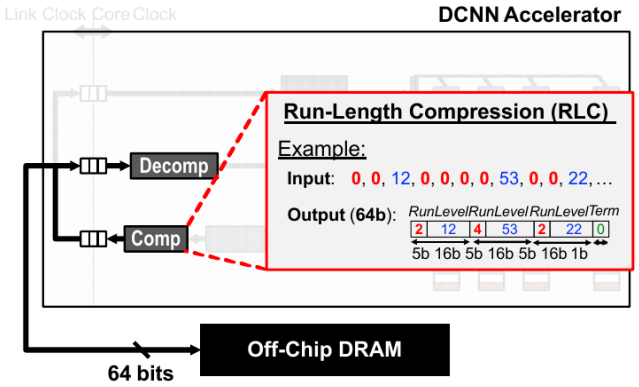
Learned Mapping Instead of having a fixed rule for quantization, the rule is learned by training (*e.g.* with K-means). Implemented with a look-up table.



4.2. Reduce number of operations

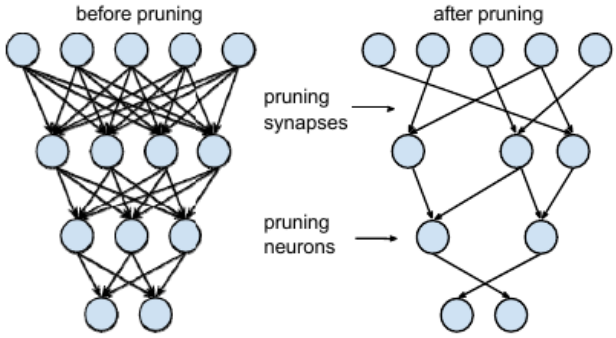
- Exploit activation statistics.
- Exploit weight statistics.
- Exploit dot product computation.
- Decomposed trained filters.
- Knowledge distillation.

Compression After ReLU, there are many zeros in output fmaps: in Eyeriss architecture, this fact is exploited to compress data (**Run-Length Compression**, RLC), which reduces DRAM bandwidth.



Moreover, when image data is zero, Eyeriss skips MAC and memory reads, reducing PE power by 45% (**Data Gating/Zero Skipping**).

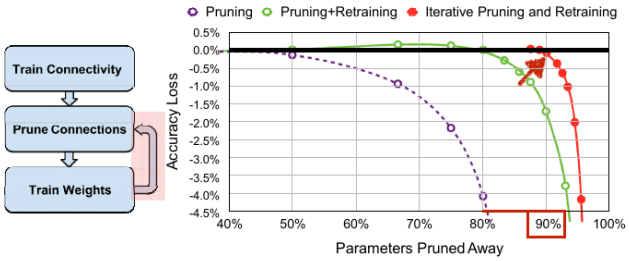
Pruning Pruning makes weights sparse; it is more effective on fully connected layers, and in AlexNet it leads to a 9x weight reduction and 3x MAC reduction.



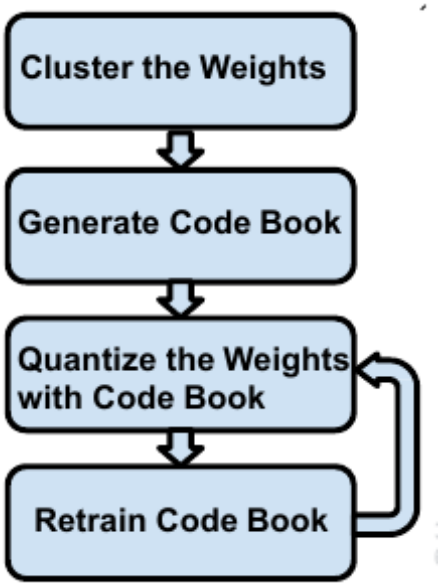
It is based on weights' magnitude: if a weight is smaller than a given threshold, it get pruned.

When pruning more than 80% parameters, there is a significant loss in accuracy;

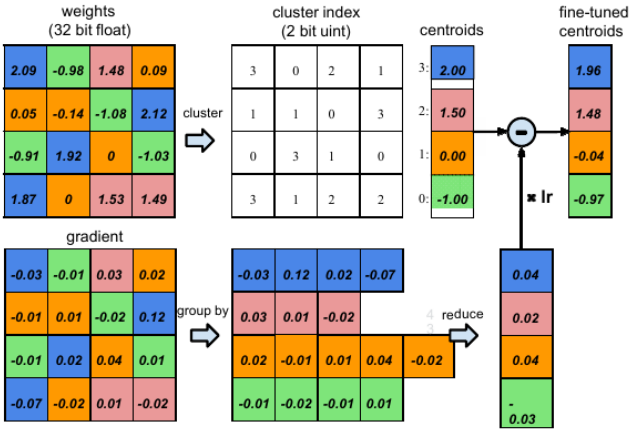
however, accuracy can be recovered by retraining the network. By iteratively prune and retrain the network, we can prune even 90% of parameters while mantaining the original accuracy.



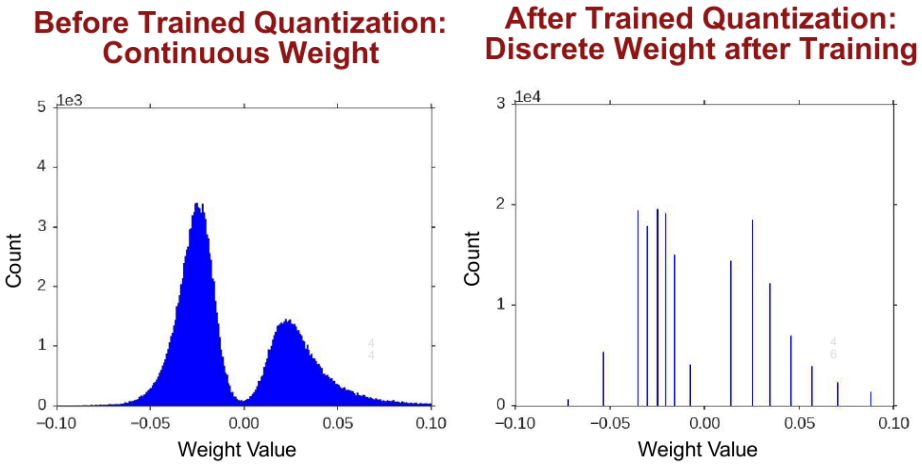
Trained quantization The weights are clustered s.t. all weights in a cluster have a similar value, and then a **code book** (*i.e.* a look-up table) is generated: for each cluster, the centroid is computed and assigned to all the weights in such cluster.



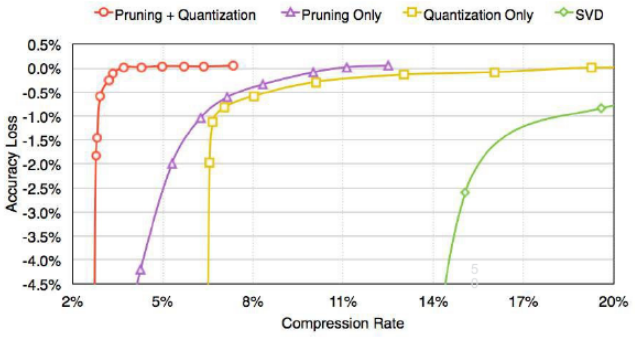
The code book is iteratively retrained in order to reduce the error due to the quantization.



After applying trained quantization, the weights shift from a "continuous" domain to a discretized one, allowing to use less bits.



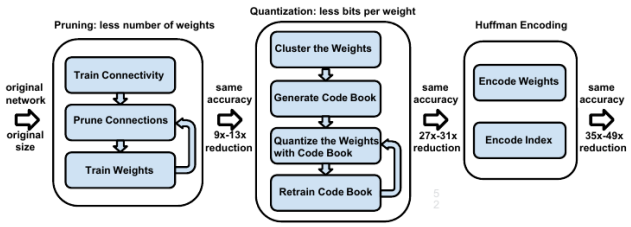
Pruning and trained quantization can work together, allowing to use far less bits without accuracy loss.



Huffman Coding A particular type of encoding that uses:

- more bits to represent in-frequent weights;
- less bits to represent frequent weights.

All these techniques can be combined, obtaining a significant bitwidth reduction (50x on VGGNet, 10x on ResNet) without accuracy loss.

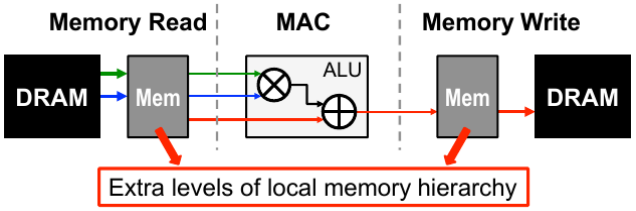


5. CNN Workload

The key operation is multiply-accumulate: input fmap is multiplied by weights element-wise, and the results are accumulated; then, a non-linearity is applied.

Weights and input fmap are read from memory, and output fmap is written to memory. If all memory R/W are **DRAM** access, memory becomes a **bottleneck** (DRAM access is 100-1000x less energy-efficient than on-chip access).

To contrast this, extra levels of local memory, internal to the chip, are added.



The data is copied (*i.e.* cached) from DRAM to chip local memory in order to reduce (up to 500x on AlexNet CONV layers) the overhead due to the read (effective only in case of **data reuse**).

Then, the partial sum gets accumulated directly on-chip, avoiding expensive access to DRAM, and saved to DRAM only once after the non-linearity.

5.1. Types of data reuse

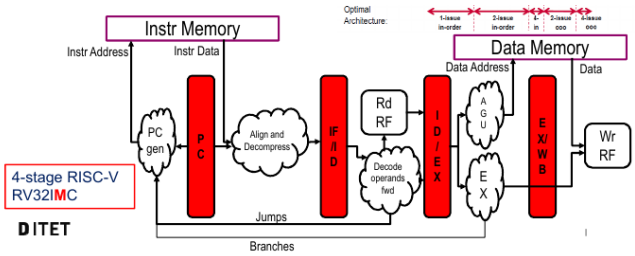
- **Convolutional Reuse** (CONV): reuse *activations* and *filter weights* during the sliding window process.
- **Fmap Reuse** (CONV, FC): reuse *activations* when applying different filters to the same activation to produce different output channels.

- **Filter Reuse** (CONV, FC with batch size > 1): reuse *filter weights* when applying the same filter on different batches.

5.2. Processors for CNNs

We can design a simple ad-hoc, pipelined (*i.e.* they execute instructions in several steps) processor to handle more efficiently MAC operations:

1. Load instruction from memory (**Fetch**).
2. Decode instruction.
3. Execute instruction using as operand the internal register.
4. Store result:
   - in the internal register, updating the previous result;
   - in memory, from which the result is read again and then stored in the register (**Load/Store**).



Increasing the performance of our processor by adding features will non-linearly increase the energy consumption. The most efficient processor is the simplest one.

Energy is defined as the product of power and time: $E = P \cdot t$; thus, to increase energy efficiency we need to minimize both power and time. The best way to solve the problem of performance is not to make a huge processor, but to use multiple simple processors and make them run in parallel.

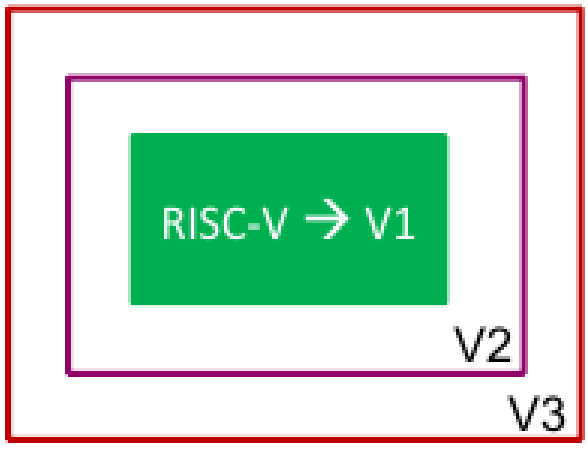
Memory is always critical: we need to make sure that memory access is done as much as possible on local memory (more affordable), and that we use as much as possible the internal memory of the processor, the register file (more efficient).

5.3. ISA Extensions

Given an open-source RISC-V processor as baseline, we can add instructions (ISA extensions) to the default instruction set in order to execute NN-related work in the most efficient way:

- **V1**: baseline RISC-V RV32IMC with:
  - Add
  - Multiply
  - Increment
  - Load/Store

  - Jump
  - Compare
- **V2**:
  - HW loops
  - Post modified Load/Store
  - Mac
- **V3**:
  - SIMD 2/4 + DotProduct + Shuffling
  - Bit manipulation unit
  - Lightweight fixed point (ML centric)



Such instructions introduce small power and area overhead (when I add more hardware, in order to keep energy consumption I have to make sure that the extra-power consumption is lower than the increase in performance).

Hardware Loops Given a for-loop as the following:

```
c = 0;
for (i = 0; i < 100; i++) {
    c = c + a[i] * b[i];
}
```

in the original RISC-V implementation we need to check the exit condition at each iteration:

```
// Initialize counter
mv x4, 100
// Initialize accumulator
mv x5, 0
Lstart:
    // Decrement counter
```

```
    addi x4, x4, -1
    // Load elements from memory
    lw x8, 0(x9)
    lw x10, 0(x11)
    // Update memory pointers
    add x9, x9, 4
    add x11, x11, 4
    // Perform MAC
    mul x8, x8, x10
    add x5, x5, x8
bne x4, x0, Lstart
```

When the number of iterations is known in advance, we can apply the **hardware loop** extension, which lets us skip the exit condition check and thus avoid counter and branch overhead:

```
// Initialize accumulator
mv x5, 0
// Set number of iterations, start and end of the loop
lp.setupi 100, Lend:
    // Load elements from memory
    lw x8, 0(x9)
    lw x10, 0(x11)
    // Update memory pointers
    add x9, x9, 4
    add x11, x11, 4
    // Perform MAC
    mul x8, x8, x10
Lend: add x5, x5, x8
```

The setup operation introduces more instructions, but these are performed only once before the loop, resulting in a 2x speed-up (whereas core area increases only by 5%).

Nested loops are implemented with two sets registers (L=0 or 1).

Post modified Load/Store Extension that lets us load elements from memory and automatically increment pointers in one single instruction instead of two.

Mac Instead of having one instruction to add and another to multiply, we can simply define a new instruction **mac** that performs both (3 input registers and 1 output register).

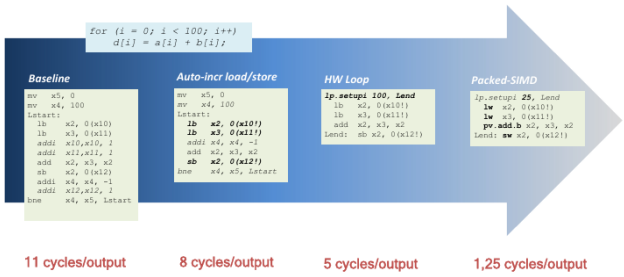
Dot Product SIMD NN inference does not need 32-bit precision: 16-bit and 8-bit suffices (with re-training). Thus, if we use 32-bit registers, it's possible to

pack together two 16-bit or four 8-bit operands together and store them in one register (**compression**), decreasing memory utilization by a factor of 2 or 4.

In case of the accumulation variable, this compression cannot be done: for example, if we sum two 8-bit numbers in the worst case we get a 16-bit number, which would overflow if stored in an 8-bit accumulation register. Therefore, accumulation is performed into a 32-bit register, and converted into an 8-bit fixed number notation only after the end of the accumulation phase.

Summary:

- Store numbers in memory in 8/16 bits.
- Load 4/2 at a time.
- Accumulate all the numbers into a 32-bit variable.
- Pack 4 accumulation results into a single 32-bit number as an input for the following layer.



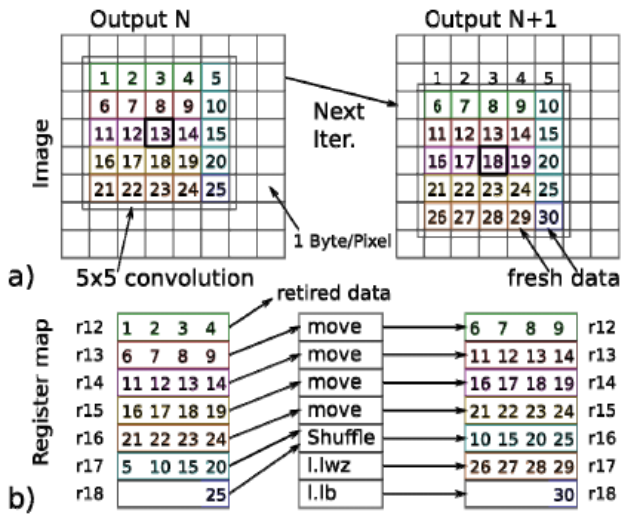
Shuffle in SIMD Given a 32-bit register containing four 8-bit numbers, Shuffle lets us:

- discard leading number;
- shifts remaining numbers to the left (thus creating a trailing hole);
- move a number from another register to fill the hole; in a single instruction.

It is fundamental to efficiently implement convolutions in SIMD version.

In case of a 5×5 convolutional filter, the following operations are performed:

- 7 sum-of-dot product;
- 4 move;
- 1 shuffle;
- 3 lw/sw;
- ~ 5 control instructions; so approximately 20 instructions per output pixel; the scalar version requires instead 100 operations per output pixel.



To perform convolution efficiently, data must be stored in memory using a particular organization; the most efficient one for matrix operations is **Height-Width-Channel** (HWC), in which the most contiguous pixels are channel pixels.

Example (3 channels)

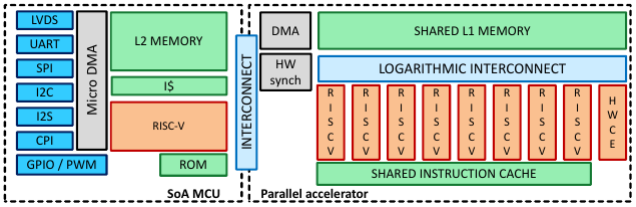
```
Address 0: h=0, w=0, c=0
Address 1: h=0, w=0, c=1
Address 2: h=0, w=0, c=2
Address 3: h=0, w=1, c=0
...
```

5.4. Parallel Ultra-Low Power (PULP)

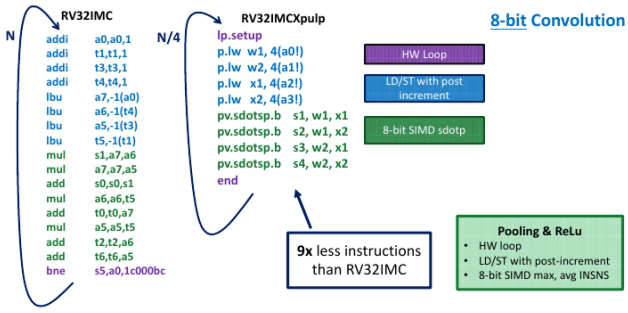
By using more processors in parallel we can increase performance while keeping energy efficiency high.

The PULP architecture is based on:

- 8 RISC-V cores with ISA extensions, each one computing an **output pixel** in parallel, since there is no data dependency (**embarassing parallel workload**);
- Shared memory, in order to allow all the cores to read filter weights and input fmaps;
- HW synchronization block to synchronize cores.

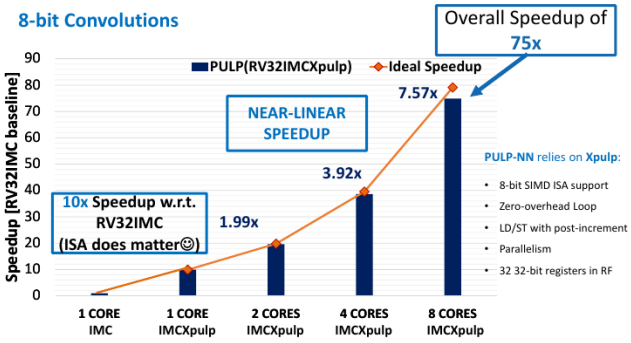


PULP exploits the ISA extensions to efficiently implement convolutions (9x less instructions than standard RISC-V):



All data is loaded from memory before performing the dot product in order to avoid stalls: in fact, load requires two cycles, so dot product would have to wait if put right after load.

Results show an almost-linear speed-up:



6. Modeling DNN Processing

6.1. Principles of Computer Design

- Take advantage of **parallelism**: *e.g.* multiple processors, disks, memory banks, etc.
- Principle of **locality**: **reuse** data and instructions as much as possible
- Focus on the **common case**: Amdahl's Law (formula which gives the theoretical speed-up in latency of a system with improved resources when

executing a task at fixed workload).

6.2. DNNs

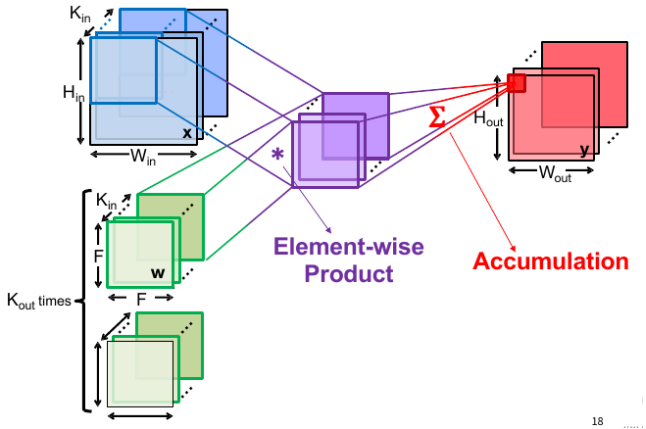
- Linear layers:
  - FC
  - CONV
- Non-Linear layers:
  - NORM
  - POOL

Since convolutions account for more than 90% of overall computation, they are our **common case**.

DNN Tensors Primer DNNs are composed by operations on N-dimensional (usually 3D) data structures (**tensors**). These operations are organized in layers, each with one (or more) input tensors and one output tensor.

Each pixel in the output tensor is computed as a function of the pixels in the input tensor (or in a subset):

$$\mathbf{y}[m,i,j] = f(\mathbf{x}[0:K_{in}, 0:H_{in}, 0:W_{in}])$$



Convolution Layer

In a CONV layer, the function is and **accumulation of convolutions**, namely a sum of element-wise products with a 4D tensor of filter weights:

$$\mathbf{y}[m,i,j] = \mathbf{b}[m] + \sum(\mathbf{w}[m, 0:K_{in}, 0:F, 0:F] \cdot \mathbf{x}[0:K_{in}, i:i+u, j:j+v])$$

Memory Access Memory access is the bottleneck: it is necessary to leverage local memory, smaller but faster and more energy-efficient, for data reuse.

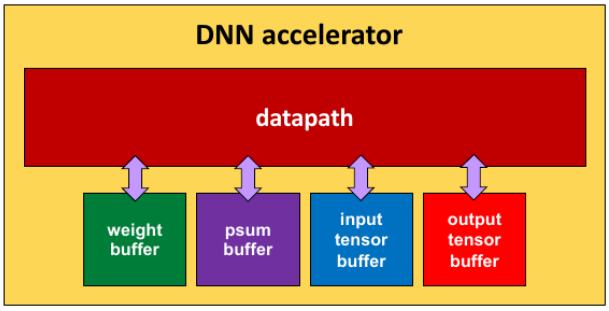
Inference and Training

- **Inference**: network is run in its forward direction; weights are fixed and used to process a single input *inferring* some information on it (*e.g.* class for classification, or a real value for regression).
- **Training - forward-prop**: similar to inference, but it operates on *batches* of inputs and saves *intermediate data* in persistent buffers.
- **Training - back-prop**: it propagates gradients of a loss function w.r.t. activation tensors and weights tensors, and uses the *intermediate results* and gradients to update weights by gradient descent (computation is vastly bigger than in forward-prop).

6.3. DNN Accelerators

Accelerator: device that makes DNNs run faster and more efficiently.

Level-0 Model Naive and ideal starting model: all data is buffered inside the accelerator.



Estimators:

- execution time and computation energy: number of **MAC operations**;
- memory dedicated to weights: number of **weights**;
- memory dedicated to activations: **size** of input and output tensors.

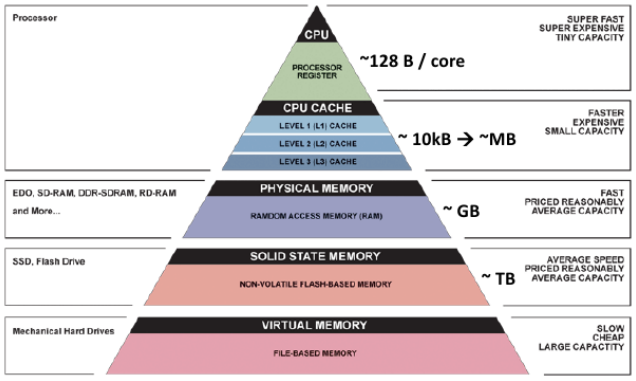
Example: VGG-16 First layer:

- $F = 3,\ K_{in} = 3,\ K_{out} = 64,\ H_{out} = W_{out} = 224$
- $\#\text{MAC} = H_{out} \cdot W_{out} \cdot F \cdot F \cdot K_{in} \cdot K_{out} = 224 \cdot 224 \cdot 3 \cdot 3 \cdot 3 \cdot 64 = 86.7\text{M}$
- $\#\text{weights} = F \cdot F \cdot K_{in} \cdot K_{out} = 3 \cdot 3 \cdot 3 \cdot 64 = 1728$
- $\text{size}_{\text{tensor}} = K_{in,out} \cdot H_{in,out} \cdot W_{in,out} =$

Level-1 Model Our L0 model is not correct, since it misses:

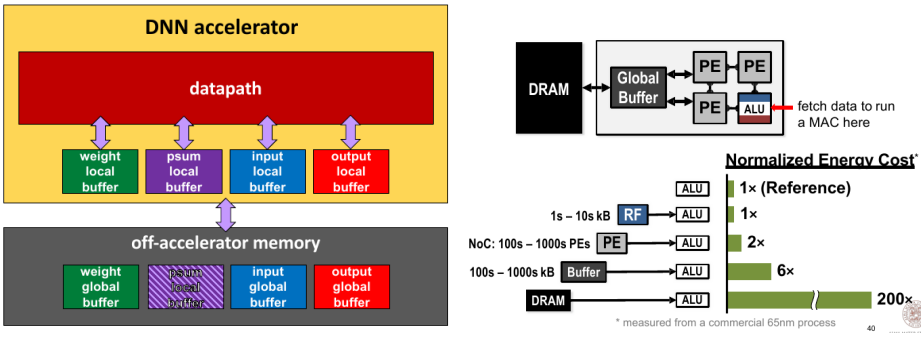
- a less naive memory hierarchy;
- a precise specification of how the accelerator **operates** (order of operations);

- a precise specification of how the accelerator **accesses data** (data layout).



Thus, we modify the L0 model such that:

- its own buffers are smaller;
- it can access global memory, with higher capacity.



Fetching data to run a MAC from buffer costs like $\sim$ 6x MACs performed in ALU. In DRAM it costs even more, as much as $\sim$ 200x.

Memory access A **dataflow** is an ordering choice, specified by means of a loop nest: for CONV layer, there are 6 nested for-loops:

```
for m in range(0, K_out):  # output channels
    for n in range(0, K_in):  # input channels
        for i in range(0, H_out):  # height
            for j in range(0, W_out):  # width
                if n == 0:
                    y[m, i, j] = 0
                psum = 0
                for ui in range(0, F):  # filter height
                    for uj in range(0, F):  # filter width
                        psum += w[m, n, ui, uj] * x[n, i + ui, j + uj]
```

```
y[m, i, j] += psum
```

Memory is flat: N-dimensional data, like tensors, have to be flattened into a certain data layout. There are many possible ways to do so:

- HWC;
- CHW;
- WHC (not used).

The specification of a certain operator, like CONV layer, does not have any specific ordering for the operands to be performed.

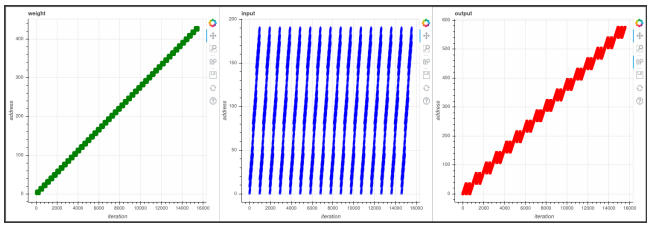
Data reuse in CNN:

- each filter weight is used $H_{out} \times W_{out}$ times as the convolutional filter goes over all input channels;
- each input channel is used K_{out} times, because each input fmap contributes to each output channel;
- each input channel pixel is also reused $F \times F$ times due to convolutional filter sliding through it (reuse happens in the spatial loop, not in the filter one);
- partial sum/output activation pixels are reused $K_{in} \times F \times F$ times (same number of MAC operations necessary to compute them).

```
for m in range(0, K_out):
  for n in range(0, K_in):
    for i in range(0, H_out):
      for j in range(0, W_out):
        psum = b[m]
        for ui in range(0, F):
          for uj in range(0, F):
            psum += w[m,n,ui,uj] * x[n,i+ui,j+uj]
        y[m,i,j] = act(psum)
```

Example With $F = 3$, $H_{out} = W_{out} = 6$, $K_{in} = 3$, $K_{out} = 16$ and C_oC_iHW data layout:

- for every output channel, each filter weight is reused every 9 iterations (F^2-sized gap);
- for every output channel, each input element is reused less frequently due to the sliding nature of convolutions:
  - only once for corner pixels;
  - onyl once in a burst of F for boundary pixels;
  - more frequently for central pixelx;
- each output element is reused for small burst of 9 iterations (accumulation phase), and then again after a huge gap.



Such sparsity is a problem, since reuse is efficient for nearby data; thus, we need a buffer (*e.g.* 32 registers) into which save data for later reuse, in order not to access again memory. The reuse of buffered data should me maximized.

So in case of our L1-model, we could exploit reuse in order to:

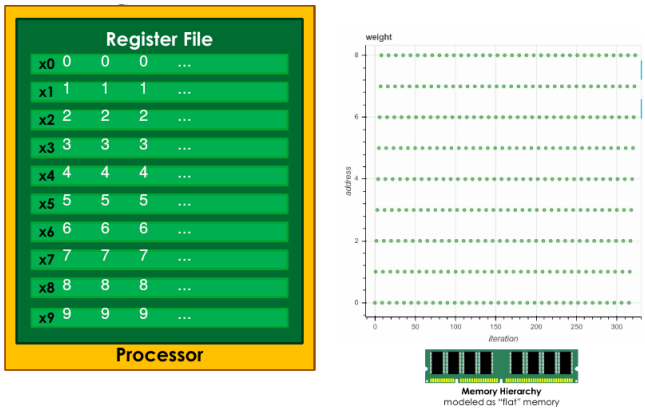
- copy data from the L1 to the L0 buffer on the first iteration;
- use it directly from L0, which is faster, on the following iterations.

6.4. Temporal Reuse

Assume that our L1 model in each cycle can only access one data element from W, X and Y tensors: an element is said to be **reused temporally** if it is accessed multiple times.

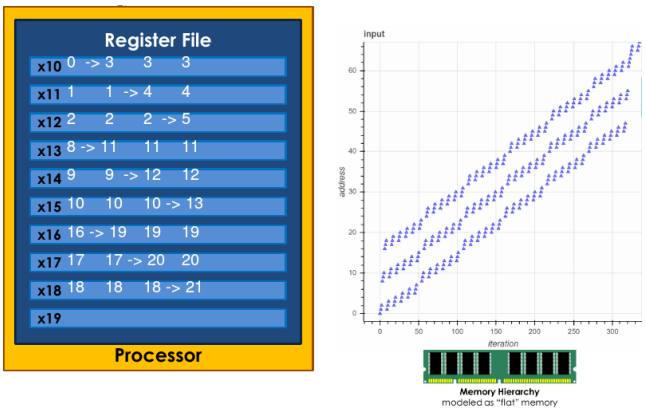
Example Given a processor with 10 fast registers, and a convolution with the same reuse as in the previous example:

- For the **weights**, it suffices to load the 9 weights of the first filter at the first iteration, and then reuse them (reuse distance is 9); after 300 iterations, we start computing over a different input channel with completely unrelated weights, so no more reuse is possible.

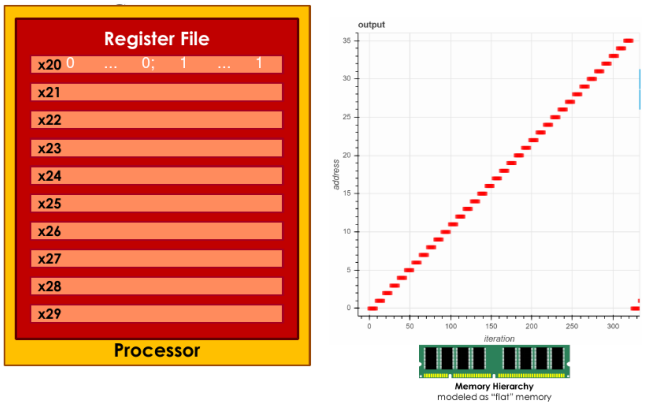


- For the **input**, we load into the register the elements that are currently traversed by the sliding window: when the filter slides we discard the eldest three element, keep six, and load three new ones; after scanning the first row, the filter starts again from the left border, but we do not exploit such

reuse (the register is too small for such ~ 50 long-distance reuse), and we have to load everything again.



- For the **output**, we load into the register each element, which is reused 9 consecutive times, and then we discard it; we do not exploit all the possible reuse: in fact, each element would be reused again after ~ 300 iterations, but such distance is too large to make its caching convenient.



If reuse distance is small, the accesses to the same data element are clustered together, and then we can cache it in a faster L0 memory.

In this case, the minimum reuse distance is 9 (for W tensor): this means that after we load the first element to be used, we have to wait 9 iterations before using it again; during those iteration we will load another different element from L1 to L0.

First experiment: if we **change data layout** from CHW and C_oC_iHW to HWC and C_oHWC_i, the data access pattern visibly changes but the data reuse pattern remains the same as before.

Second experiment: if we **change loop order** from $C_oC_iH_oW_oFF$ to

$C_o H_o W_o F F C_i$, the data access pattern visibly changes, but also the data reuse pattern changes:

- reuse distance for weights increases from 9 to ~ 27 iterations;
- reuse for inputs is still not good, each element is reused every 27 iteration;
- reuse for output is fully exploited, each element is reused 27 times with a reuse distance of 1.

This change is so significant that we use it to divide DNN accelerators in a **taxonomy** of different classes:

- **weight-stationary** accelerators are those in which the W reuse distance is small;
- **output-stationary** accelerators are those in which the Y reuse distance is small;
- **input-stationary** accelerators are those in which the X reuse distance is small.

In general, in order to minimize a reuse distance you tipically have to **trade-off** with the others.

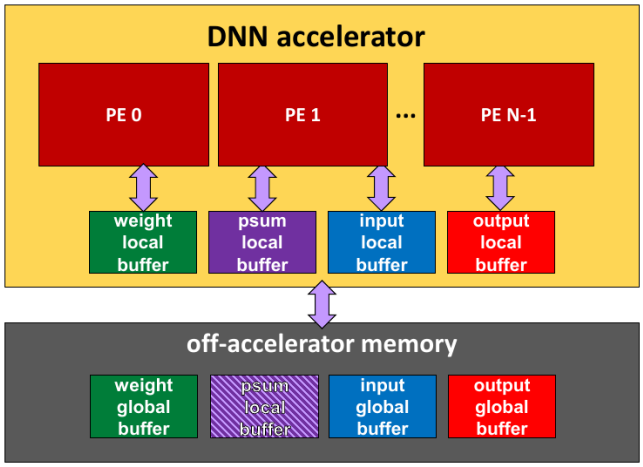
7. Parallelism and Tiling

Up until now, we analyzed the **common case** (CONV layer) and applied the principle of **locality** (**data reuse**), focusing only on **sequential execution** and on **temporal reuse**.

To get a real accelerator, we need to exploit the **parallelism** of the **loop nest** (*e.g.* the loop over input channels can be parallelized).

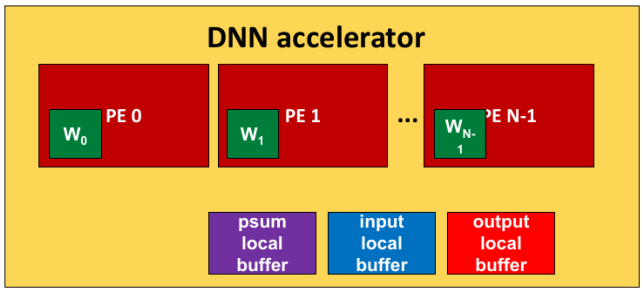
7.1. Parallel DNN Accelerator

We can modify our L1 model, providing N processing elements (**PE**), in order to enable parallelism.

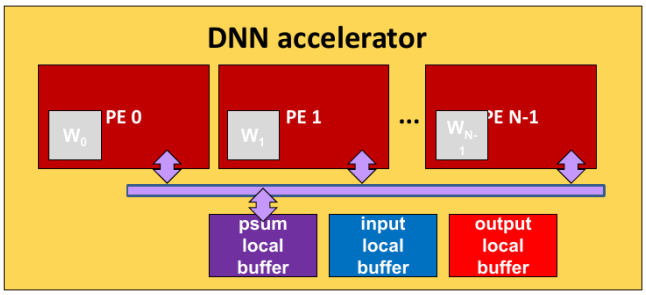


However, there is a problem concerning how to connect buffers to the processing elements.

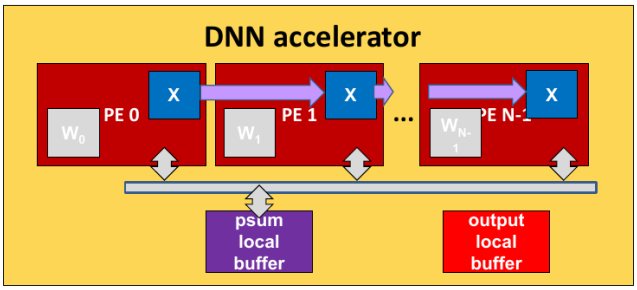
Private Data The part of the data used by only one of the PWs is said to be **not spatially reused**; such data can be thought of as **private**, internalized into each PE. However, it can still be reused in a temporal direction.



Multicasts/Reductions The part of the data used by all the parallel PEs at the same time is said to be **spatially reused**; such data can be thought of as **multicasted** from the buffer to all PEs (when loaded) or **reduced** from all PEs into a single element. In this case there may be temporal reuse as well.



Systolic Arrays When data is used by all the PEs in adjacent iterations it is said to be reused **spatio-temporally**. It can be thought of as being moved from each PE to the next. This case is typical of specialized HW accelerators called **systolic arrays**.



In practical systems it is easier to access contiguous data from memory: in fact, if the accesses are far away from each other, they are sequentialized (even if the bandwidth would have allowed parallel access); thus, we must choose a data layout such that data access becomes contiguous (**coalescence**).

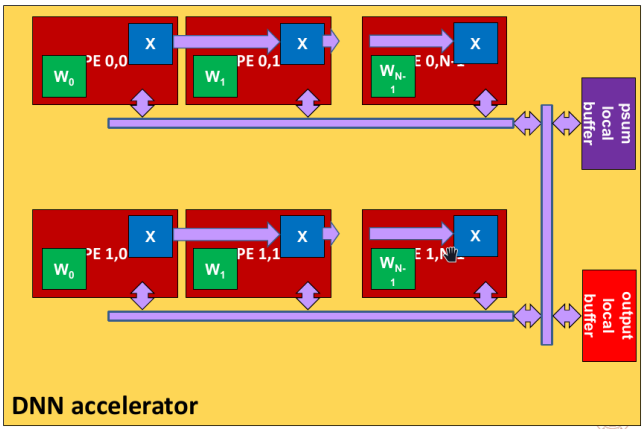
In particular, the layout HWC and C_oHWC_i, with the parallelized for over input channel as the innermost loop in the loop nest, ensures that in just one memory access each PE can load one element.

Hierarchical parallelism in DNN accelerators A real accelerator might use different techniques to parallelize each set of loops.

Example

- Outer parallel_for loop with different cores.
- Inner parallel_for loop with SIMD/vectorized instructions on the same core.

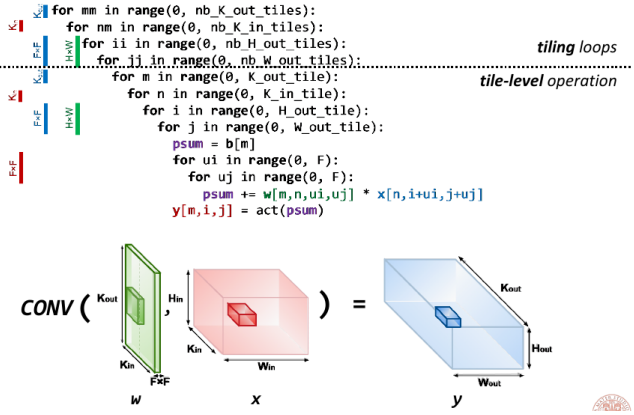
The resulting accelerator architecture can be thought of a matrix of PEs; buffers can be accessible only by PEs of one loop, or by PEs of both loops.



7.2. Data Tiling

The loop nest is a relatively simple scanning pattern over the characteristic tensors of the DNN layer (w, x, psum, y), but there may be more complicated scannings.

In particular, **data tiling** is based on the idea that sometimes it may be more convenient to concentrate on a subset of elements at a time (*e.g.* a **small volume**), instead of a full dimension.



Tipically we do not tile filters, since they're already small.

Problems:

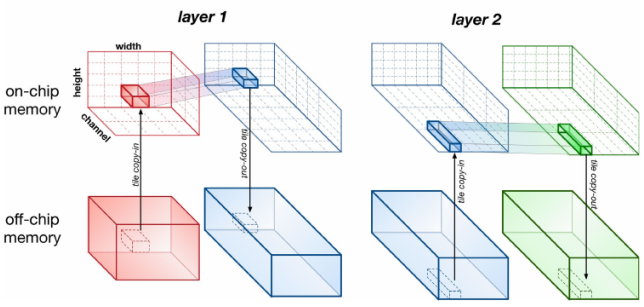
- it is not guaranteed that the volume is perfectly divisible by the tile size, so the last tile may be smaller than the other ones, and such situation must be handled with some checks;
- in case there is no padding, the output fmap size will be smaller than the input fmap's one, so the tiles must slightly overlap to avoid gaps.

Reasons for tiling:

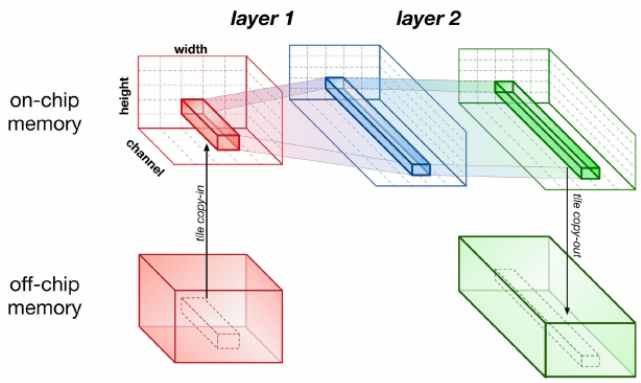
- match the **reuse distance** of a tensor to the available storage, in order to maximally **exploit data reuse** at that level;
- align the range of tile-level loops to hardware features:
  - limited **number of PEs** instead of parallelizing the full for loop (not enough PEs), with tiling we can parallelized only a small region at a time;
  - limited **scalability of interconnects**: broadcasting a lot of data may be expensive, but with tiling we can reduce the number of data broadcasted;
  - limited **cache/scratchpad size**: reuse distance could be too large to be cached into PEs' registers, but with tiling we can reduce the size of currently processed data, and as a result we also reduce reuse distance;
- hide **memory transfer costs**: when we work in a multi-layered environment, usually while executing a layer A (LA) we load the weights of the following layer B (LB) in parallel; if LA is much smaller than LB, the former may finish the execution while the latter may still be loading data, so the solution is to use tiling in order to load only the tile which will be executed first (resulting in a **pipeline of computation and data movement with balanced stages**).

Example **DORY**, a platform for cache-less microcontrollers, applies tiling in order to transfer the relevant portions of data from an external memory into the processor's registers. Moreover, it pipelines the computation of a layer with the loading of the tiles of the next layer: as a result, the execution time of the whole convolution should be dominated by the actual convolution operations, and the memory costs should be masked.

Layer-Wise execution Each layer is executed in isolation: thus, the network is executed layer-by-layer. There is an unavoidable traffic for activations, but tiling is unconstrained.

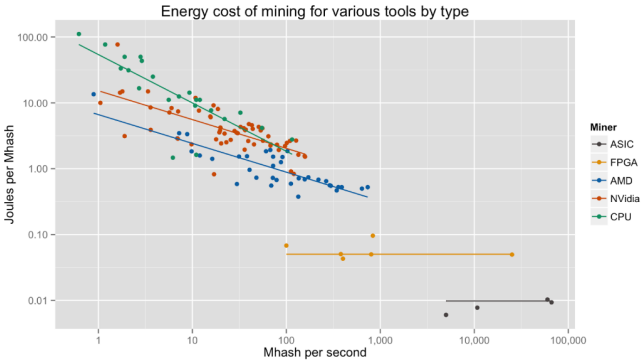


Depth-First execution The tiled tensor is propagated through all the network's layers before computing the other tensors. The traffic for activations is minimum, but tiling is now constrained.



8. Intro to GPUs

The more hardware is specialized for a certain task, the more it's efficient.



8.1. Parallelism

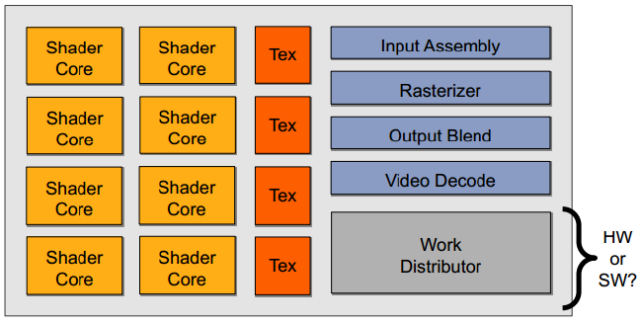
The main strategy to achieve higher efficiency is to exploit parallelism. The basic level of parallelism is **Instruction-Level Parallelism**: the processor is able to run independent instructions in parallel (*e.g.* while decoding an already fetched instruction, it loads the following one in the same cycle).

A more high-level parallelism is **Task-Level Parallelism**: two independent tasks (*e.g.* two branches of a neural network) can be executed in parallel.

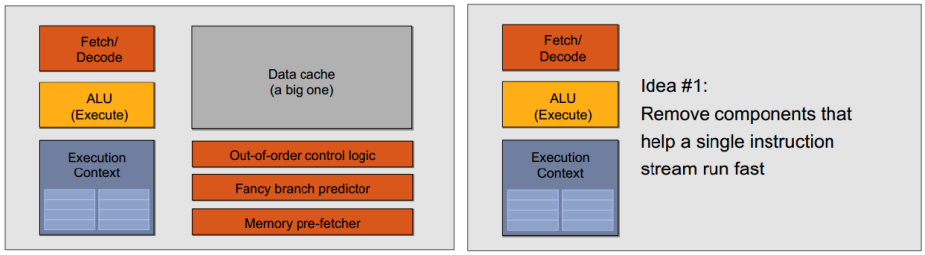
The kind of parallelism we want to exploit with DNNs and GPUs is **Data Parallelism**, namely applying the same operation to a large amount of data.

8.2. GPU

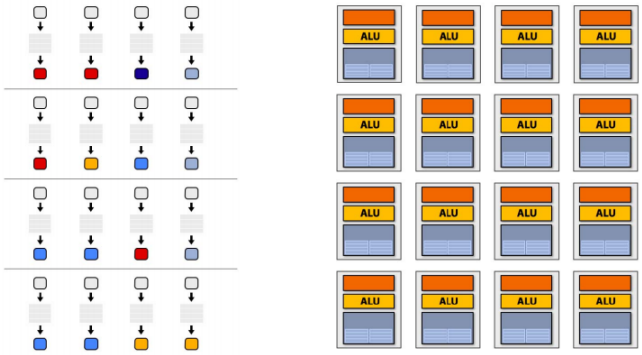
A GPU is a heterogeneous multi-processor chip, highly tuned for graphics. They were born as specialized hardware accelerators for graphics, but then became general-purpose hardware accelearators for generic computations.



CPU-style cores In order to put a lot of programmable CPU-style cores into a GPU, it is necessary to remove all the components dedicated to make a single computation faster: in fact, with a CPU we want to perform many operations at a time, whereas with a GPU we want to perform a single operation on as many data as possible. So, we remove everything but Fetch/Decode, ALU and the Execution Context.



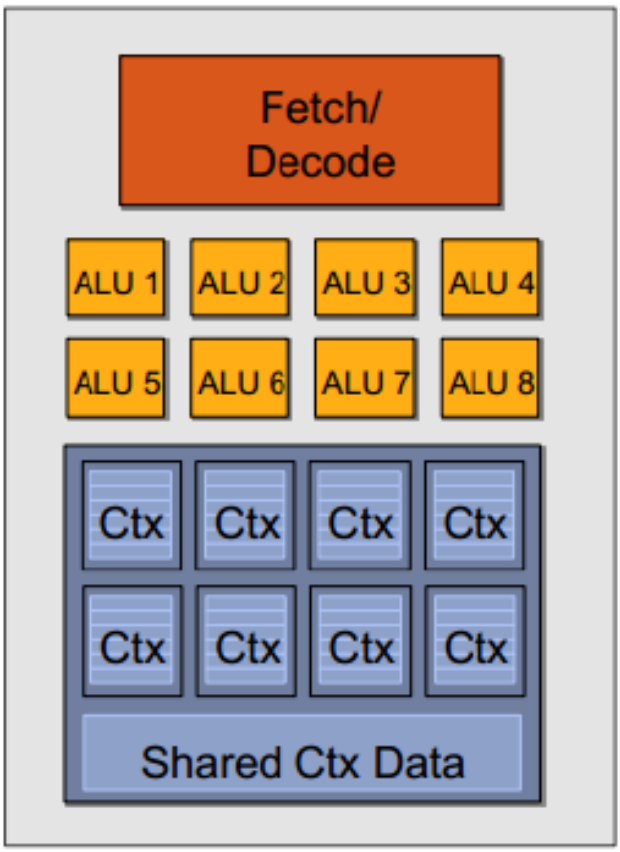
By replicating the above "slim" core 16 times, we get a 16-core GPU, capable of computing 16 simultaneous instruction streams.



From CPU-style core to Streaming Multiprocessor However, each core has its own instruction stream, but we want more cores to execute the same code (on different pieces of data); there are two solutions:

1. **Instruction stream sharing**: share the instruction stream between different cores.
2. **SIMD processing**: inside each slim core, use multiple ALUs which share the same Fetch/Decode.

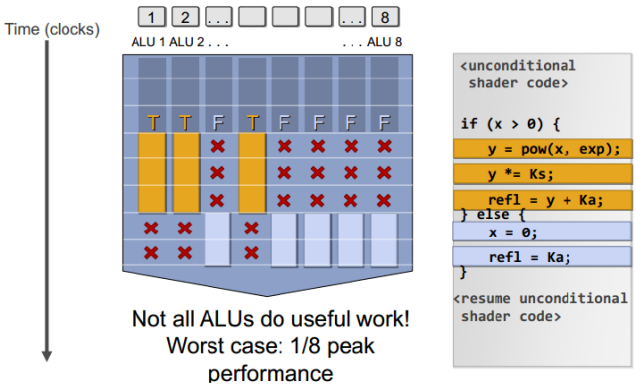
GPUs rely on the second option, which allows to ammortize the cost and complexity of managing an instruction stream across many ALUs; in NVIDIA's GPUs these units are called Streaming Multiprocessors (SM).



Multiple SMs Each SM with 8 ALUs is replicated 16 times, resulting into a 128 ALUs. These way, we have a SIMD-like processing but we also can execute multiple programs in different SMs, in parallel.

In NVIDIA's and AMD's GPUs, SIMD processing is not achieved by means of explicit SIMD instructions, but by scalar instructions which are vectorized implicitly by the hardware.

Branches Even if we have a **single instruction stream** shared between multiple ALUs, nothing prevents part of this instruction stream from being inside an if/else. In this case, as the condition depends on data itself (specific to each thread), each thread have to **execute different branches sequentially**, depending on which side of the branch is taken.



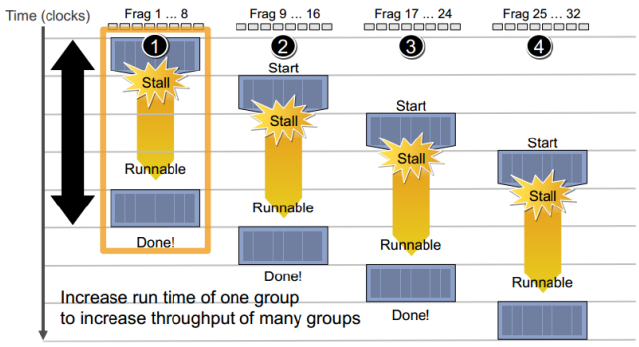
Modern GPUs have strategies to assign useful work to threads that are waiting in branches.

Stalls Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation. In fact, since GPUs tipically have small caches, when a core needs to access memory it has to communicate with the DRAM; such operation has a large latency, from 100 to 1000 cycles. Therefore, a core may be stalled until the memory operation completes.

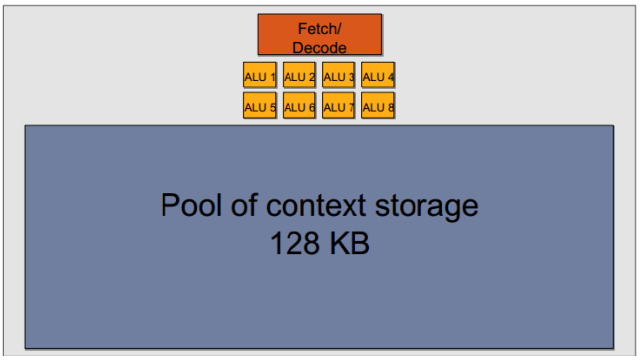
The solution is to exploit data parallelism: as we have lots of independent fragments, we can **interleave processing** of many fragments on a single core.

GPUs are **throughput-oriented**: unlike latency-oriented hardware, which try to minimize the time in which a single data is processed, they try to maximize the number of data elements processed at a time.

Therefore, when a fragment gets stalled GPUs switch to another fragment, hiding the stall.



Storing contexts The context contains the register files and the stack of each thread.



The structure of the context impacts on latency:

- small context per thread means that we could use more separate contexts overall, resulting in more latency hiding but in less space to store data;
- large context per thread means that we could have less separate contexts overall, resuling in low latency hiding but in more space to store data and increase reuse.

8.3. Data in GPUs

In CPU-style cores one of the most important (and large) parts is **data cache**, since it allows to greatly reduce latency (with a cache hit, you only pay 1/2 cycles instead of 100).

Bandwidth In GPUs we have more throughput-oriented cores, and instead of using large traditional cache we rely on **high-bandwidth** connection to memory.

Example Element-wise multiply between two long vectors A and B:

1. Load input A[i]
2. Load input B[i]
3. Load input C[i]
4. Compute A[i] * B[i] + C[i];
5. Store result into D[i].

We perform four memory operations (16 bytes) for every MUL-ADD; Radeon HD5870 can do 1600 MUL-ADDs per clock, and it needs $\sim$ 20 TB/s of bandwidth

to keep functional units busy.

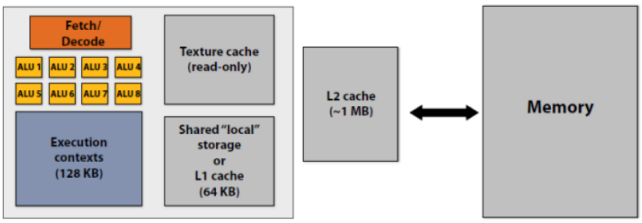
This task results in less than 1% efficiency, but the GPU is 6x faster than a CPU in performing it.

Bandwidth is limited: if processors request data at too high a rate, the memory system cannot keep up, regardless of latency hiding. Overcoming bandwidth limits are a common challenge for GPU-compute application developers.

To reduce bandiwdth requirements, there are several approaches:

- request data less often, and do more math instead (*arithmetic intensity*);
- fetch data from memory less often, and share/reuse data across fragments (**on-chip communication** or storage).

In fact, in modern GPUs there is cache-like storage: this allows to reduce latency, and thus the need of parallelism to hide it (*i.e.* we need less threads to overcome stalls).



8.4. NVIDIA Architecture

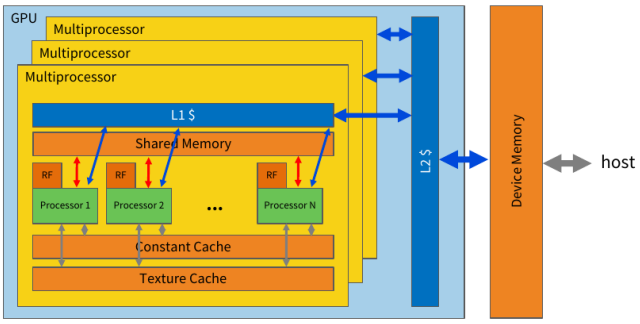
Logical/Physical mapping:

- **threads** are mapped to **processors**;
- **blocks** are mapped to **multiprocessors**.

Threads and blocks are indexed over 3 dimensions.

Components:

- instruction stream: shared inside a multiprocessor/block;
- **shared memory** (scratchpad): private to a multiprocessor/block;
- **register files**: private to a processor/thread;
- **constant/texture cache**: private to a multiprocessor/block;
- **global device memory**: shared by all multiprocessors/blocks;
- **level 1 data cache** (hundreds of kB): shared inside a multiprocessor/block;
- **level 2 data cache** (few MB): shared by all multiprocessors/blocks.



8.5. Deep Learning GPU workload

Deep Learning:

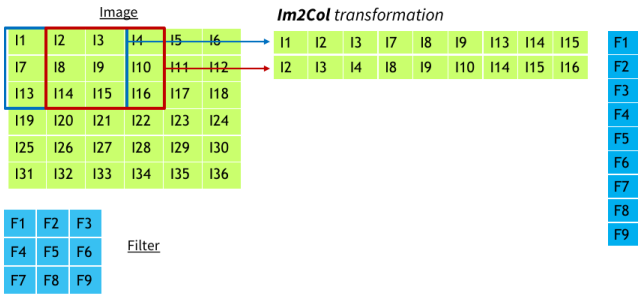
- has thousands of **independent pieces of work**:
  - uses many ALUs on many cores;
  - supports massive interleaving for latency hiding;
- is amenable to **instruction stream sharing**:
  - maps to SIMD execution well;
- is **compute-heavy**:
  - ratio of math operations to memory access is high;
  - not limited by memory bandwidth.

Inference and training In inference weights are fixed, and each result of a layer is directly consumed by the following layer, without the need to store it. In training instead we have to store intermediate results in persistent buffers. However, GPUs allow to efficiently operate on **batches** of data, whose size is totally controllable. Computation in back-prop works similarly to forward-prop, but the amount of data used is vastly bigger.

CUDA and CUDNN

- **Flexible API**: arbitrary dimension ordering, striding, and sub-regions for 4D tensors.
- **Less memory, more performance**: efficient forward and backward convolution routines with zero-memory overhead.
- **Easy integration**: black box implementation of convolution and other routines.

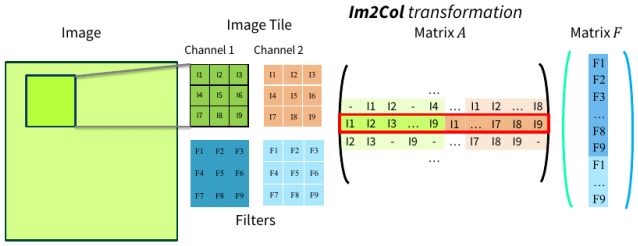
Example The classic convolution can be implemented as a general matrix-vector multiplication (GEMV) via *Im2Col* transformation:



It results in a lot of data duplication, which can be handled in two ways:

- **temporal reuse**, which implies either that the matrix is explicitly built, or that the we use a sliding window;
- the matrix is not explicitly built, it is just used as a conceptual construction.

If we consider different channels, the convolution becomes a GEMM:



Differences w.r.t. GEMV:

- longer dot product;
- more filter kernels, which must be flattened and concatenated.

Dimensions of the GEMM:

- shape of **A**: $(H_o \cdot W_o) \times (C_i \cdot F \cdot F)$
- shape of **F**: $(C_i \cdot F \cdot F) \times C_o$
- shape of output fmap: $(H_o \cdot W_o) \times C_o$

If we add another dimension N for **batches**, the dimensions of the GEMM become:

- shape of **A**: $N \times (H_o \cdot W_o) \times (C_i \cdot F \cdot F)$
- shape of **F**: $(C_i \cdot F \cdot F) \times C_o$
- shape of output fmap: $N \times (H_o \cdot W_o) \times C_o$

9. NVIDIA Volta and Ampere

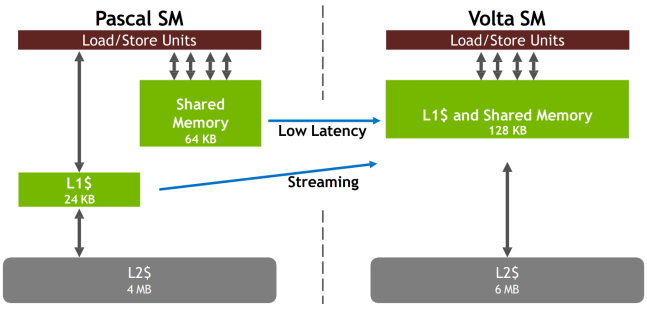
Volta (2017) and Ampere (2020) architectures explicitly target Deep Learning as their main application scenarios.

9.1. Volta V100

- 64 FP32 units
- 32 FP16 units
- 64 INT32 units
- 8 **tensor cores** (HW accelerators for DL)
- 256 kB of register file
- 128 kB of **unified L1$/shared memory**
- 2048 active threads

In Pascal SM shared memory and L1$ cache were separated, and the latter had the capability of having unlimited cache misses in flight, meaning that if you have a cache miss you can switch to a different warp as many times as you want.

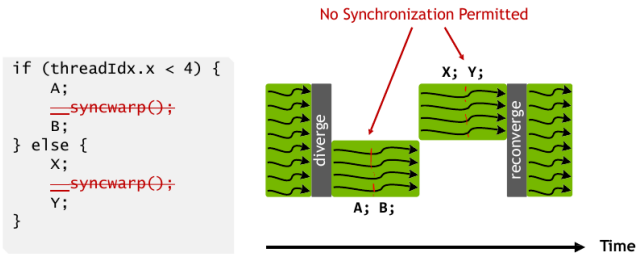
In Volta SM shared memory and L1$ cache are **unified** into a single memory, which can then be partitioned in a "shared memory" part and a "L1$" part depending on the application needs.



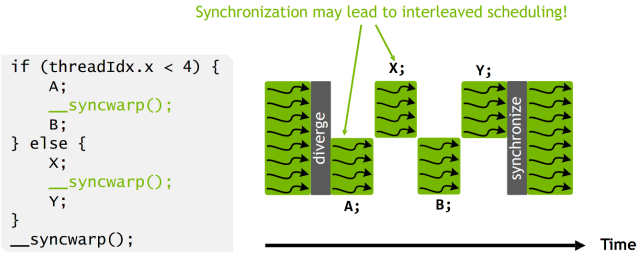
By making the cache larger and easier to utilize, V100 narrows the performance gap between shared memory and cache: in Pascal, there was a 30% gap between cache performance and shared memory performance, whereas in Volta that gap has been reduced to 7%.

Shared memory is multi-banked, meaning that each thread can access independently its own piece of data. Cache has a lower amount of banks, so if two threads try to access the same bank at the same time only one is granted, while the other have to wait one more cycle.

Warp implementation Pre-Volta architectures had the program counter (PC) and the stack (S) shared between all the threads in a warp. Therefore, synchronization was not permitted inside branches.



In Volta, each thread has its own PC and S, so there is a relaxation of the SIMD model and synchronization after the divergent part is allowed.



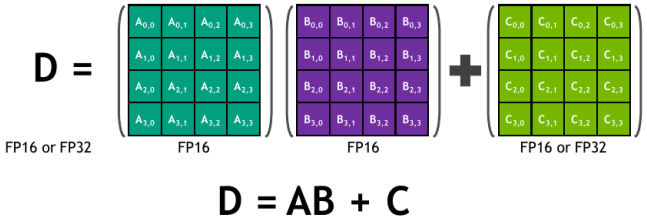
Volta Tensor Core DL uses a lot the operation of convolution and matrix multiplication (for FC layer). Convolutions can be performed in three ways:

- direct (sliding window);
- indirect (GEMM-like);
- Winograd;
- FFT.

Direct and indirect approaches can be thought of as different dataflows: the operations performed are the same, but in a different order.

Winograd and FFT require more effort but can again be computed as matrix multiplication.

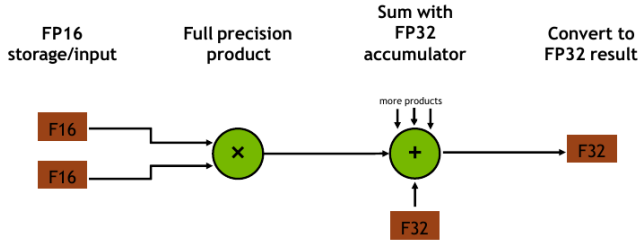
Tensor cores are modules specialized in performing 4×4 matrix multiplications, with half precision operands (16-bits); the addition operand and the output can be either half or full precision (32-bits).



A full warp can perform a 16×16 matrix multiplication by means of tiling: since tensor cores support only 4×4 matrices, they access a subset of the whole 16×16

input. When the tensor core operation is ended, there is a synchronization point after which the result is distributed across the warp: this means that from the viewpoint of each thread the whole operation is transparent.

Since the 4×4 multiplication operands are FP16, it's possible to massively simplify the hardware required for the operation while increasing the overall efficiency: in fact, the product is done in full-precision without the risk of overflow (**mixed precision**).



FP16 can also be used for training. Additionally, it also supports FP16 accumulator mode when doing inference.

Matrix multiply with mixed precision in Volta has shown a 9.3x speed-up w.r.t. Pascal.

9.2. Ampere A100

It features 108 SMs and 6912 CUDA cores; as memory speed is very important, it uses HBM2 memory with 1.56 TB/s bandwidth.

Ampere makes possible to use new and improved representations for floating point numbers:

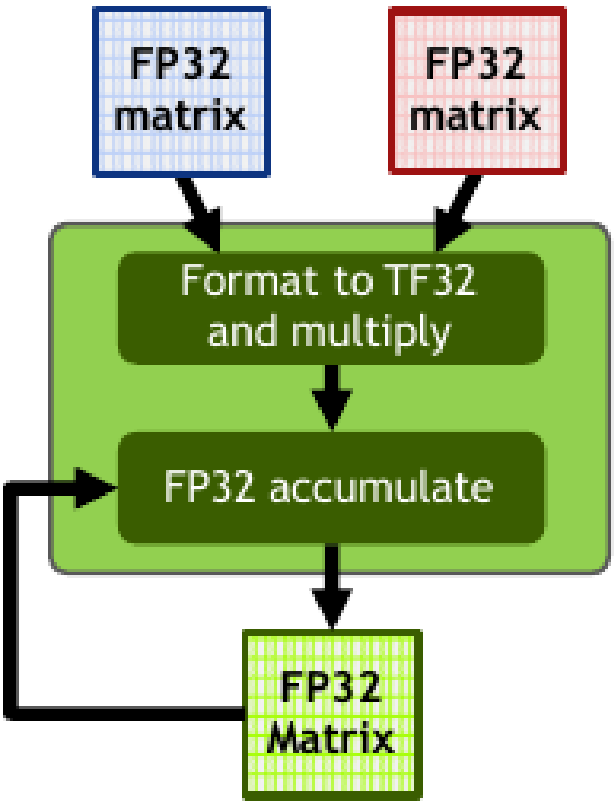
Input operands	Accumulator	Op. Exponent	Op. Mantissa	TOPS
FP32	FP32	8	23	19.5
TF32	FP32	8	10	156
FP16	FP32	5	10	312
BF16	FP32	8	7	312
FP16	FP16	5	10	312
INT8	INT32	0	8	624
INT4	INT32	0	4	1248
BINARY	INT32	0	1	4992

Brain floating point format (BF16), proposed by Google, allows to mantain the same range as FP32 while reducing the precision (in order to use only 16 bits). However, the reduced precision could lead to underflow in case of small numbers.

So a third alternative, TF32, was designed: it has an 8-bits exponent and 10-bits of mantissa. When a FP16 or BF16 number is fed in input, it gets promoted

to TF32 by padding the exponent or the mantissa, respectively. Likewise, a FP32 can get demoted to TF32 by removing least significant bits in the mantissa (reducing precision).

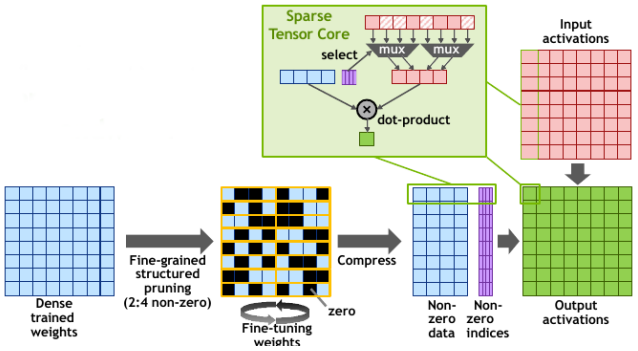
In other words, the hardware of the Ampere accelerator is built around the TF32 format (19 bits), and the other formats are handled by means of promotions and demotions.



Integer operations are also supported: they are extremely faster (with a loss in precision).

Finally, by combining multiple FP32 operations it is possible to create a double precision floating point (FP64) in the tensor core, not that useful for DL but useful for other applications (*e.g.* HPC).

A100 tensor core also exploits sparsity by skipping multiplications by zero, achieving a speed-up up to 2x: in particular, it is possible to make a matrix sparse by selecting weights with a very low value (*e.g.* below 10^{-5}) and turn them into zero (**fine-grained structured pruning**). Then, the other weights are fine-tuned in order to compensate.



Strong scaling: same workload operated faster.

Weak scaling: greater workload, same execution time (easier to achieve, because the larger the workload the easier to fed the workers with new operations).

NVIDIA claims to have achieved strong scaling.

9.3. Automatic Mixed Precision

Automatic Mixed Precision (**AMP**) combines single precision (FP32) with lower precision (*e.g.* FP16) when training a network. It achieves the same accuracy as FP32 training and uses the same hyper-parameters.

Benefits:

- accelerates math-intensive operations with specialized hardware (*i.e.* tensor cores);
- accelerates memory-intensive operations by reducing memory traffic;
- reduce memory requirements, enables training of larger models/larget mini-batches/larger inputs.

Recommendations:

- operations that can use 16-bit storage (FP16 and BF16):
  - matrix multiplication;
  - most point-wise operations (*e.g.* ReLU, TanH, add, sub, mul);
- operations that need more mantissa (FP32 and FP16):
  - adding small values to large numbers can lead to rounding errors;
  - reduction operations (sums, softmax, normalization);
- operations that need more range (FP32 and BF16):
  - point-wise operations where $|f(x)| \gg |x|$ (*e.g.* exp, log, pow);
  - loss functions.

16-bits are sometimes insufficient for weight updates, which during late stages of training can become too small for addition in FP16/BF16, and eventually get clipped to zero. Thus, it's better to keep weights in FP32 so that small updates accumulate across iterations.

The same concept applies to the loss values: to retain small gradients, during backpropagation the gradient is kept in 16-bit format but scaled by a certain factor (in order to prevent it from being too small).

Summary of AMP:

Components	TF32	FP16/BF16
Casting	Everything in FP32	Decide which operations to compute in 16-bits or FP32
Weight storage		Keep model weights in FP32
Loss scaling		Sometimes scale loss value to retain small gradients

10. DNNs at the Edge

Training and developing DNN using powerful GPUs through the cloud has disadvatages:

1. **Transmission cost**: Whatever is the input that we process, it has to be transmitted to the cloud GPU, with a cost in terms of power.

Example If we're forced to transmit something, we would like to be efficient; so, instead of sending the MNIST image it is better to send the byte representing the digit in the image (*i.e.* send directly the prediction of the model). Sending the **semantic information** instead of the raw ones is always better.

2. **Latency**: a remote server is tipically very fast, but has also a high latency; for some applications (*e.g.* robots, autonomous driving) we need low-latency devices.
3. **Privacy**: the network is not secure, so we need to use a lot of counter-actions (encryption) to be safe, which are very complex to handle.

An alternative approach, free from the above problems, is **performing AI operations directly on the device**. These devices can be very small, with limited computational resources and low power.

10.1. Processing

- **Domain transforms**: Fast Fourier Transform (FFT), Discrete Wavelet Transform (DWT), Discrete Cosine Transform (DCT).
- **Compression**: camera image compression (*e.g.* JPEG), audio compression (*e.g.* MP3).
- **Signal Filtering**: FIR filters, Kalman filters for state estimation.

- **Inference of ML models** for data understanding and machine vision:
  - *supervised*: decision trees, support vector machines (SVM), random forests;
  - *unsupervised*: k-means, principal component analysis (PCA).
- **Inference of DL models**: DNNs, RNNs.

A traditional way to solve these problems is relying on **Digital Signal Processing** (DSP), based on the following facts:

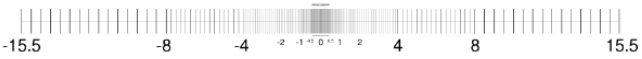
- many kernel are dominated by **linear algebra**, which translates to many loops of **multiply-add** (*e.g.* dot product);
- in many cases we use **real-valued data** within a range that is well defined at every computation step.

10.2. Representing real numbers

With floating point representation we can tweak range and precision changing the amount of bits used for exponent and mantissa:

- the exponent controls range;
- the mantissa controls precision.

A real number $r \in \mathbb{R}$, in floating point format, is represented as $r \sim s \cdot m \cdot 2^e$, where $s \in [-1, 1]$, $m, e \in \mathbb{Z}$ are respectively sign, mantissa and exponent, and represented explicitly in bits.



With fixed point representation on the other hand the exponent is implicitly known by the application:, and the range is fixed and the precision inside this range is uniform.

A real number is represented as $r \sim i \cdot 2^q$, where $i, q \in \mathbb{Z}$, and q is fixed.

Pros:

- as the precision of operands is uniform, it's trivial to predict the output precision;
- only **integers** are needed for data and operations: $r_1 \cdot r_2 = i_1 \cdot 2^{q_1} \cdot i_2 \cdot 2^{q_2} = i_1 \cdot i_2 \cdot 2^{q_1+q_2}$.

Cons:

- it's not possible to tweak the range;
- near zero the representation is not denser.

Another thing to consider with fixed-point representation is the following: suppose we have to multiply two numbers r_1, r_2 with an 8-bits mantissa; then:

$$r_1 \cdot r_2 = i_1 \cdot 2^{q_1} \cdot i_2 \cdot 2^{q_2} = i_1 \cdot i_2 \cdot 2^{q_1+q_2}$$

so the output would be 16-bits long. Thus, we can:

- keep 16 bits;
- **requantize**, modifying the precision in order to represent the 16-bit number with 8 bits; if, after the product, I want to keep as a reference exponent (**quantum**) q_1, I must right-shift the mantissa of q_2 bits (*i.e.* divide the mantissa by q_2): $i_{out} = (i_1 \cdot i_2)_{16} \gg q_2$.

Example Given:

$$x_8 = 200 \cdot 2^{-3}$$
$$y_8 = 43 \cdot 2^{-4}$$
$$z = 200 \cdot 43 \cdot 2^{-3-4} = 8600 \cdot 2^{-7}$$

as we can see z is no more representable with 8 bits, unlike x and y. If we decide to requantize keeping as exponent -3, z becomes:

$$z_8 = 8600 \cdot 2^{-4} \cdot 2^{-7} \cdot 2^4 = (8600 \gg 4) \cdot 2^{-3} = \frac{8600}{16} \cdot 2^{-3} \sim 537 \cdot 2^{-3}$$

where 537 is still not representable with 8 bits. Thus, we need to choose a greater quantum like 2^{-1}:

$$z_8 = 8600 \cdot 2^{-6} \cdot 2^{-7} \cdot 2^6 = (8600 \gg 6) \cdot 2^{-2} = \frac{8600}{64} \cdot 2^{-3} \sim 134 \cdot 2^{-3}$$

where 134 is finally representable with 8 bits.

10.3. Fixed-point and DNNs

DNN tensors tend to have a **predictable range** at inference time:

- **weights are fixed** after training, so the range is known at design time;
- **activation** tensors depend on input, but the range can be **evaluated statistically**;

At training time, **gradients** do not tipically have a predictable range, as it varies along the training procedure.

Precision to represent weights and activations:

- 16 bits $\rightarrow$ 65536 representable levels $\rightarrow$ smallest value in $[0, 1]$ range is $1.53 \cdot 10^{-5}$
- 8 bits $\rightarrow$ 256 representable levels $\rightarrow$ smallest value in $[0, 1]$ range is $3.9 \cdot 10^{-3}$
- 4 bits $\rightarrow$ 16 representable levels $\rightarrow$ smallest value in $[0, 1]$ range is $6.7 \cdot 10^{-2}$

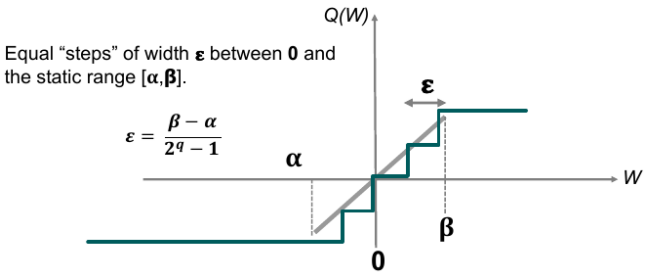
DNNs must be **retrained** and **tuned** to execute with **low bitwidth**.

Fake Quantization *Quantization* maps sets of real numbers to integer numbers. The shape of the quantization function distinguishes different QNN methods, one of which is **linear quantization**.

Given:

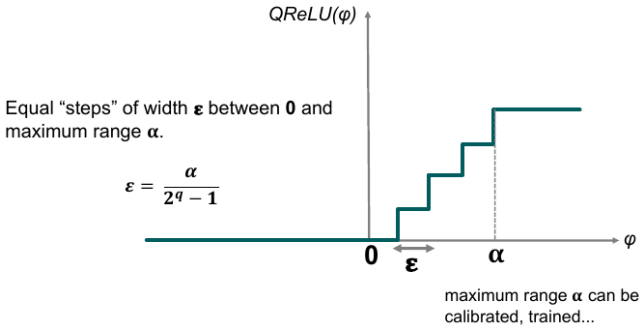
$$y = \text{ReLU}(\text{Conv}(W, x)) \Rightarrow$$
$$\phi = \text{Conv}(W, x),\ y = \text{ReLU}(\phi)$$

quantizing weight is easy, since their range is well defined. If we assume that $\forall w_i, w_i \in [\alpha, \beta]$, we start mapping each weight with an identity function. Then, according to the **quantum**, and so the precision, we can construct a step function such that each step has length $\epsilon = \frac{\beta - \alpha}{2^q - 1}$, where q is the number of bits of the exponent.



Being a piecewise constant function, such quantization function is not differentiable, and thus it **cannot be trained**. The solution, for the purpose of back-prop, is to **ignore quantization** and treat it as if it were the initial identity function (**straight-through estimator**). It is not well understood why this works in practice.

For input and output tensors, we can modify the activation functions (*e.g.* ReLU) which is already non-linear by definition. We estimate the range based on the values that we observe while processing the validation set, and we choose the boundaries of such range (in case of ReLU, 0 is the left-most boundary and α is the right-most one). Then, in the case of ReLU, we perform clipping based on α (**QReLU**). Finally, we quantize the range $[0, \alpha]$ with q bits, where each step has length $\epsilon = \frac{\alpha}{2^q - 1}$.



To back-prop non-differentiable activations, we can again use straight-through estimator by ignoring the quantization (but by mantaining clipping).

Fake quantization:

- W, x, y are still represented with FP32 numbers.
- The model is fully trainable.
- Other layers (*e.g.* Normalization) are tipically not quantized, but BatchNorm sometimes is folded inside the previous linear layers (Conv, FC).

A fake quantized network can be easily deployed to integers: in fact, it does not require knowledge of the "real-valued" world beyond its input, which can be quantized to INT32, and its output. Even BatchNorm can be translated to an integer version via affine transformation. Activations can be translated via division/right-shift and clipping.

11. PULP Architecture

Specialized Digital Signal Processors (like Motorola DSP56000) where designed with DSP in mind, introducing or enhancing concepts such as:

- **fixed-point** computation by using integers instead of float to efficiently represent real-valued data;
- **hardware loops** to remove branch overhead on inner loops when the number of iterations is known;
- **address pre/post increment** to optimize memory access to regular arrays by automatically increment an address after a load;
- **DMA** (Direct Memory Access) to decouple data movement from computation;
- **very long instruction word** (VLIW) to exploit further hardware parallelism, load multiple instructions at once, and achieve better performance.

DSPs are very niche products nowadays, but these ideas are common in modern microcontrollers (MCUs). The only difference between modern MCUs and DSPs is that instead of VLIW they use SIMD vectorization and parallel execution.

11.1 PULPissimo

It is a modern DSP-oriented MCU architecture, based on RISC-V:

- 1 core;
- 512 kB of L2 memory.

Its purpose is to collect data from sensor, **process** it **directly on the system**, and then send results.

Example Given the following loop (tipically, the innermost loop of a software kernel):

```
for (j = 0; j <= N; j++>) {
    S = S + (int)(coeff[j] * data[j]);
}
```

the standard assembly code is:

```
Lstart:
addi t5, t5, 1
mv   a5, t6
add  a6, t3, a5
addi a5, a5, 1
add  a1, t1, a5
lbu  a6, 0(a6)
lbu  a1, 0(a1)
mul  a1, a1, a6
add  a0, a0, a1
bne  a5, t4, Lstart
```

which performs 1 MAC over 10 instructions, resulting in an efficiency eff $= 1/10 = 0.1$.

By using the MAC instruction, the assembly becomes:

```
Lstart:
add   a6, t3, a5
addi  a5, a5, 1
add   a0, t1, a5
lbu   a6, 0(a6)
lbu   a0, 0(a0)
p.mac a1, a6, a0
bne   a5, t4, Lstart
```

which performs 1 MAC over 7 instructions, resulting in an efficiency eff $= 1/7 = 0.143$.

By using the post-increment load instruction, the assembly becomes:

```
Lstart:
```

```
p.lbu a6, 1(a5!)
p.lbu t1, 1(a0!)
p.mac a1, t1, a6
bne   t3, a5, Lstart
```

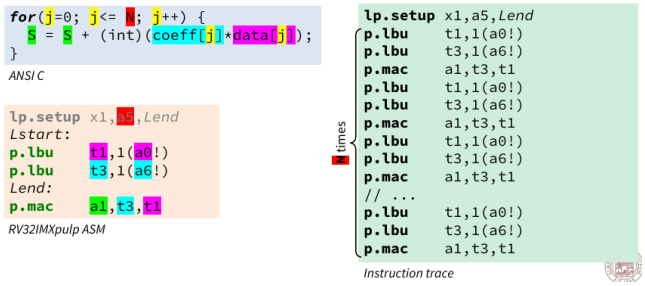
which performs 1 MAC over 4 instructions, resulting in an efficiency $\text{eff} = 1/4 = 0.25$.

By using hardware loops, the assembly becomes:

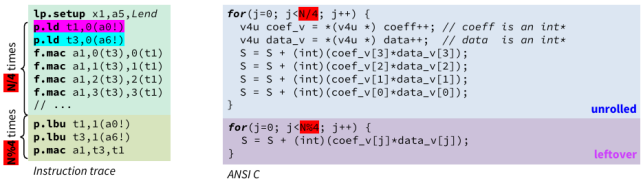
```
lp.setup x1, a5, Lend
Lstart:
p.lbu    t1, 1(a0!)
p.lbu    t3, 1(a6!)
Lend:
p.mac    a1, t3, t1
```

which performs 1 MAC over 3 instructions, resulting in an efficiency $\text{eff} = 1/3 = 0.333$.

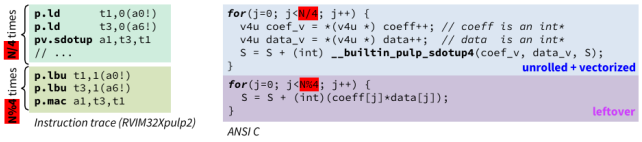
To get even more performance, we can exploit parallelism: to do so, we must unroll the loop nest.



Then, we reorder and group the unrolled instructions by 4, s.t. we can load a word of 4 elements at once: the loop is divided into an unrolled section and into a leftover section (to handle the remainder when the number of iterations is not divisible by 4).



Similarly, we want to perform MAC on 4 elements with a single vectorized SIMD instruction, `pv.sdotup`:



After all the changes (MAC instruction, post-increment load, hardware loops, loop stripping&unrolling, and SIMD vectorization), we perform 4 MAC operations with just 3 instructions, resulting in an efficiency eff $= 4/3 = 1.333$, 13x better than the original version.

11.2. From PULPissimo to PULP

PULP uses acceleration techniques to further exploit the parallelism in linear algebra and AI applications. To accelerate some tasks we could use:

- hardware accelerators, which however work only for specific applications;
- **programmable accelerators**, which can be retargeted at many applications.

A programmable accelerator will be essentially made of multiple cores similar to the ones present in the MCU to keep them simple and energy efficient.

Power and efficiency The performance of a single core is low, but using a more powerful and complex core would increase energy consumption. Thus, to increase the performance while remaining efficient we can put together multiple cores (*e.g.* 10). However, extracting performance from parallelism requires a lot of effort and attention.

Differently from GPUs (with SIMD multiprocessors), in PULP we have a cluster of processors with **MIMD**. This MIMD architecture targets a **shared memory** (not a cache) parallel programming model: 4-16 DSP cores sharing directly a shared L1 memory (**tightly coupled data memory**, TCDM). This memory is organized in **multiple banks** to allow concurrent access.

11.3. PULP execution model

It similar to OpenMP: at boot the cluster is inactive and a single thread runs on the Fabric Controller (the MCU core); the MCU core will mount the cluster and call a function on cluster core 0, and then cluster core 0 will fork an execution team on multiple cluster cores. They can be synchronized with barriers and critical sections. All cose is in L2 memory, cached for cluster threads.

Optimize computational backend Given a standard output stationary loop nest, HWC layour for activations and C_oHWC_i for weights:

```
for i in range(0, H_out):
    for j in range(0, W_out):
        for m in range(0, K_out):
```

```
            psum = 0
            for ui in range(0, F):
                for uj in range(0, F):
                    for n in range(0, K_in):
                        psum += w[m, ui, uj, n] * x[i + ui, j + uj, n]
            y[i, j, m] = act(psum)
```

in order to improve it we can:

1. Make access to x more regular by reordering in *im2col* buffer.

```
for i in range(0, H_out):
    for j in range(0, W_out):
        for m in range(0, K_out):
            for ui in range(0, F):
                for uj in range(0, F):
                    for n in range(0, K_in):
                        im2col[ui, uj, n] = x[i + ui, j + uj, n]
            psum = 0
            for ui in range(0, F):
                for uj in range(0, F):
                    for n in range(0, K_in):
                        psum += w[m, ui, uj, n] * im2col[ui, uj, n]
            y[i, j, m] = act(psum)
```

2. *im2col* and w accesses are linear and identical.

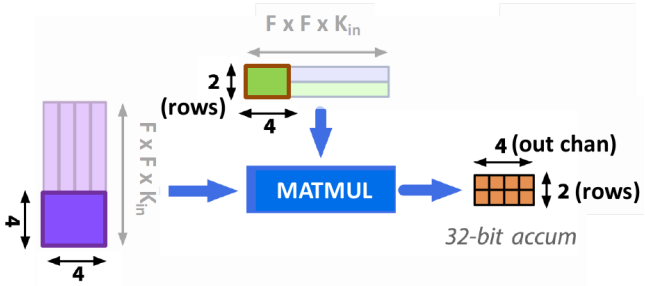
```
for i in range(0, H_out):
    for j in range(0, W_out):
        for m in range(0, K_out):
            for idx in range(0, F * F * K_in):
                for uj in range(0, F):
                    for n in range(0, K_in):
                        im2col[ui * F * K_in, uj * K_in, n] = x[i + ui, j + uj, n]
            psum = 0
            for idx in range(0, F * F * K_in):
                psum += w[m, idx] * im2col[idx]  # matrix multiplication
            y[i, j, m] = act(psum)
```

To speed-up the matrix multiplication we can use the same techniques used before:

1. Maximize data reuse in register file.
2. Improve kernel regularity.
3. Exploit parallelism.

- Load 16 weights:
  - 4 out channels;
  - 4 in channels.
- Load 8 pixels:

  - 2 rows;
  - 4 in channels.
- Compute 32 MAC over 8 accumulators.



Then loop over in channels and filter size. There is a dimension that is not touched by tiling: the loop over W_out. We can parallelize over this loop by having each core computing a different column of output activation. With 8 cores we achieve 15.5 MAC/cycle.