Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA)

Unit 11

DNN computing on GPU

LOTS of MM (BLAS3) in Single Precision, Parallel !!

Kwai Wong, Stan Tomov,
Julian Halloy, Stephen Qiu, Eric Zhao

April 14, 2022

University of Tennessee, Knoxville

Acknowledgements:

- Support from NSF, UTK, JICS, ICL, NICS
- LAPENNA, www.jics.utk.edu/lapenna, NSF award #202409
- www.icl.utk.edu, cfdlab.utk.edu, www.xsede.org, www.jics.utk.edu/recsem-reu,
- MagmaDNN is a project grown out from the RECSEM REU Summer program supported under NSF award #1659502
- Source code: www.bitbucket.org/icl/magmadnn
- www.bitbucket.org/cfdl/opendnnwheel

INNOVATIVE COMPUTING LABORATORY

Overview of LAPENNA

The major goal of The Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA) program is to provide essential knowledge to advance literacy in AI to sustain the growth and development of the workforce in the cyberinfrastructure (CI) ecosystem. This program aims to prepare college faculty, researchers, and industrial practitioners to design, enable and direct their own course curricula, collaborative projects, and training programs for in-house data-driven sciences programs. The LAPENNA program focuses on delivering computational techniques, numerical algorithms and libraries, and implementation of AI software on emergent CPU and GPU platforms.

Ecosystem Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA)

Modeling, Numerical Linear Algebra, Data Analytics, Machine Learning, DNN, GPU, HPC

Session 1	Session 2	Session 3	Session 4
7/2020 - 12/2020	1/2021- 6/2021	7/2021 - 1/2022	1/2022 - 6/2022
16 participants	16 participants	16 participants	16 participants
Faculty/Students	Faculty/Students	Faculty/Students	Faculty/Students
10 webinars	10 webinars	10 webinars	10 webinars
4 Q & A	4 Q & A	4 Q & A	4 Q & A

Colleges courses, continuous integration, online courses, projects, software support

Web-based resources, tutorials, webinars, training, outreach

Teams and Participants of LAPENNA

- ✓ **PIs : Kwai Wong (JICS), Stan Tomov (ICL), University of Tennessee, Knoxville**
  - **Stephen Qiu, Julian Halloy, Eric Zhao (Students)**

- ✓ **Team : Clemson University**
- ✓ **Teams : University of Arkansas**
- ✓ **Team : University of Houston, Clear Lake**
- ✓ **Team : Miami University, Ohio**
- ✓ **Team : Boston University**
- ✓ **Team : West Virginia University**
- ✓ **Team : Louisiana State University, Alexandria**
- ✓ **Teams : Jackson Laboratory**
- ✓ **Team : Georgia State University**
- ✓ **Teams : University of Tennessee, Knoxville**
- ✓ **Teams : Morehouse College, Atlanta**
- ✓ **Team : North Carolina A & T University**
- ✓ **Team : Clark Atlanta University, Atlanta**
- ✓ **Team : Alabama A & M University**
- ✓ **Team : Slippery Rock University**
- ✓ **Team : University of Maryland, Baltimore County**

Meeting Time

- ✓ **Webinar Meeting time. Thursday 8:00 – 10:00 pm ET,**
- ✓ **Tentative schedule, www.jics.utk.edu/lapenna --> Spring 2022**

Topic: LAPENNA Spring 2022 Webinar
Time: Feb 3, 2022 08:00 PM Eastern Time (US and Canada)
Every week on Thu, 12 occurrence(s)
Feb 3, 2022 07:30 PM
Feb 10, 2022 07:30 PM
Feb 17, 2022 07:30 PM
Feb 24, 2022 07:30 PM
Mar 3, 2022 07:30 PM
Mar 10, 2022 07:30 PM
Mar 17, 2022 07:30 PM
Mar 24, 2022 07:30 PM
Mar 31, 2022 07:30 PM
Apr 7, 2022 07:30 PM
Apr 14, 2022 07:30 PM
Apr 21, 2022 07:30 PM

Join from PC, Mac, Linux, iOS or Android: https://tennessee.zoom.us/j/94140469394
Password: 708069

Schedule of LAPENNA Spring 2022
Thursday 8:00pm -10:00pm Eastern Time

The goal of The Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA) program is to provide essential knowledge to advance literacy in AI to sustain the growth and development of the workforce in the cyberinfrastructure (CI) ecosystem for data-driven applications.

Month	Week	Date	Topics
February	Week 01	3	Logistics, High Performance Computing
	Week 02	10	Computational Ecosystem, Linear Algebra
	Week 03	17	Introduction to DNN, Forward Path (MLP)
	Week 04	24	Backward Path (MLP), Math, Example
March	Week 05	3	Backpro (MLP), CNN Computation
	Week 06	10	CNN Backpropagation, Example
	Week 07	17	CNN Network, Linear Algebra
	Week 08	24	Segmentation, Unet,
	Week 09	31	Object Detection, RC vehicle
April	Week 10	7	RNN, LSTEM, Transformers
	Week 11	14	DNN Computing on GPU
	Week 12	21	Overview, Closing
June or July	Workshop	To be arranged	4 Days In Person at UTK

Date of Workshop
In person

INNOVATIVE COMPUTING LABORATORY

- **Option A : July 11, 12, 13, 14 (end at noon)**
- **Option B : July 18, 19, 20, 21 (end at noon)**
- **Option C : July 25, 26, 27, 28 (end at noon)**
- **Option D : August 8, 9, 10, 11 (end at noon)**

References

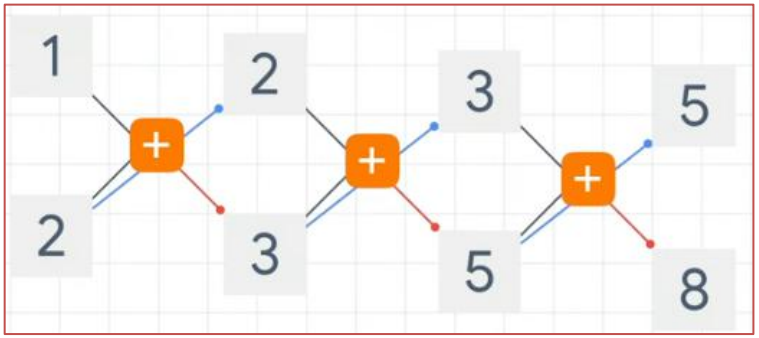
- ✓ https://www.youtube.com/watch?v=SEnXr6v2ifU
- ✓ https://www.youtube.com/watch?v=rZufA635dq4
- ✓ https://www.youtube.com/watch?v=8L11aMN5KY8
- ✓ https://www.youtube.com/watch?v=bcM5AQSAzUY&feature=youtu.be
- ✓ https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLV_1KI9mrSpGFoaxoL9BCZeen_s987Yxb&index=1 - RL david silver
- ✓ https://www.youtube.com/watch?v=fNxaJsNG3-s - NLP from zero to hero
- ✓ https://www.youtube.com/watch?v=Js2WJkhdU7k - neural structure series
- ✓ https://www.youtube.com/watch?v=SEnXr6v2ifU - MIT RNN 6.S191, 6.S094
- ✓ https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2019/schedule.html - Justin Johnson,UM course
- ✓ https://www.youtube.com/watch?v=eyxmSmjmNS0 - classic paper explained
- ✓ http://introtodeeplearning.com/ - MIT DL course
- ✓ https://www.youtube.com/watch?v=Nrsy6vF7rSw Washington University Jeff Heaton – T81-558
- ✓ https://developers.google.com/machine-learning/gan - GAN
- ✓ https://www.youtube.com/watch?v=5N9V07EIfIg&list=PLOg0ngHtcqbPTlZzRHA2ocQZqB1D_qZ5V&index=1 - statistical learning –Tevor Hastie and Robert Tibshirani
- ✓ https://www.datacamp.com/community/tutorials/lstm-python-stock-market
- ✓ https://datascienceplus.com/long-short-term-memory-lstm-and-how-to-implement-lstm-using-python/
- ✓ https://arxiv.org/pdf/1506.00019.pdf - A Critical Review of Recurrent Neural Networks for Sequence Learning
- ✓ https://medium.com/@mliuzzolino/hello-rnn-55a9237b7112
- ✓ https://machinelearningmastery.com/develop-character-based-neural-language-model-keras/
- ✓ https://www.youtube.com/watch?v=DUxYvf1IW4Q
- ✓ http://cs231n.stanford.edu/
- ✓ https://www.tensorflow.org/guide/keras/rnn
- ✓ https://www.tensorflow.org/tutorials/text
- ✓ https://www.youtube.com/watch?v=o3y1w6-Xhjg - Simon institute
- ✓ https://github.com/nicodjimenez/lstm
- ✓ http://nicodjimenez.github.io/2014/08/08/lstm.html
- ✓ **https://github.com/JY-Yoon/RNN-Implementation-using-NumPy/blob/master/RNN%20Implementation%20using%20NumPy.ipynb**
- ✓ https://towardsdatascience.com/recurrent-neural-networks-by-example-in-python-ffd204f99470

Content of Unit 10

- ✓ **RNN**
- ✓ **CODES**
- ✓ **LSTM, CODES**
- ✓ **Natural Language Processing, NLP**

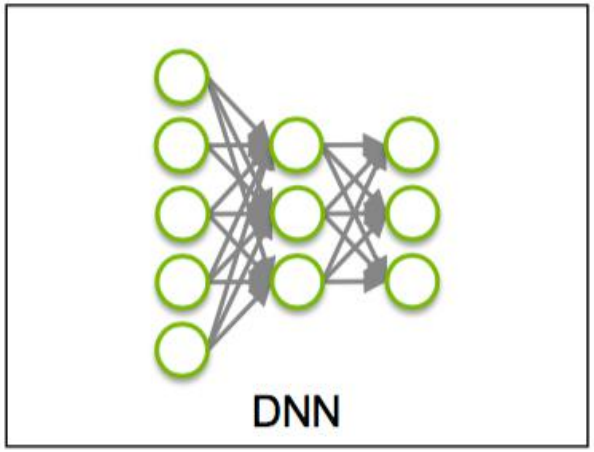
1	2	3	5	8	13	21	34	55	89
n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9

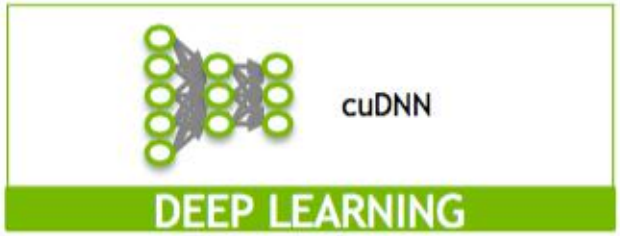
$$n_X = n_{X-1} + n_{X-2}$$

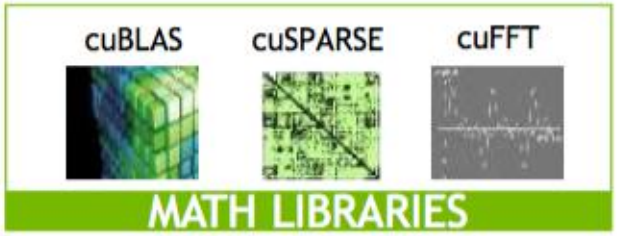


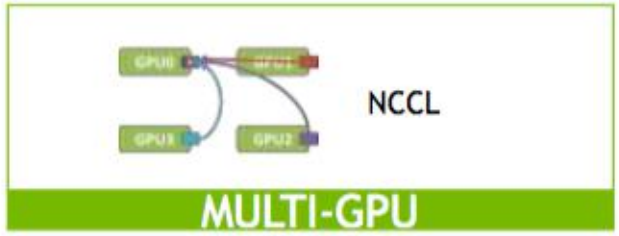
RNN Example : Fibonacci sequence

Machine Learning – GPU acceleration









Source captured frin - Julie Bernauer – HPC Advisory Council Stanford Tutorial – 2017/02/07

Supervised learning is a type of machine learning which automate decision-making processes by generalizing from known examples. That is, the users provides the algorithm with pairs of inputs and desired outputs. The algorithm finds a way to produce the desired output given an input. In other words, we construct a model for the problem. The model is governed by some unknown parameters which we will learn from the data.

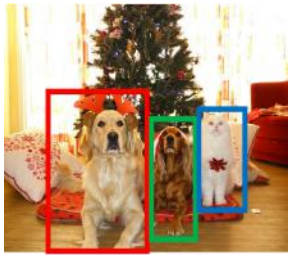
Supervised Learning Data:
(x, y) x is data, y is label

Goal: Learn a function to map x -> y

Examples:
Classification (discrete values),
regression (numerical values),

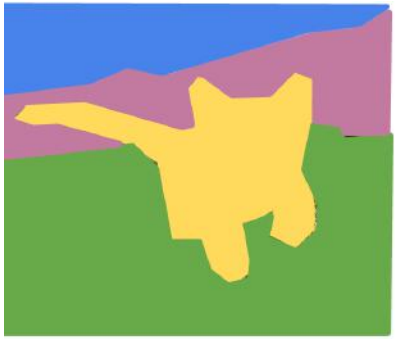
CAT

Classification



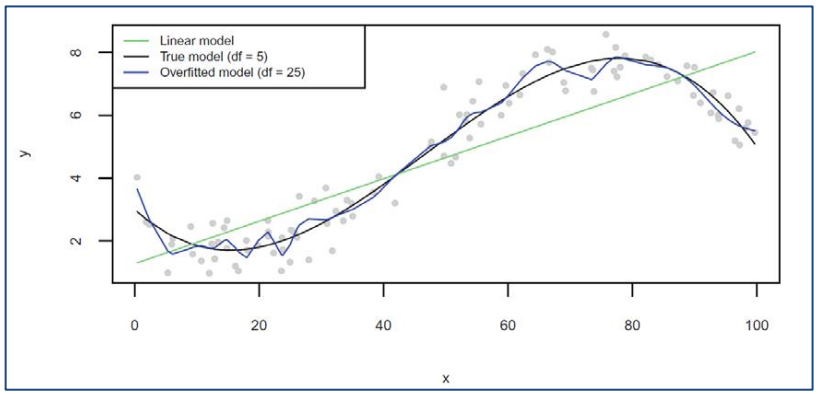
DOG, DOG, CAT

Object Detection



GRASS, CAT, TREE, SKY

Semantic Segmentation



Basic Ideas

Typical Neural Network – MLP

Convolutional Neural Network

STEP 1 : Model Definition

STEP 2 : Cost Function

Step 3 : Optimization Scheme

Step 4 : Numerical Implementation (fitting)

Step 5 : Evaluation

Simple MNIST MLP Fully Connected Neural Network

Input

output

labels

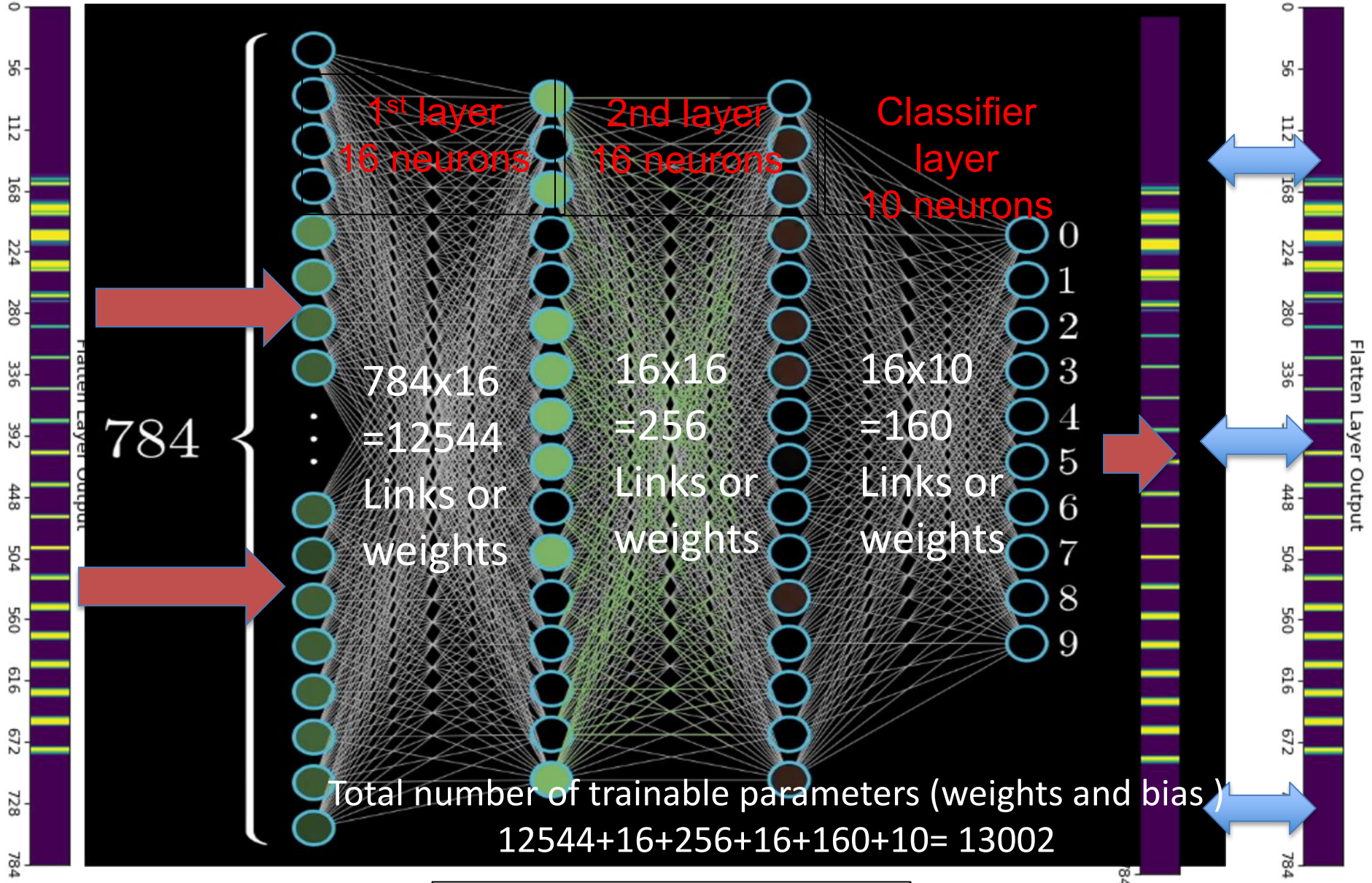
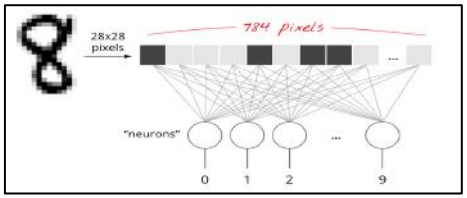


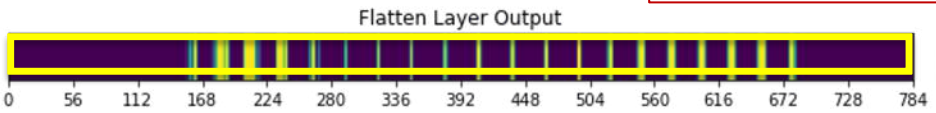
Image:https://www.3blue1brown.com/

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



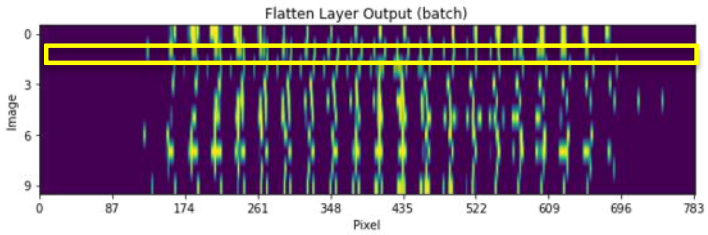
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

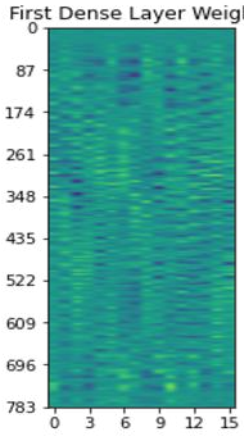
16 neurons
784 x 16 W matrix

10 x 15 output matrix

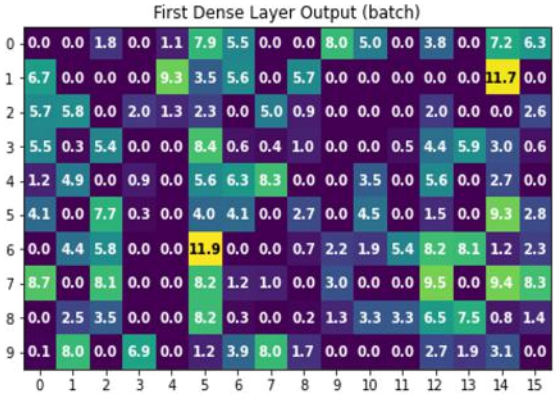
Input to 2nd layer



*

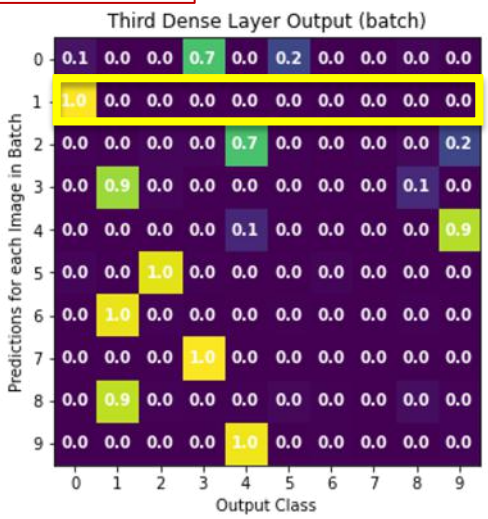


=



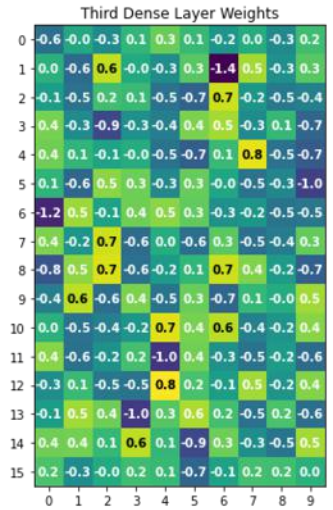
10 x 15 matrix

*



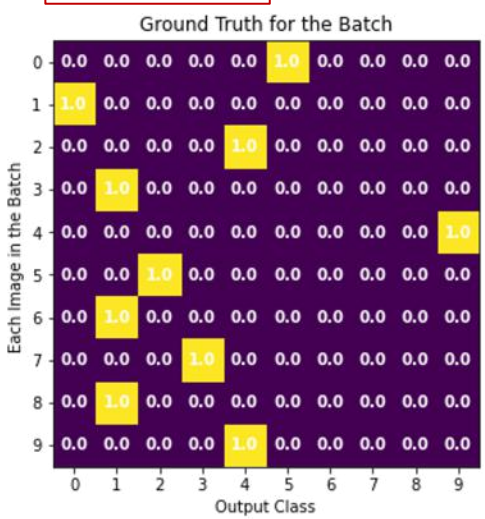
=

10 x 10 output matrix



16 neurons
15 x 10
W matrix

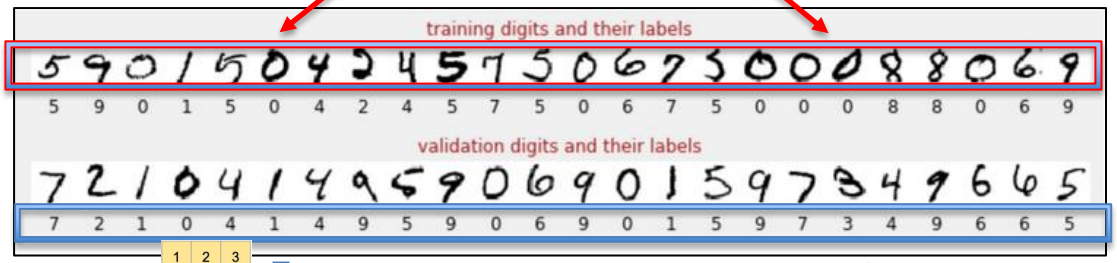
10 x 10 output matrix



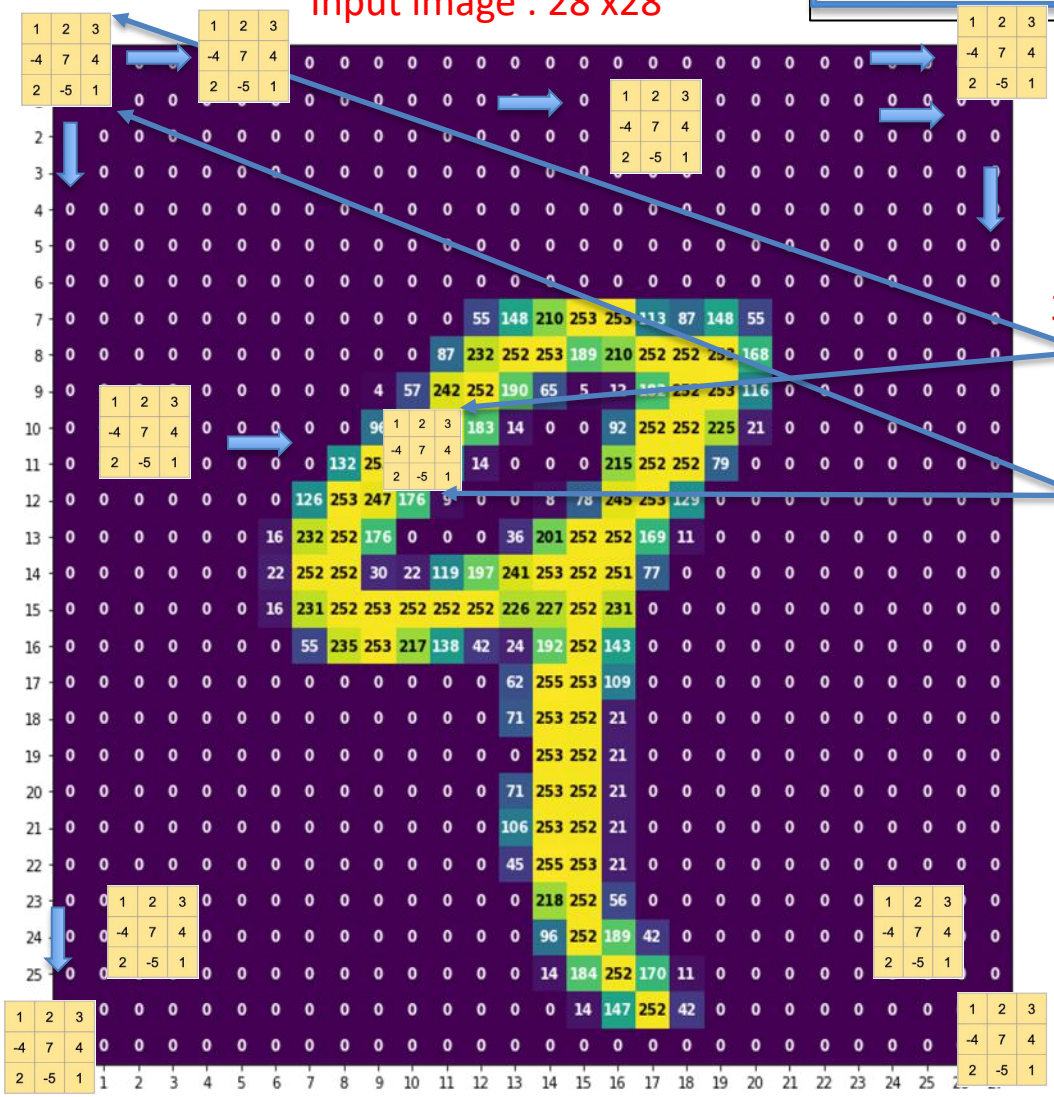
10 x 10 labeled matrix

MNIST Example (28x28 pixels image)

Handwritten digits in the MNIST dataset are 28x28 pixel grayscale images, use the original image (28x28 matrix).

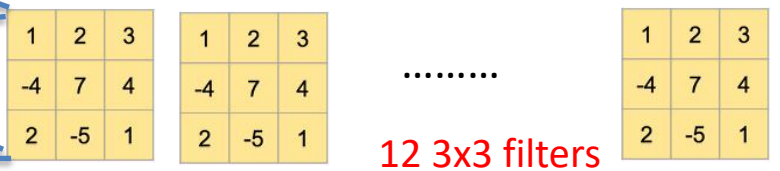


Input image : 28 x28

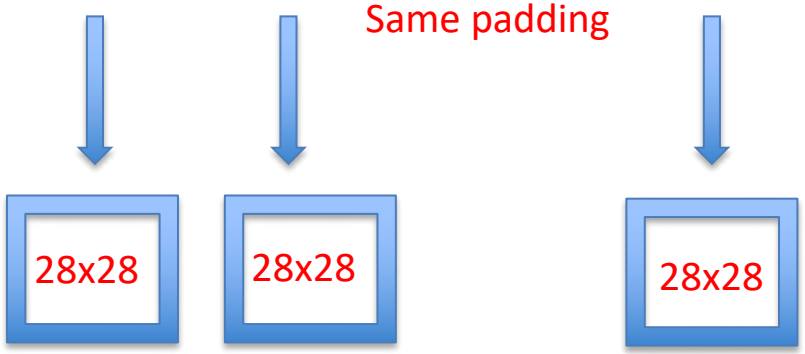


12 filters, each one acts on an image until it is fully covered.

3 x 3 filter (kernel)



12 3x3 filters
Same padding

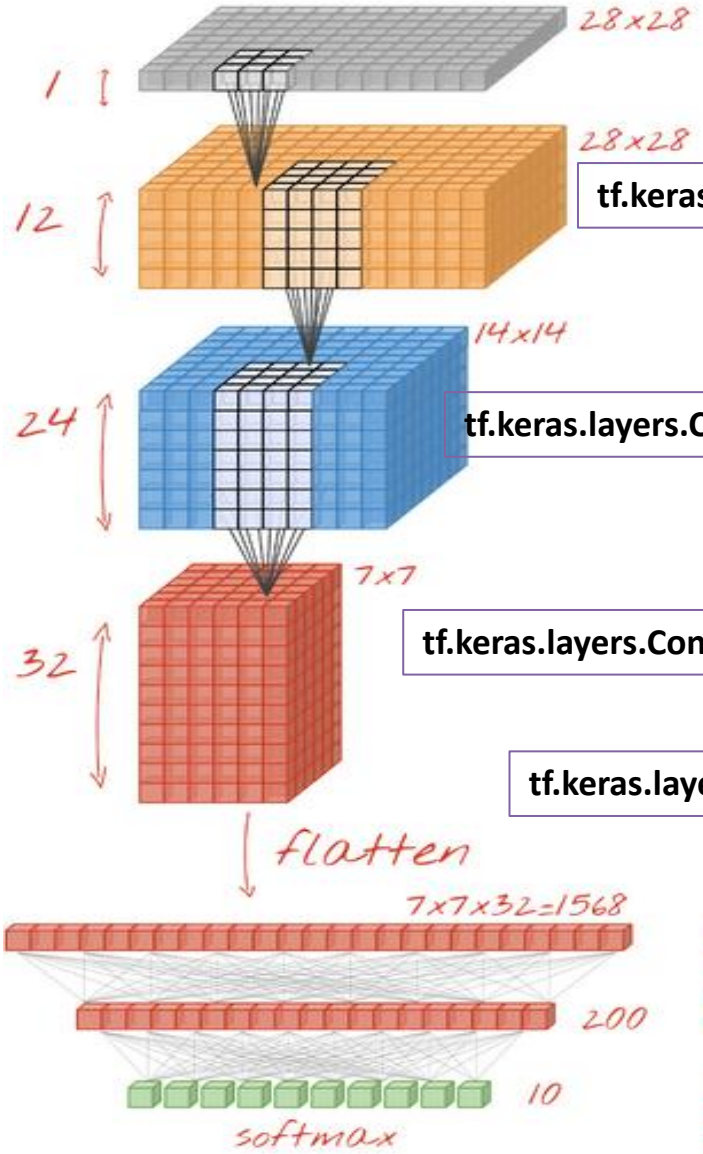


12 Output features : each 28 x28 matrix

Convolution NN : MNIST 28x28 Pixels (Spatial Representation, model)

[FW, FH, C, H] = [filter size ? X ?, input channel (Depth), output channel (no. of filter)]

tf.keras.layers.Reshape(input_shape=(28*28,), target_shape=(28, 28, 1))



Convolutional 3x3 filters=12

$W_1[3, 3, 1, 12]$

tf.keras.layers.Conv2D(kernel_size=3, filters=12, padding='same', activation='relu')

Convolutional 6x6 filters=24

$W_2[6, 6, 12, 24]$ stride 2

tf.keras.layers.Conv2D(kernel_size=6, filters=24, padding='same', activation='relu',strides=2)

Convolutional 6x6 filters=32

$W_2[6, 6, 24, 32]$ stride 2

tf.keras.layers.Conv2D(kernel_size=6, filters=32, padding='same', activation='relu', strides=2)

tf.keras.layers.Flatten()

https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0

Dense layer

$W_4[1568, 200]$

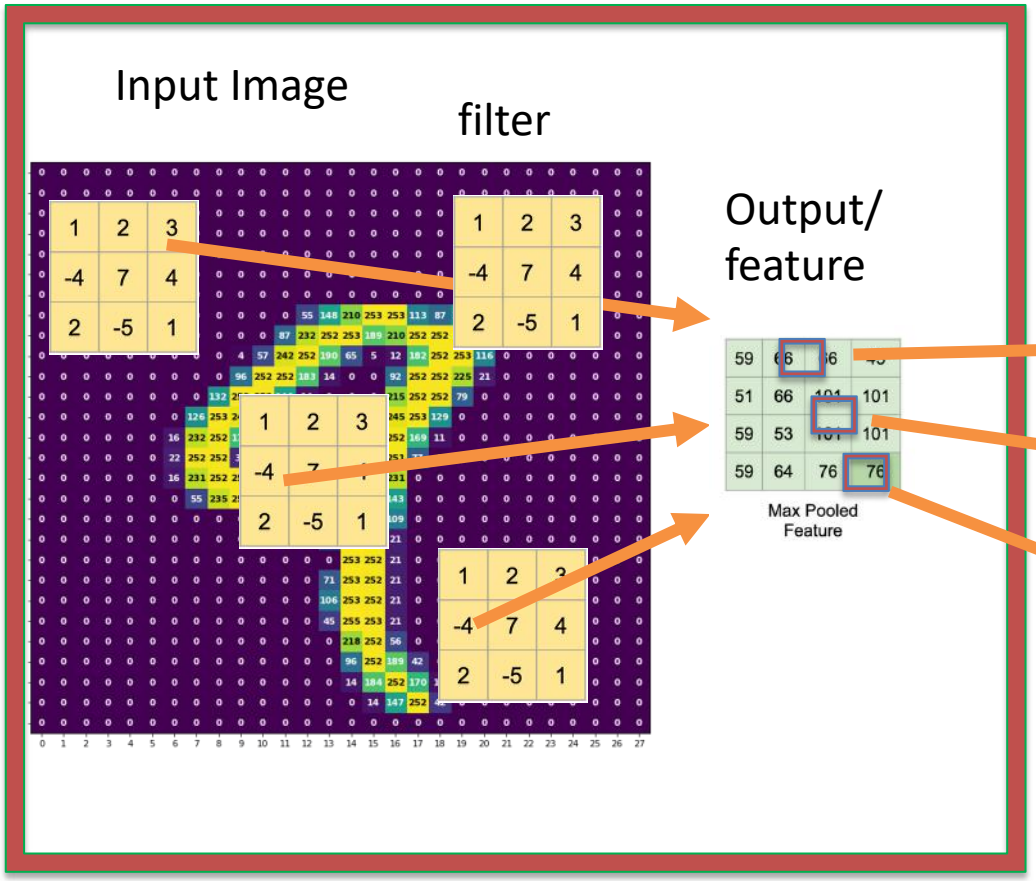
tf.keras.layers.Dense(200, activation='relu')

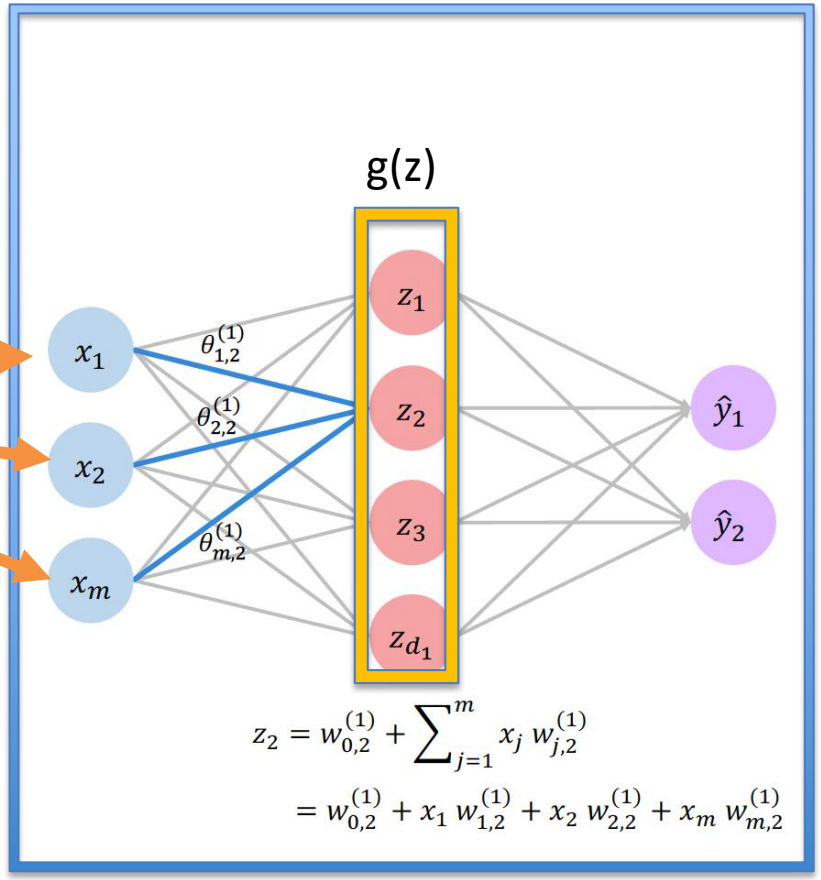
Softmax dense layer

$W_5[200, 10]$

tf.keras.layers.Dense(10, activation='softmax')

Parametric Model : Convolutional Neural Network (CNN)
Convolutional filter + Connected Neural Network





Convolutional filtering

Multilayer Perceptron

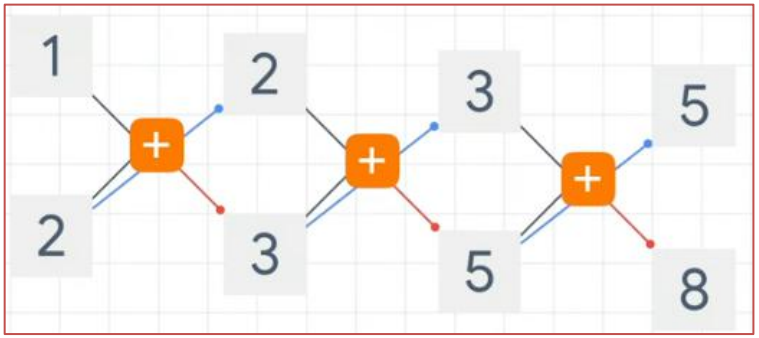
LOTS of MM (BLAS3), in single precision!!

RNN

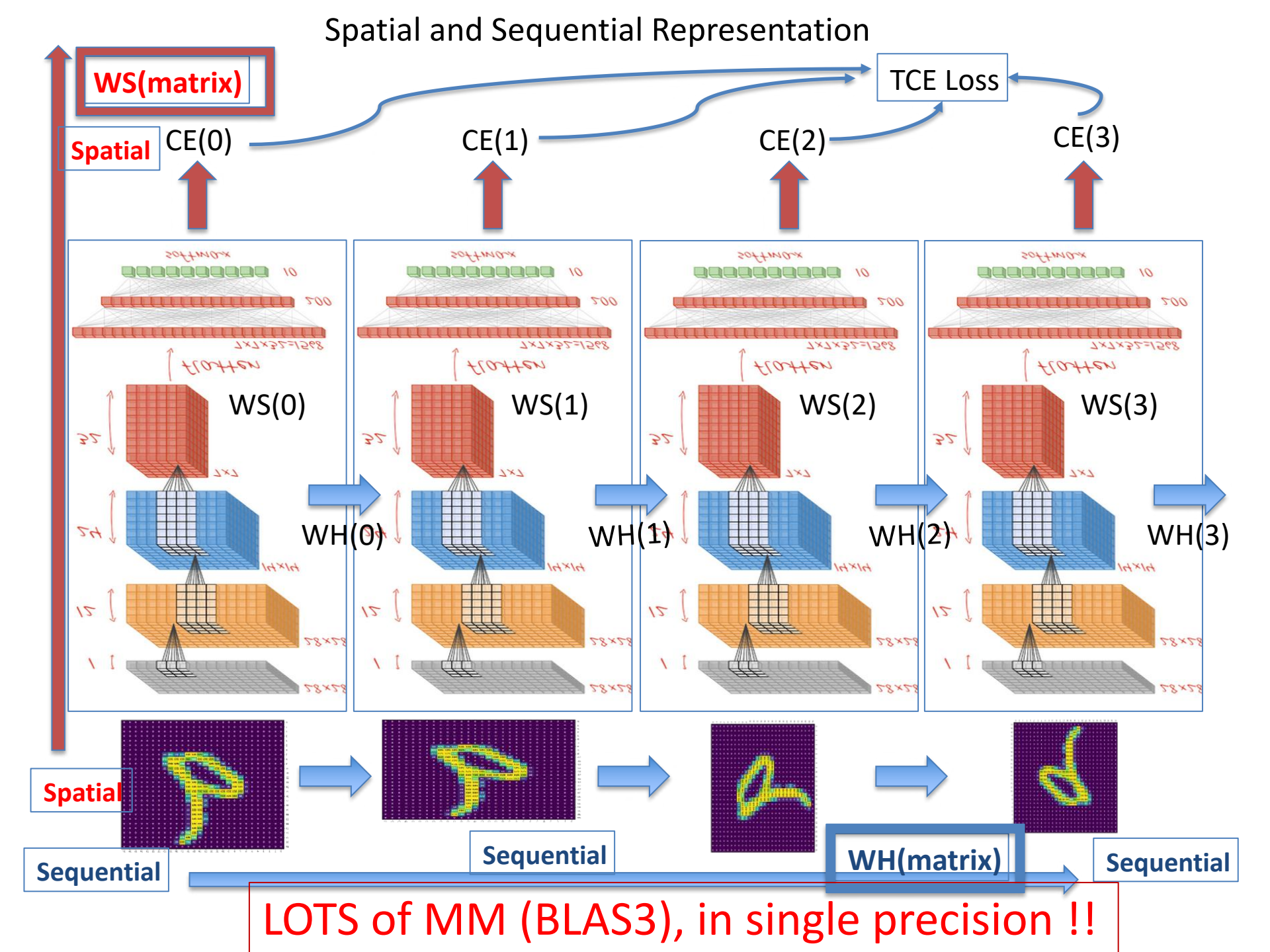
- ✓ **RNN**
- ✓ **LSTM, CODES**
- ✓ **Natural Language Processing, NLP**
- ✓ **GAN**

1	2	3	5	8	13	21	34	55	89
n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9

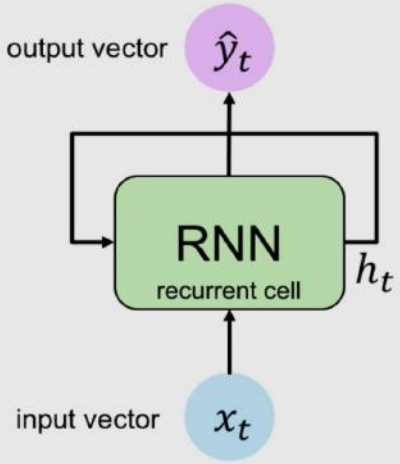
$$n_X = n_{X-1} + n_{X-2}$$



RNN Example : Fibonacci sequence


Spatial and Sequential Representation
WS(matrix)
TCE Loss
Spatial
CE(0)
CE(1)
CE(2)
CE(3)
WS(0)
WS(1)
WS(2)
WS(3)
WH(0)
WH(1)
WH(2)
WH(3)
Spatial
Sequential
Sequential
WH(matrix)
Sequential
LOTS of MM (BLAS3), in single precision !!

Sequential Representation : RNN



Apply a **recurrence relation** at every time step to process a sequence:

$$h_t = f_W(h_{t-1}, x_t)$$

cell state | function parameterized by W | old state | input vector at time step t

Note: the same function and set of parameters are used at every time step

Output Vector

$$\hat{y}_t = W_{hy}^T h_t$$

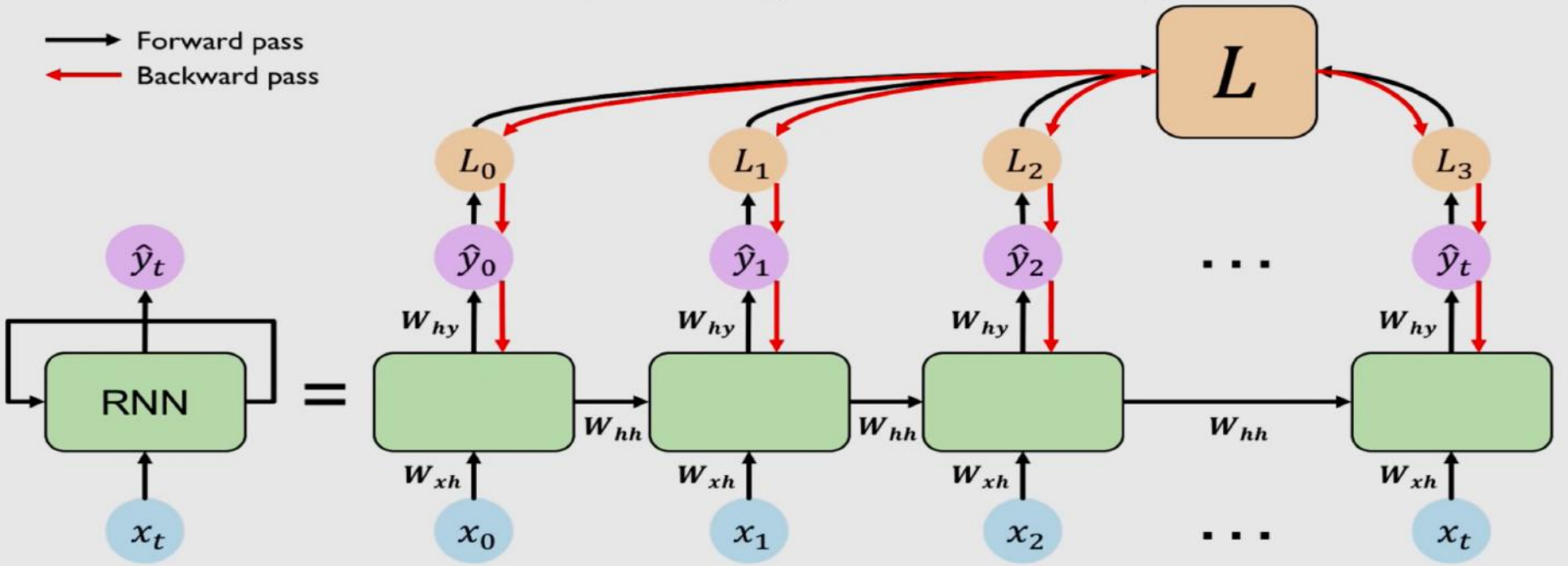
Update Hidden State

$$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$$

Input Vector

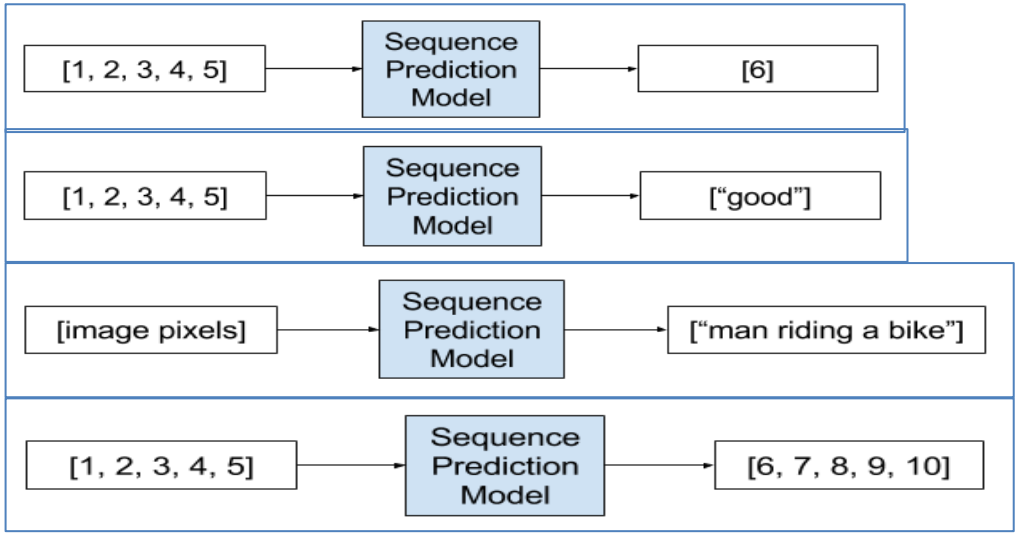
$$x_t$$

the same function and the same set of parameters (W) are used at every time step.



https://web.eecs.umich.edu/~justincj/slides/eecs498/498_FA2019_lecture12.pdf

Predicting Character : RNN



Sequence prediction

Sequence classification

Sequence generation

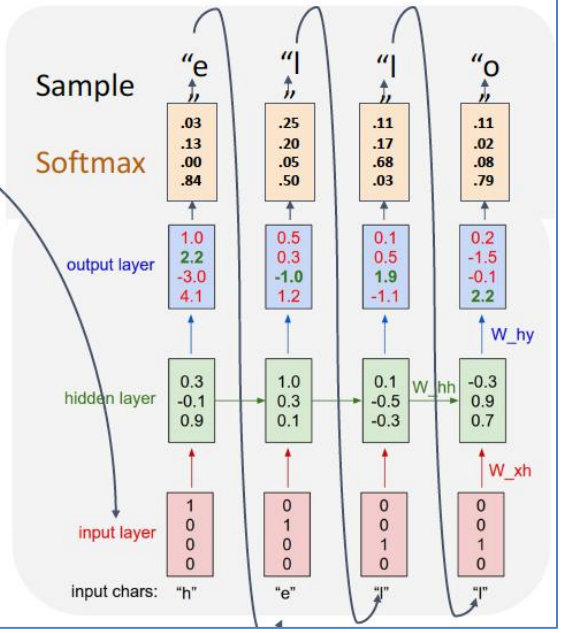
Sequence-to-sequence prediction

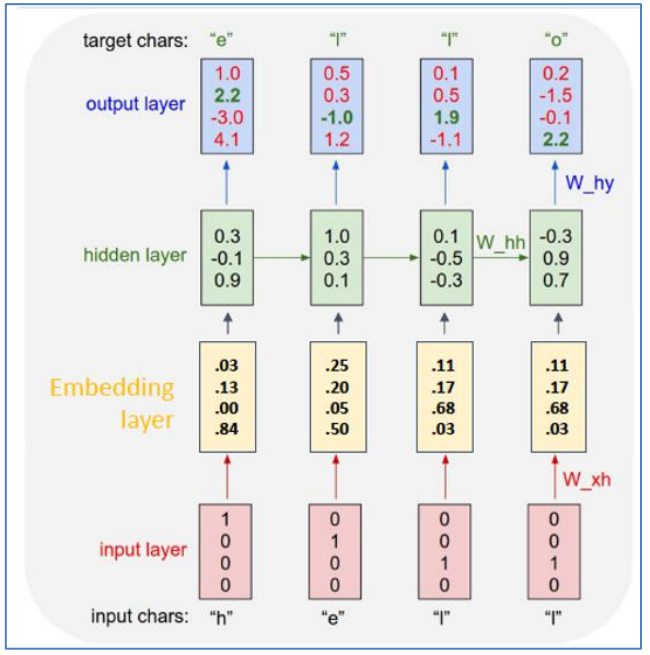
Example: Language Modeling

So far: encode inputs as **one-hot-vector**

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{14} \\ w_{31} & w_{32} & w_{33} & w_{14} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} w_{11} \\ w_{21} \\ w_{31} \end{bmatrix}$$

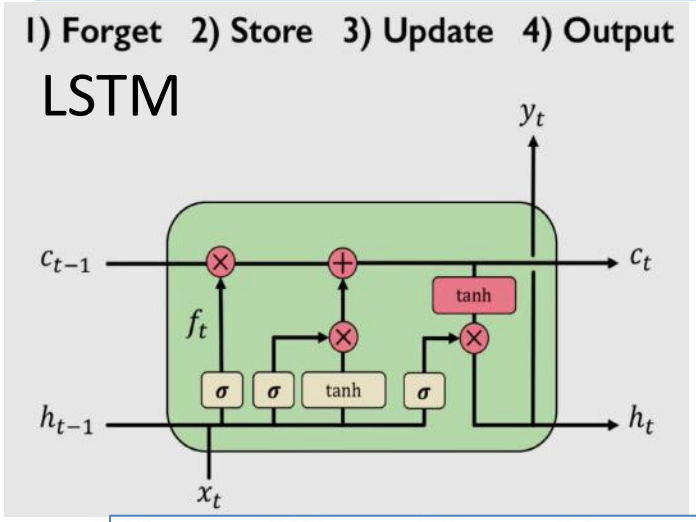
Matrix multiply with a one-hot vector just extracts a column from the weight matrix. Often extract this into a separate **embedding** layer

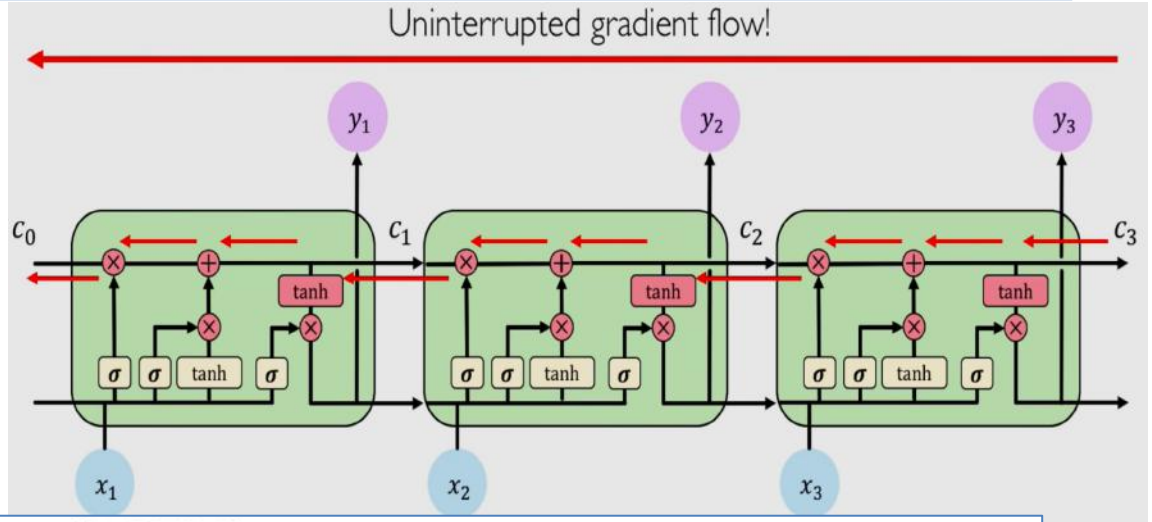




Long Short Term Memory Network (LSTM)

Backpropagation from C(t) to C(t-1) only elementwise multiplication by f, no matrix multiply by W

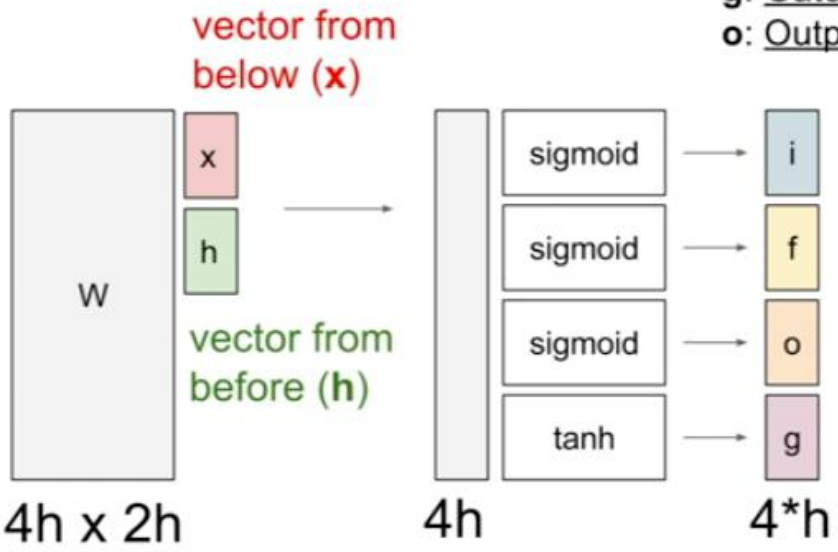




Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

f: Forget gate, Whether to erase cell
i: Input gate, whether to write to cell
g: Gate gate (?), How much to write to cell
o: Output gate, How much to reveal cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

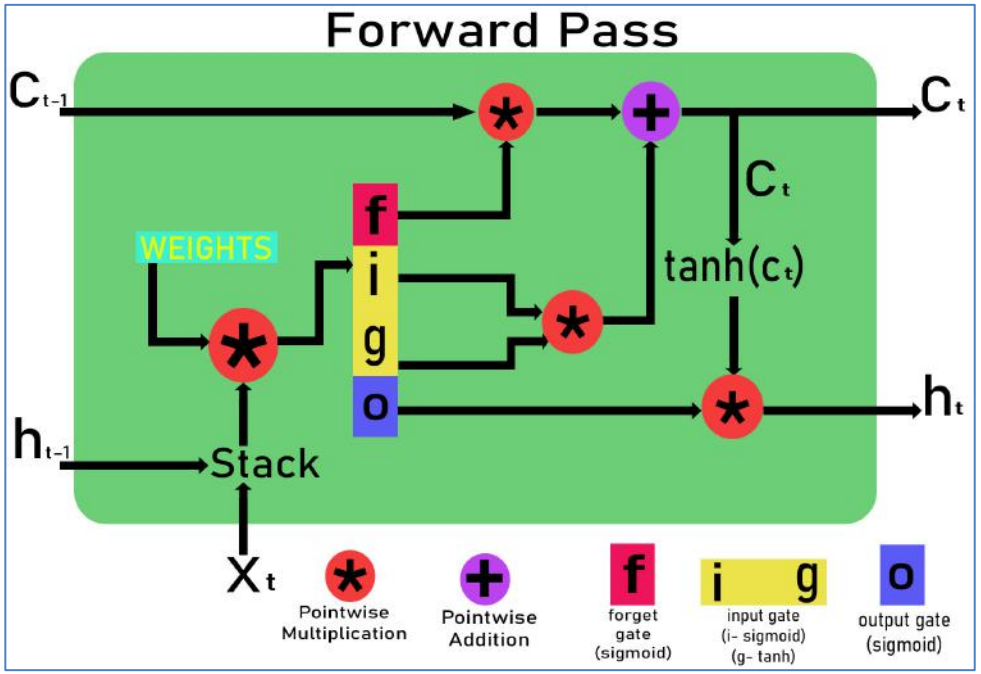
$$h_t = o \odot \tanh(c_t)$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$



Inputs: x_t and h_{t-i} , c_{t-1} are given to the LSTM cell

Passing through input gate:

$Z_g = w_{xg} * x + w_{hg} * h_{t-1} + b_g$

$g = \tanh(Z_g)$

$Z_j = w_{xi} * x + w_{hi} * h_{t-1} + b_i$

$i = \text{sigmoid}(Z_i)$

Input_gate_out = g*i

Passing through forget gate:

$Z_f = w_{xf} * x + w_{hf} * h_{t-1} + b_f$

$f = \text{sigmoid}(Z_f)$

Forget_gate_out = f

Passing through the output gate:

$Z_o = w_{xo} * x + w_{ho} * h_{t-1} + b_o$

$o = \text{sigmoid}(z_O)$

Out_gate_out = o

Weights for different gates are :

Input gate : w_{xi}, w_{xg}, b_i, w_{hj}, w_g , b_g

Forget gate : w_{xf}, b_f, w_{hf}

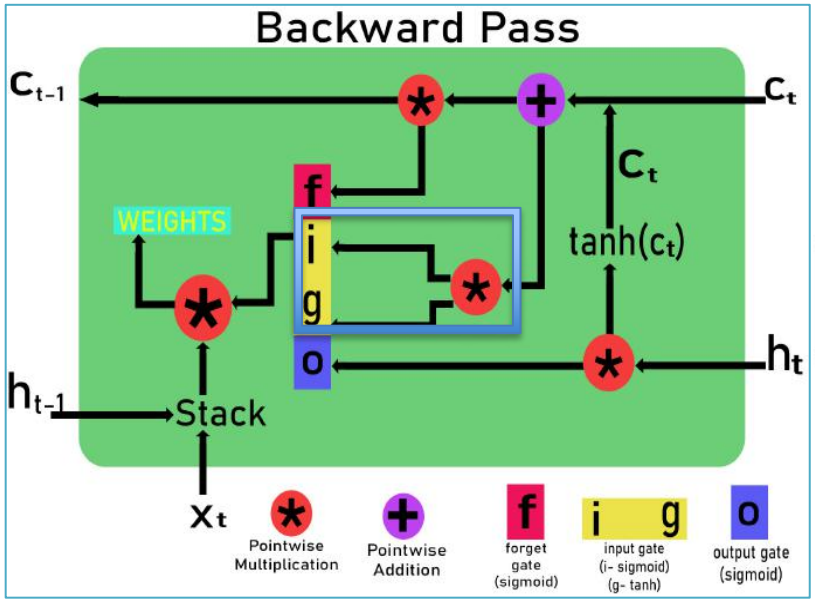
Output gate : w_{xo}, b_o, w_{ho}

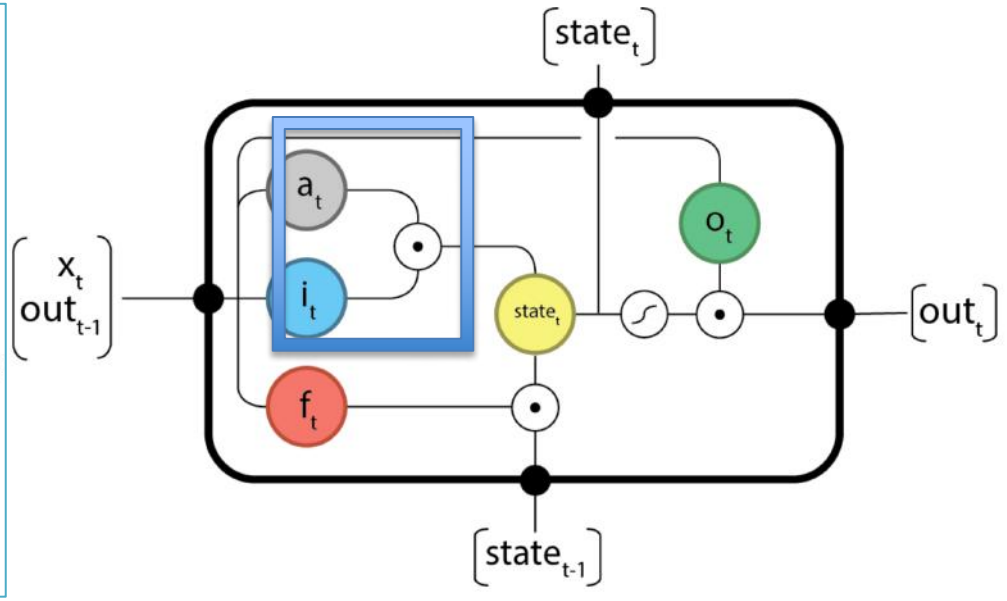
Calculating the current cell state

c_t : $c_t = (c_{t-1} * \text{forget\_gate\_out}) + \text{input\_gate\_out}$

Calculating the output gate ht:

h_t=out_gate_out * tanh(ct)





$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

h(t) = out(t)
C(t) = state(t)
g(t)=a(t)

W = [Wi, Ui ; Wf, Uf ; Wo, Uo ; Wg, Ug]

Input activation:

$$a_t = \tanh(W_a \cdot x_t + U_a \cdot out_{t-1} + b_a)$$

Input gate:

$$i_t = \sigma(W_i \cdot x_t + U_i \cdot out_{t-1} + b_i)$$

Forget gate:

$$f_t = \sigma(W_f \cdot x_t + U_f \cdot out_{t-1} + b_f)$$

Output gate:

$$o_t = \sigma(W_o \cdot x_t + U_o \cdot out_{t-1} + b_o)$$

Internal state:

$$state_t = a_t \odot i_t + f_t \odot state_{t-1}$$

Output:

$$out_t = \tanh(state_t) \odot o_t$$

Let the gradient pass down by the above cell be:
$E\_delta = dE/dh_t$
If we are using MSE (mean square error)for error then, **E_delta=(y-h(x)),**
Here y is the orignal value and h(x) is the predicted value.

Gradient with respect to output gate
$dE/do = (dE/dh_t) * (dh_t/do) = E\_delta * (dh_t / do)$
$dE/do = E\_delta * \tanh(c_t)$

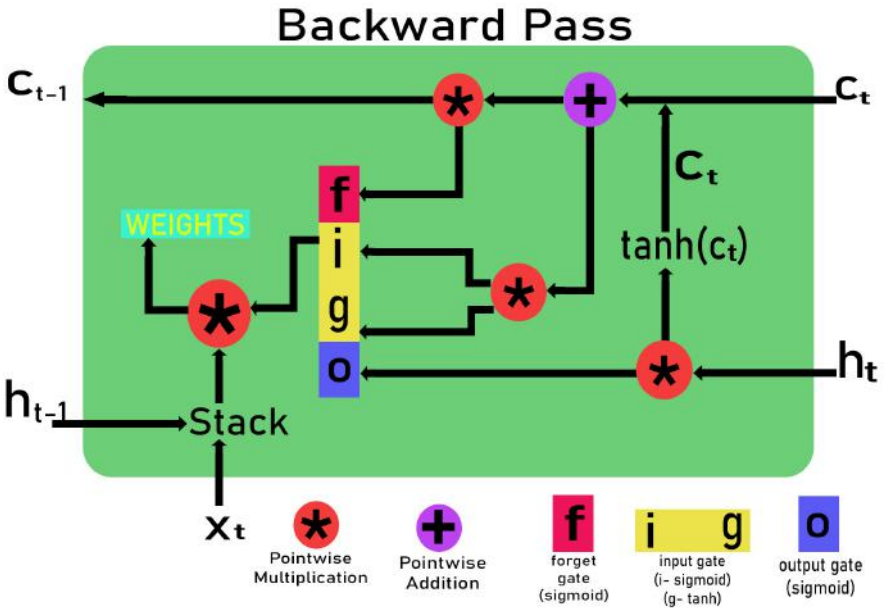
Gradient with respect to c_t
$dE/dc_t = (dE / dh_t) * (dh_t/dc_t) = E\_delta * (dh_t/dc_t)$
$dE/dc_t = E\_delta * o * (1-\tanh^2(c_t))$

Gradient with respect to input gate dE/di, dE/dg
$dE/di = (dE/dh) * (dh/dc) * (dc_t / di)$
$dE/di = E\_delta * o * (1-\tanh^2(c_t)) * g$
Similarly, $dE/dg = E\_delta * o * (1-\tanh^2(c_t)) * i$

Gradient with respect to forget gate
$dE/df = E\_delta * (dE/dc_t) * (dc_t / dt)$
$dE/df = E\_delta * o * (1-\tanh^2(c_t)) * c_{t-1}$

Gradient with respect to c_{t-1}
$dE/dc_t = E\_delta * (dE/dc_t) * (dc_t / dc_{t-1})$
$dE/dc_t = E\_delta * o * (1-\tanh^2(c_t)) * f$

Backpropagation Through Time (BPTT)



Calculating the current cell state
$c_t : c_t = (c_{t-1} * \text{forget\_gate\_out}) + \text{input\_gate\_out}$
$= [C(t-1) * f + g*i]$
Calculating the output gate ht:
$h_t = \text{out\_gate\_out} * \tanh(ct) = o * \tanh(ct)$

https://www.geeksforgeeks.org/lstm-derivation-of-back-propagation-through-time/

Backpropagation Through Time (BPTT)

Every gradient term is known with c, x, g, h, and z(x,h,b)

Gradient with respect to output gate weights:

$dE/dw_{xo} = dE/do * (do/dw_{xo}) = E\_delta * \tanh(c_t) * sigmoid(z_o) * (1-sigmoid(z_o) * x_t$

$dE/dw_{ho} = dE/do * (do/dw_{ho}) = E\_delta * \tanh(c_t) * sigmoid(z_o) * (1-sigmoid(z_o) * h_{t-1}$

$dE/db_o = dE/do * (do/db_o) = E\_delta * \tanh(c_t) * sigmoid(z_o) * (1-sigmoid(z_o)$

Gradient with respect to forget gate weights:

$dE/dw_{xf} = dE/df * (df/dw_{xf}) = E\_delta * o * (1-\tanh^2(c_t)) * c_{t-1} * sigmoid(z_f) * (1-sigmoid(z_f) * x_t$

$dE/dw_{hf} = dE/df * (df/dw_{hf}) = E\_delta * o * (1-\tanh^2(c_t)) * c_{t-1} * sigmoid(z_f) * (1-sigmoid(z_f) * h_{t-1}$

$dE/db_o = dE/df * (df/db_o) = E\_delta * o * (1-\tanh^2(c_t)) * c_{t-1} * sigmoid(z_f) * (1-sigmoid(z_f)$

Gradient with respect to input gate weights:

$dE/dw_{xi} = dE/di * (di/dw_{xi}) = E\_delta * o * (1-\tanh^2(c_t)) * g * sigmoid(z_i) * (1-sigmoid(z_i) * x_t$

$dE/dw_{hi} = dE/di * (di/dw_{hi}) = E\_delta * o * (1-\tanh^2(c_t)) * g * sigmoid(z_i) * (1-sigmoid(z_i) * h_{t-1}$

$dE/db_i = dE/di * (di/db_i) = E\_delta * o * (1-\tanh^2(c_t)) * g * sigmoid(z_i) * (1-sigmoid(z_i)$

$dE/dw_{xg} = dE/dg * (dg/dw_{xg}) = E\_delta * o * (1-\tanh^2(c_t)) * i * (1-\tanh^2(z_g)) * x_t$

$dE/dw_{hg} = dE/dg * (dg/dw_{hg}) = E\_delta * o * (1-\tanh^2(c_t)) * i * (1-\tanh^2(z_g)) * h_{t-1}$

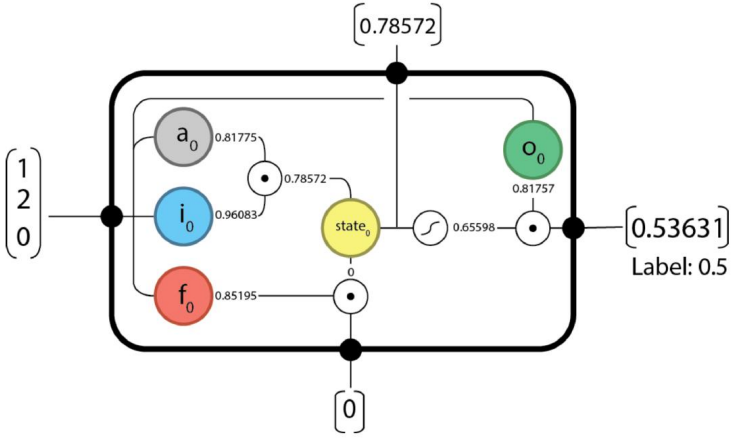
$dE/db_g = dE/dg * (dg/db_g) = E\_delta * o * (1-\tanh^2(c_t)) * i * (1-\tanh^2(z_g))$

w+ = w + lr * grad(w), h+= h – lr*grad(h)

LSTM – Example – Forward Path

Gate S(t) = C(t)

$$gates_t = \begin{bmatrix} a_t \\ i_t \\ f_t \\ o_t \end{bmatrix}, \; W = \begin{bmatrix} W_a \\ W_i \\ W_f \\ W_o \end{bmatrix}, \; U = \begin{bmatrix} U_a \\ U_i \\ U_f \\ U_o \end{bmatrix}, \; b = \begin{bmatrix} b_a \\ b_i \\ b_f \\ b_o \end{bmatrix}$$



$$W_a = \begin{bmatrix} 0.45 \\ 0.25 \end{bmatrix}, U_a = \begin{bmatrix} 0.15 \end{bmatrix}, b_a = \begin{bmatrix} 0.2 \end{bmatrix}$$

$$W_i = \begin{bmatrix} 0.95 \\ 0.8 \end{bmatrix}, U_i = \begin{bmatrix} 0.8 \end{bmatrix}, b_i = \begin{bmatrix} 0.65 \end{bmatrix}$$

$$W_f = \begin{bmatrix} 0.7 \\ 0.45 \end{bmatrix}, U_f = \begin{bmatrix} 0.1 \end{bmatrix}, b_f = \begin{bmatrix} 0.15 \end{bmatrix}$$

$$W_o = \begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix}, U_o = \begin{bmatrix} 0.25 \end{bmatrix}, b_o = \begin{bmatrix} 0.1 \end{bmatrix}$$

input

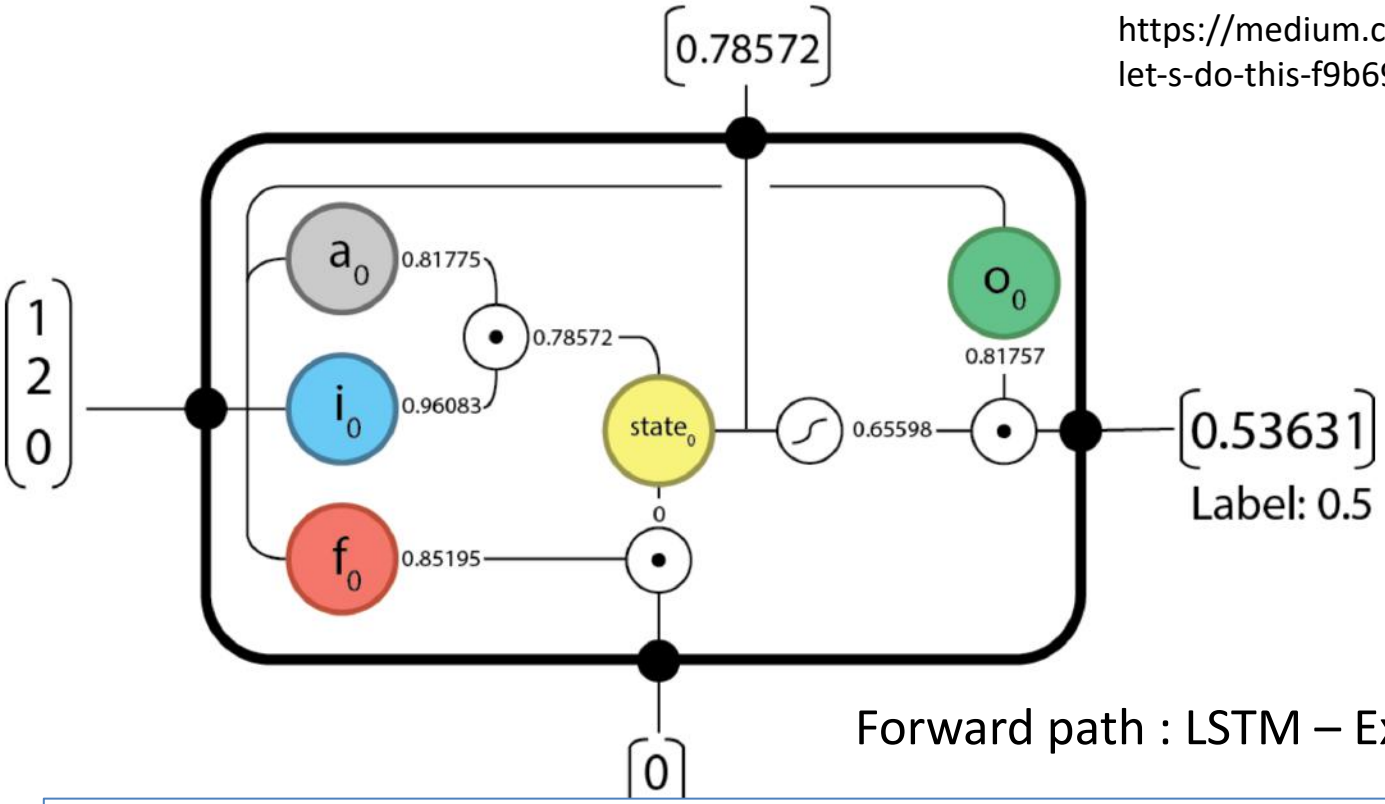
$$x_0 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad x_1 = \begin{bmatrix} 0.5 \\ 3 \end{bmatrix}$$

LOTS of MM (BLAS3), single precision !!

X0 label = Y0 = 0.5
X1 label = Y1 = -1.25

https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9

https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9



Forward path : LSTM – Example, t=0

$$a_0 = \tanh(W_a \cdot x_0 + U_a \cdot out_{-1} + b_a) = \tanh(\begin{bmatrix}0.45 & 0.25\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}0.15\end{bmatrix}\begin{bmatrix}0\end{bmatrix} + \begin{bmatrix}0.2\end{bmatrix}) = 0.81775$$

$$i_0 = \sigma(W_i \cdot x_0 + U_i \cdot out_{-1} + b_i) = \sigma(\begin{bmatrix}0.95 & 0.8\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}0.8\end{bmatrix}\begin{bmatrix}0\end{bmatrix} + \begin{bmatrix}0.65\end{bmatrix}) = 0.96083$$

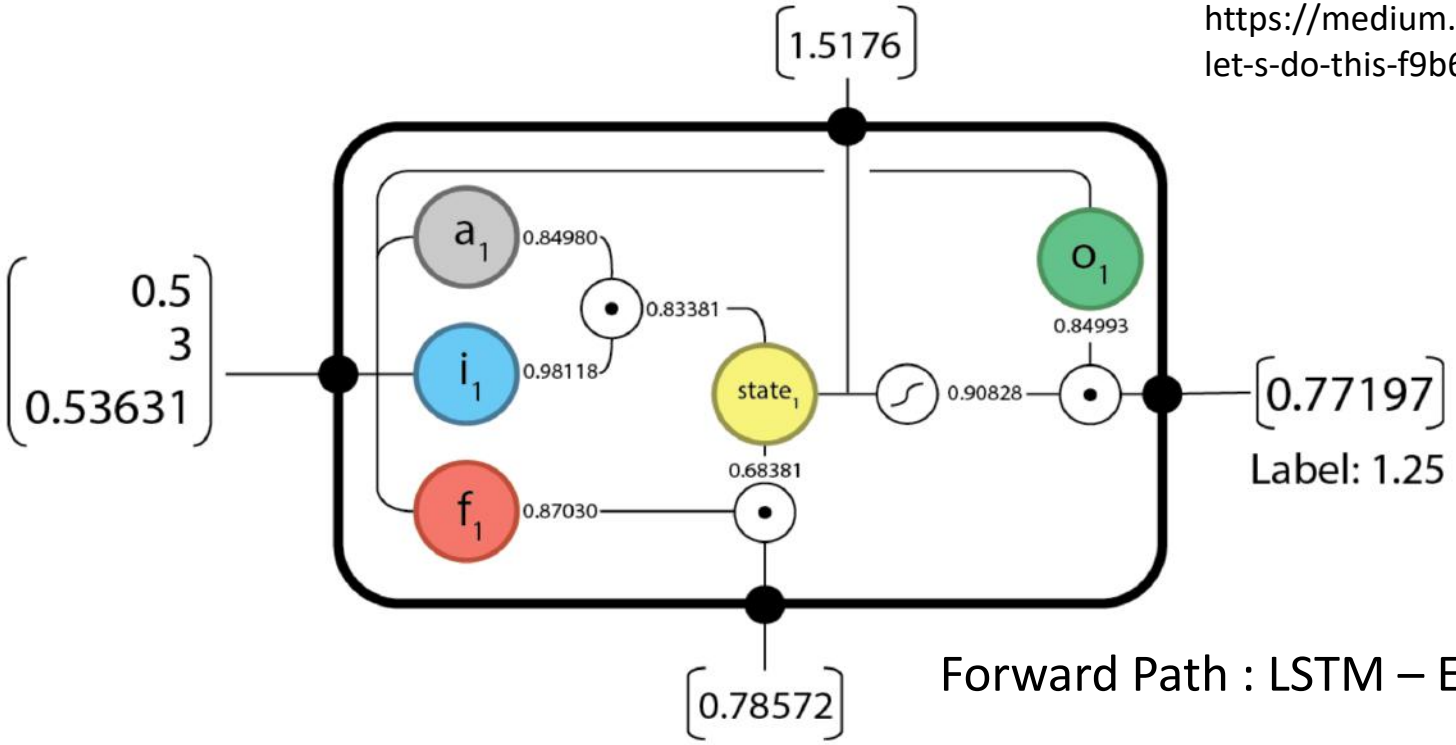
$$f_0 = \sigma(W_f \cdot x_0 + U_f \cdot out_{-1} + b_f) = \sigma(\begin{bmatrix}0.7 & 0.45\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}0.1\end{bmatrix}\begin{bmatrix}0\end{bmatrix} + \begin{bmatrix}0.15\end{bmatrix}) = 0.85195$$

$$o_0 = \sigma(W_o \cdot x_0 + U_o \cdot out_{-1} + b_o) = \sigma(\begin{bmatrix}0.6 & 0.4\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}0.25\end{bmatrix}\begin{bmatrix}0\end{bmatrix} + \begin{bmatrix}0.1\end{bmatrix}) = 0.81757$$

$$state_0 = a_0 \odot i_0 + f_0 \odot state_{-1} = 0.81775 \times 0.96083 + 0.85195 \times 0 = 0.78572$$

$$out_0 = \tanh(state_0) \odot o_0 = \tanh(0.78572) \times 0.81757 = 0.53631$$

https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9



Forward Path : LSTM – Example, t=1

$$a_1 = \tanh(W_a \cdot x_1 + U_a \cdot out_0 + b_a) = \tanh\left([0.45\ 0.25]\begin{bmatrix}0.5\\3\end{bmatrix} + [0.15]\,[0.53631] + [0.2]\right) = 0.84980$$

$$i_1 = \sigma(W_i \cdot x_1 + U_i \cdot out_0 + b_i) = \sigma\left([0.95\ 0.8]\begin{bmatrix}0.5\\3\end{bmatrix} + [0.8]\,[0.53631] + [0.65]\right) = 0.98118$$

$$f_1 = \sigma(W_f \cdot x_1 + U_f \cdot out_0 + b_f) = \sigma\left([0.7\ 0.45]\begin{bmatrix}0.5\\3\end{bmatrix} + [0.1]\,[0.53631] + [0.15]\right) = 0.87030$$

$$o_1 = \sigma(W_o \cdot x_1 + U_o \cdot out_0 + b_o) = \sigma\left([0.6\ 0.4]\begin{bmatrix}0.5\\3\end{bmatrix} + [0.25]\,[0.53631] + [0.1]\right) = 0.84993$$

$$state_1 = a_1 \odot i_1 + f_1 \odot state_0 = 0.84980 \times 0.98118 + 0.87030 \times 0.78572 = 1.5176$$

$$out_1 = \tanh(state_1) \odot o_1 = \tanh(1.5176) \times 0.84993 = 0.77197$$

https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9

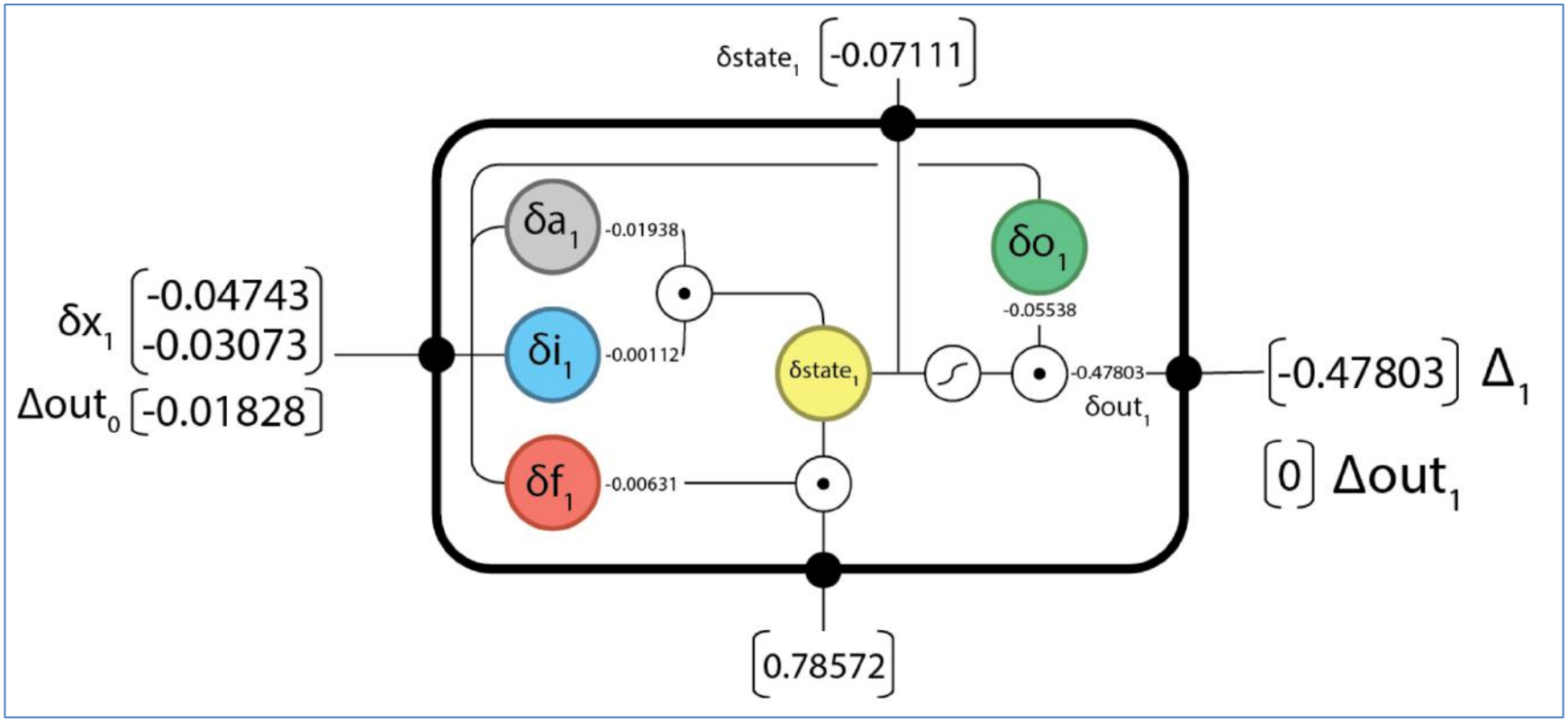
$$\Delta_1 = \partial_x E = 0.77197 - 1.25 = -0.47803$$
$\Delta out_1 = 0$ because there are no future time-steps.

$$\delta out_1 = \Delta_1 + \Delta out_1 = -0.47803 + 0 = -0.47803$$
$$\delta state_1 = \delta out_1 \odot o_1 \odot (1 - \tanh^2(state_1)) + \delta state_2 \odot f_2 = -0.47803 \times 0.84993 \times (1 - \tanh^2(1.5176)) + 0 \times 0 = -0.07111$$

$$\delta a_1 = \delta state_1 \odot i_1 \odot (1 - a_1^2) = -0.07111 \times 0.98118 \times (1 - 0.84980^2) = -0.01938$$
$$\delta i_1 = \delta state_1 \odot a_1 \odot i_1 \odot (1 - i_1) = -0.07111 \times 0.84980 \times 0.98118 \times (1 - 0.98118) = -0.00112$$
$$\delta f_1 = \delta state_1 \odot state_0 \odot f_1 \odot (1 - f_1) = -0.07111 \times 0.78572 \times 0.87030 \times (1 - 0.87030) = -0.00631$$
$$\delta o_1 = \delta out_1 \odot \tanh(state_1) \odot o_1 \odot (1 - o_1) = -0.47803 \times \tanh(1.5176) \times 0.84993 \times (1 - 0.84993) = -0.05538$$

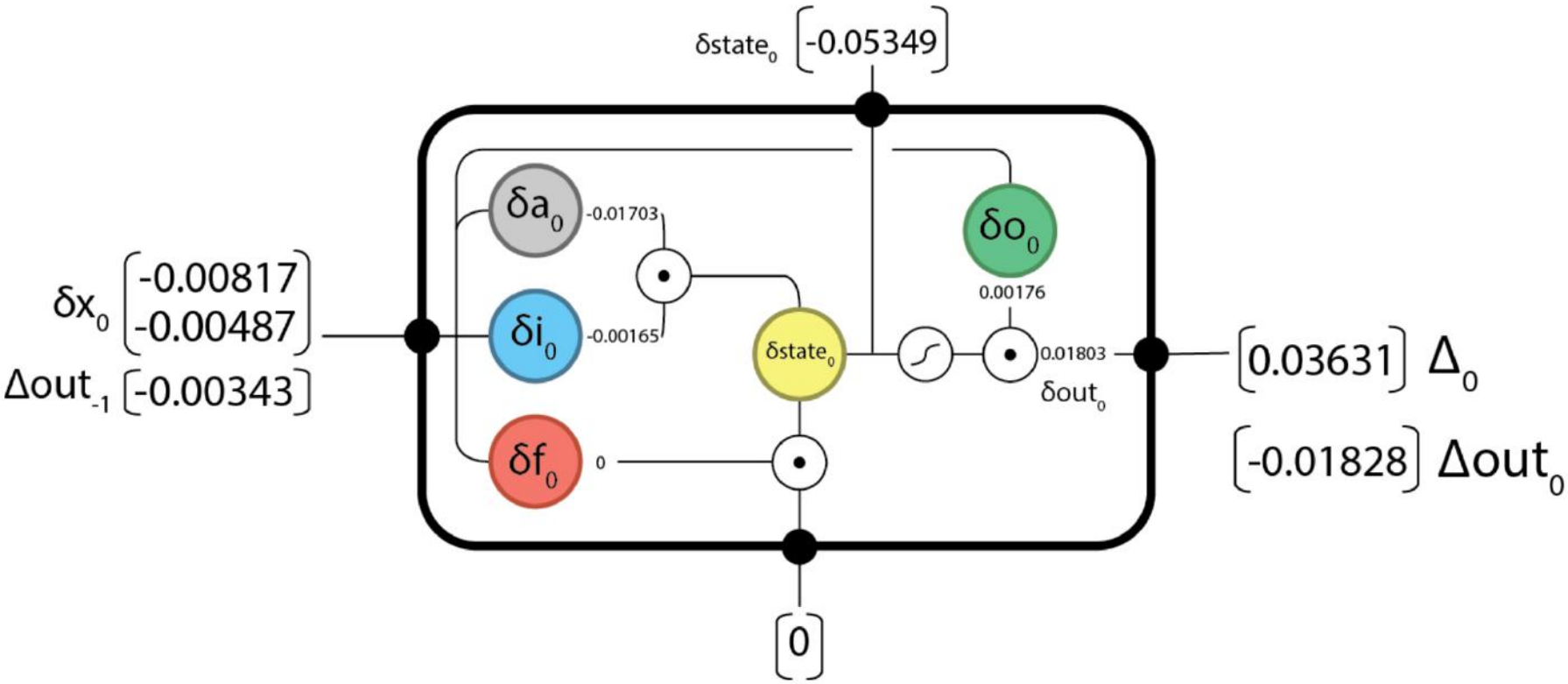
$$\delta x_1 = W^T \cdot \delta gates_1$$
$$= \begin{bmatrix} 0.45 & 0.95 & 0.70 & 0.60 \\ 0.25 & 0.80 & 0.45 & 0.40 \end{bmatrix} \begin{bmatrix} -0.01938 \\ -0.00112 \\ -0.00631 \\ -0.05538 \end{bmatrix} = \begin{bmatrix} -0.04743 \\ -0.03073 \end{bmatrix}$$
$$\Delta out_0 = U^T \cdot \delta gates_1$$
$$= \begin{bmatrix} 0.15 & 0.80 & 0.10 & 0.25 \end{bmatrix} \begin{bmatrix} -0.01938 \\ -0.00112 \\ -0.00631 \\ -0.05538 \end{bmatrix} = -0.01828$$

Backward Path: t = 1



https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9

https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9



https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9

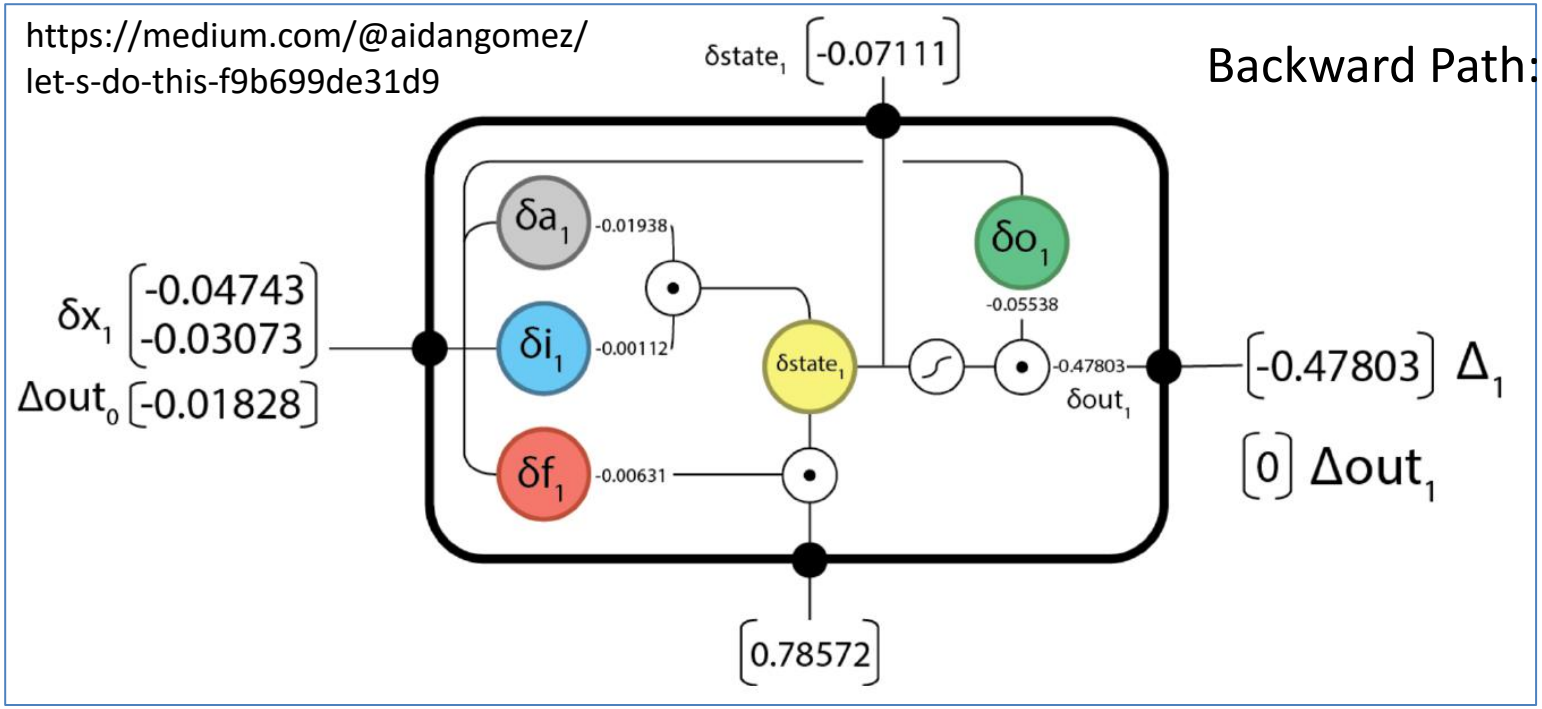
$$\Delta_0 = \partial_x E = 0.53631 - 0.5 = 0.03631$$
$$\Delta out_0 = -0.01828, \text{ passed back from T=1}$$

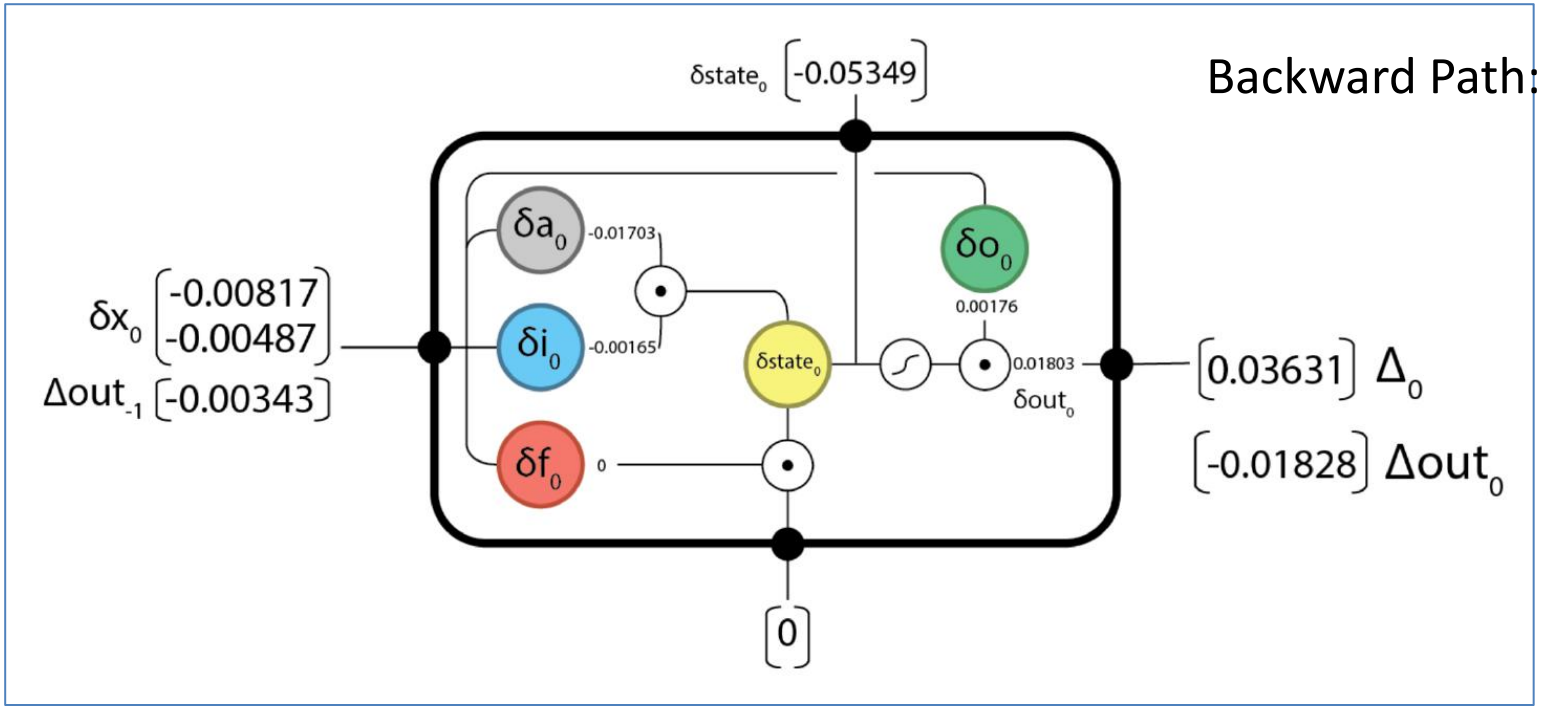
$$\delta out_0 = \Delta_0 + \Delta out_0 = 0.03631 + -0.01828 = 0.01803$$
$$\delta state_0 = \delta out_0 \odot o_0 \odot (1 - \tanh^2(state_0)) + \delta state_1 \odot f_1 = 0.01803 \times 0.81757 \times (1 - \tanh^2(0.78572)) + -0.07111 \times 0.87030 = -0.05349$$
$$\delta a_0 = \delta state_0 \odot i_0 \odot (1 - a_0^2) = -0.05349 \times 0.96083 \times (1 - 0.81775^2) = -0.01703$$
$$\delta i_0 = \delta state_0 \odot a_0 \odot i_0 \odot (1 - i_0) = -0.05349 \times 0.81775 \times 0.96083 \times (1 - 0.96083) = -0.00165$$
$$\delta f_0 = \delta state_0 \odot state_{-1} \odot f_0 \odot (1 - f_0) = -0.05349 \times 0 \times 0.85195 \times (1 - 0.85195) = 0$$
$$\delta o_0 = \delta out_0 \odot \tanh(state_0) \odot o_0 \odot (1 - o_0) = 0.01803 \times \tanh(0.78572) \times 0.81757 \times (1 - 0.81757) = 0.00176$$

$$\delta x_0 = W^T \cdot \delta gates_0$$
$$= \begin{bmatrix} 0.45 & 0.95 & 0.70 & 0.60 \\ 0.25 & 0.80 & 0.45 & 0.40 \end{bmatrix} \begin{bmatrix} -0.01703 \\ -0.00165 \\ 0 \\ 0.00176 \end{bmatrix} = \begin{bmatrix} -0.00817 \\ -0.00487 \end{bmatrix}$$

$$\Delta out_{-1} = U^T \cdot \delta gates_1$$
$$= \begin{bmatrix} 0.15 & 0.80 & 0.10 & 0.25 \end{bmatrix} \begin{bmatrix} -0.01703 \\ -0.00165 \\ 0 \\ 0.00176 \end{bmatrix} = -0.00343$$

https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9





$$\delta W = \sum_{t=0}^{T} \delta gates_t \otimes x_t$$

$$= \begin{bmatrix} -0.01703 \\ -0.00165 \\ 0 \\ 0.00176 \end{bmatrix} \begin{bmatrix} 1.0 & 2.0 \end{bmatrix} + \begin{bmatrix} -0.01938 \\ -0.00112 \\ -0.00631 \\ -0.05538 \end{bmatrix} \begin{bmatrix} 0.5 & 3.0 \end{bmatrix} = \begin{bmatrix} -0.02672 & -0.0922 \\ -0.00221 & -0.00666 \\ -0.00316 & -0.01893 \\ -0.02593 & -0.16262 \end{bmatrix}$$

$$\delta U = \sum_{t=0}^{T-1} \delta gates_{t+1} \otimes out_t$$

$$= \begin{bmatrix} -0.01938 \\ -0.00112 \\ -0.00631 \\ -0.05538 \end{bmatrix} \begin{bmatrix} 0.53631 \end{bmatrix} = \begin{bmatrix} -0.01039 \\ -0.00060 \\ -0.00338 \\ -0.02970 \end{bmatrix}$$

$$\delta b = \sum_{t=0}^{T} \delta gates_{t+1}$$

$$= \begin{bmatrix} -0.01703 \\ -0.00165 \\ 0 \\ 0.00176 \end{bmatrix} + \begin{bmatrix} -0.01938 \\ -0.00112 \\ -0.00631 \\ -0.05538 \end{bmatrix} = \begin{bmatrix} -0.03641 \\ -0.00277 \\ -0.00631 \\ -0.05362 \end{bmatrix}$$

$$W_a = \begin{bmatrix} 0.45267 \\ 0.25922 \end{bmatrix}, U_a = \begin{bmatrix} 0.15104 \end{bmatrix}, b_a = \begin{bmatrix} 0.20364 \end{bmatrix}$$

$$W_i = \begin{bmatrix} 0.95022 \\ 0.80067 \end{bmatrix}, U_i = \begin{bmatrix} 0.80006 \end{bmatrix}, b_i = \begin{bmatrix} 0.65028 \end{bmatrix}$$

$$W_f = \begin{bmatrix} 0.70031 \\ 0.45189 \end{bmatrix}, U_f = \begin{bmatrix} 0.10034 \end{bmatrix}, b_f = \begin{bmatrix} 0.15063 \end{bmatrix}$$

$$W_o = \begin{bmatrix} 0.60259 \\ 0.41626 \end{bmatrix}, U_o = \begin{bmatrix} 0.25297 \end{bmatrix}, b_o = \begin{bmatrix} 0.10536 \end{bmatrix}$$

$$W^{new} = W^{old} - \lambda * \delta W^{old}$$

https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9

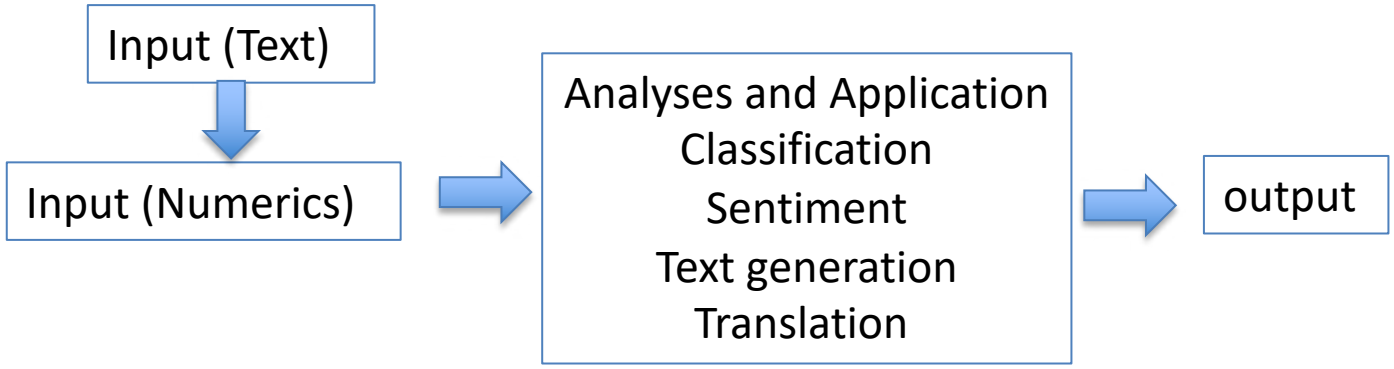
LOTS of MM (BLAS3), single precision

Content of Unit 11

- ✓ RNN,
- ✓ RNN CODES
- ✓ LSTM, CODES
- ✓ Natural Language Processing, NLP examples

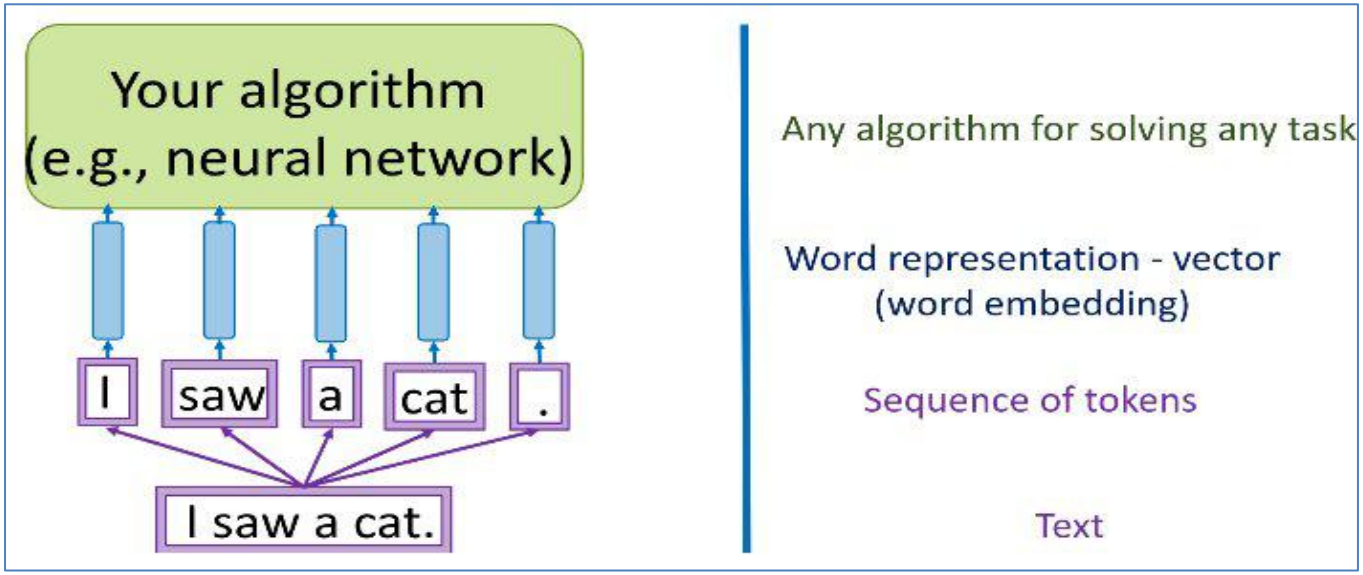
https://towardsdatascience.com/recurrent-neural-networks-by-example-in-python-ffd204f99470

https://www.youtube.com/playlist?list=PLQY2H8rRoyvzDbLUZkbudP-MFQZwNmU4S



https://www.tensorflow.org/tutorials/text/text_generation

Input : Word Representation – Represent a Word as a Vector



One Hot Encoding : too many words long vector, Lose order and frequency

Set of all the words in the corpus		R1: Great Restaurant and great service !	R2: They can do better to provide better service	R3: Only two thumbs up, worst service ever
great		1	0	0
restaurant		1	0	0
and		1	0	0
service		1	1	0
they		0	1	0
can		0	1	0
do		0	1	0
better		0	1	0
to		0	1	0
provide		0	1	0
only		0	0	1
Two		0	0	1
thumbs		0	0	1
up		0	0	1
worst		0	0	1
ever		0	0	1

Tokenization : convert a document to a sequence of words to numbers

Word Index = {'<OOV>': 1, 'my': 2, 'love': 3, 'dog': 4, 'i': 5, 'you': 6, 'cat': 7, 'do': 8, 'think': 9, 'is': 10, 'amazing': 11}

Sequences = [[5, 3, 2, 4], [5, 3, 2, 7], [6, 3, 2, 4], [8, 6, 9, 2, 4, 10, 11]]

Padded Sequences: [[0 5 3 2 4] [0 5 3 2 7] [0 6 3 2 4] [9 2 4 10 11]]

Test Sequence = [[5, 1, 3, 2, 4], [2, 4, 1, 2, 1]]

Padded Test Sequence:
[[0 0 0 0 0 5 1 3 2 4] [0 0 0 0 0 2 4 1 2 1]]

https://towardsdatascience.com/text-data-representation-with-one-hot-encoding-tf-idf-count-vectors-co-occurrence-vectors-and-f1bccbd98bef

https://www.kdnuggets.com/2018/11/data-representation-natural-language-processing.html

Word Embedding (Word2Vec)

Word embedding is the collective name for a set of language modeling and feature learning techniques in natural language processing (NLP) where words or phrases from the vocabulary are mapped to vectors of real numbers. Conceptually it involves a mathematical embedding from a space with many dimensions per word to a continuous vector space with a much lower dimension. Methods to generate this mapping include neural networks,[1] dimensionality reduction on the word co-occurrence matrix,[2][3][4] probabilistic models,[5] explainable knowledge base method,[6] and explicit representation in terms of the context in which words appear.[7] (from Wikipedia)

Word2vec is a technique for natural language processing. The word2vec algorithm uses a neural network model to learn word associations from a large corpus of text. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence. As the name implies, word2vec represents each distinct word with a particular list of numbers called a vector. The vectors are chosen carefully such that a simple mathematical function (the cosine similarity between the vectors) indicates the level of semantic similarity between the words represented by those vectors. (from Wikipedia)
Word2vec can utilize either of two model architectures to produce a distributed representation of words: continuous bag-of-words (CBOW) or continuous skip-gram. In the continuous bag-of-words architecture, the model predicts the current word from a window of surrounding context words. The order of context words does not influence prediction (bag-of-words assumption). In the continuous skip-gram architecture, the model uses the current word to predict the surrounding window of context words. The skip-gram architecture weighs nearby context words more heavily than more distant context words.[1][7] According to the authors' note,[8] CBOW is faster while skip-gram is slower but does a better job for infrequent words. (from Wekipedia)

Word2vec is implemented by shallow, two-layer neural networks that trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus assigned a corresponding vector in high dimension space. Similar words will have similar vectors in this high dimension space.[Cite:mikolov2013efficient]

https://www.youtube.com/watch?v=nWxtRlpObIs&list=PLjy4p-07OYzulelvJ5KVaT2pDlxivl_BN&index=58
https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

Word2Vec Code

```
from tensorflow.keras.utils import get_file

try:
    path = get_file('GoogleNews-vectors-negative300.bin.gz',
        origin='https://s3.amazonaws.com/dl4j-distribution/' +\
                    'GoogleNews-vectors-negative300.bin.gz')
except:
    print('Error downloading')
    raise

print(path)
```

```
import gensim

# Not that the path below refers to a location on my hard drive.
# You should download GoogleNews Vectors (see suggested software above)
model = gensim.models.KeyedVectors.load_word2vec_format(path, binary=True)
```

```
import numpy as np
w1 = model['king']
w2 = model['queen']
dist = np.linalg.norm(w1-w2) print(dist)
```

This shows the classic word2vec equation of **queen = (king - man) + female**

2.4796925

```
model.most_similar(positive=['woman', 'king'], negative=['man'])
```

```
[('queen', 0.7118192911148071),
('monarch', 0.6189674139022827),
('princess', 0.5902431607246399),
('crown_prince', 0.5499460697174072),
 ('prince', 0.5377321243286133),
('kings', 0.5236844420433044),
('Queen_Consort', 0.5235945582389832),
('queens', 0.5181134343147278),
('sultan', 0.5098593235015869),
('monarchy', 0.5087411999702454)]
```

Suggested Software for Word2Vec

✓ GoogleNews Vectors, GitHub Mirror, Python Gensim

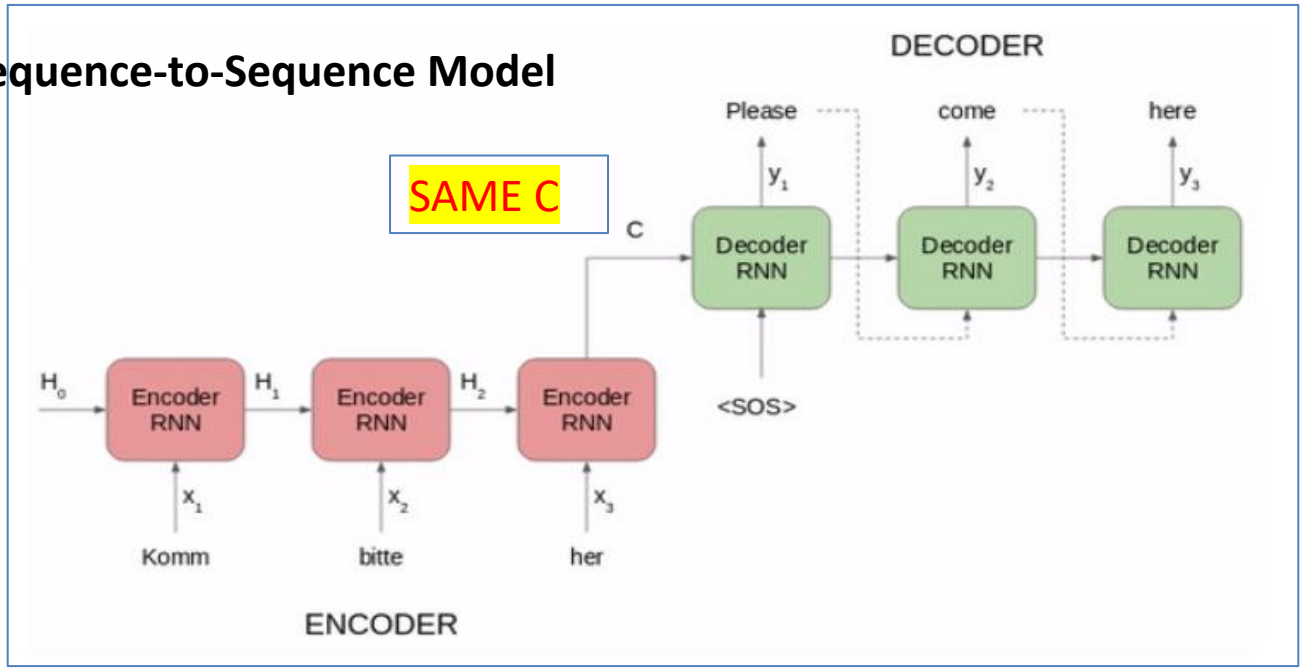
w = model['hello']

print(len(w)) = 300

print(w)

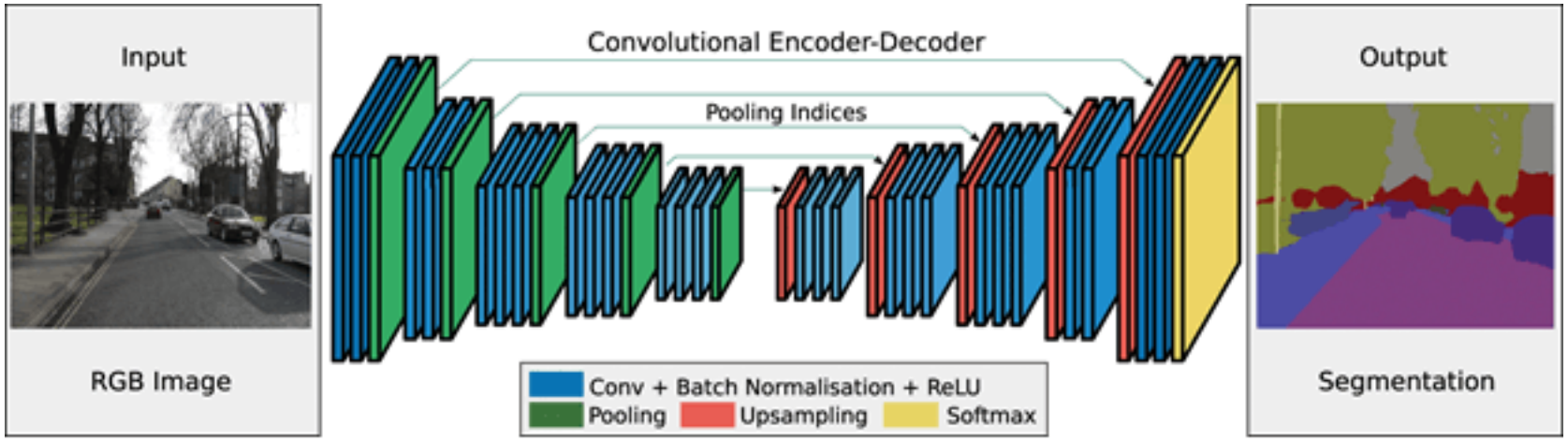
```
[-0.05419922  0.01708984 -0.00527954  0.33203125 -0.25       -0.01397705
 -0.15039062 -0.265625    0.01647949  0.3828125  -0.03295898 -0.09716797
 -0.16308594 -0.04443359  0.00946045  0.18457031  0.03637695  0.16601562
  0.36328125 -0.25585938  0.375       0.171875    0.21386719 -0.19921875
  0.13085938 -0.07275391 -0.02819824  0.11621094  0.15332031  0.09082031
  0.06787109 -0.0300293  -0.16894531 -0.20800781 -0.03710938 -0.22753906
  0.26367188  0.012146    0.18359375  0.31054688 -0.10791016 -0.19140625
  0.21582031  0.13183594 -0.03515625  0.18554688 -0.30859375  0.04785156
 -0.10986328  0.14355469 -0.43554688 -0.0378418   0.10839844  0.140625
 -0.10595703  0.26171875 -0.17089844  0.39453125  0.12597656 -0.27734375
 -0.28125     0.14746094 -0.20996094  0.02355957  0.18457031  0.00445557
 -0.27929688 -0.03637695 -0.29296875  0.19628906  0.20703125  0.2890625
 -0.20507812  0.06787109 -0.43164062 -0.10986328 -0.2578125  -0.02331543
  0.11328125  0.23144531 -0.04418945  0.10839844 -0.2890625  -0.09521484
 -0.10351562 -0.0324707   0.07763672 -0.13378906  0.22949219  0.06298828
  0.08349609  0.02929688 -0.11474609  0.00534058 -0.12988281  0.02514648
  0.08789062  0.24511719 -0.11474609 -0.296875   -0.59375     -0.29492188
 -0.13378906  0.27734375 -0.04174805  0.11621094  0.28320312  0.00241089
  0.13867188 -0.00683594 -0.30078125  0.16210938  0.01171875 -0.13867188
  0.48828125  0.02880859  0.02416992  0.04736328  0.05859375 -0.23828125
  0.02758789  0.05981445 -0.03857422  0.06933594  0.14941406 -0.10888672
 -0.07324219  0.08789062  0.27148438  0.06591797 -0.37890625 -0.26171875
 -0.13183594  0.09570312 -0.3125      0.10205078  0.03063965  0.23632812
  0.00582886  0.27734375  0.20507812 -0.17871094 -0.31445312 -0.01586914
  0.13964844  0.13574219  0.0390625  -0.29296875  0.234375   -0.33984375
 -0.11816406  0.10644531 -0.18457031 -0.02099609  0.02563477  0.25390625
  0.07275391  0.13574219 -0.00138092 -0.2578125  -0.2890625   0.10107422
  0.19238281 -0.04882812  0.27929688 -0.3359375  -0.07373047  0.01879883
 -0.10986328 -0.04614258  0.15722656  0.06689453 -0.03417969  0.16308594
  0.08642578  0.44726562  0.02026367 -0.01977539  0.07958984  0.17773438
 -0.04370117 -0.00952148  0.16503906  0.17285156  0.23144531 -0.04272461
  0.02355957  0.18359375 -0.41601562 -0.01745605  0.16796875  0.04736328
  0.14257812  0.08496094  0.33984375  0.1484375  -0.34375     -0.14160156
 -0.06835938 -0.14648438 -0.02844238  0.07421875 -0.07666016  0.12695312
  0.05859375 -0.07568359 -0.03344727  0.23632812 -0.16308594  0.16503906
  0.1484375  -0.2421875  -0.3515625  -0.30664062  0.00491333  0.17675781
  0.46289062  0.14257812 -0.25       -0.25976562  0.04370117  0.34960938
  0.05957031  0.07617188 -0.02868652 -0.09667969 -0.01281738  0.05859375
 -0.22949219 -0.1953125  -0.12207031  0.20117188 -0.42382812  0.06005859
  0.50390625  0.20898438  0.11230469 -0.06054688  0.33203125  0.07421875
 -0.05786133  0.11083984 -0.06494141  0.05639648  0.01757812  0.08398438
  0.13769531  0.2578125   0.16796875 -0.16894531  0.01794434  0.16015625
  0.26171875  0.31640625 -0.24804688  0.05371094 -0.0859375   0.17089844
 -0.39453125 -0.00156403 -0.07324219 -0.04614258 -0.16210938 -0.15722656
  0.21289062 -0.15820312  0.04394531  0.28515625  0.01196289 -0.26953125
 -0.04370117  0.37109375  0.04663086 -0.19726562  0.3046875  -0.36523438
 -0.23632812  0.08056641 -0.04248047 -0.14648438 -0.06225586 -0.0534668
 -0.05664062  0.18945312  0.37109375 -0.22070312  0.04638672  0.02612305
 -0.11474609  0.265625   -0.02453613  0.11083984 -0.02514648 -0.12060547
  0.05297852  0.07128906  0.00063705 -0.36523438 -0.13769531 -0.12890625]
```

RNN based Sequence-to-Sequence Model



German to English Translation using seq2seq

- ✓ Both **Encoder** and **Decoder** are RNNs
- ✓ The hidden state from the last unit is known as the **context vector.** This contains information about the input sequence



Non-linear transformation from an input sequence to an output sequence.

Justin Johnson of the University of Michigan : Deep Learning for Computer Vision

https://neptune.ai/blog/image-segmentation-in-2020

Sequence-to-Sequence with RNNs

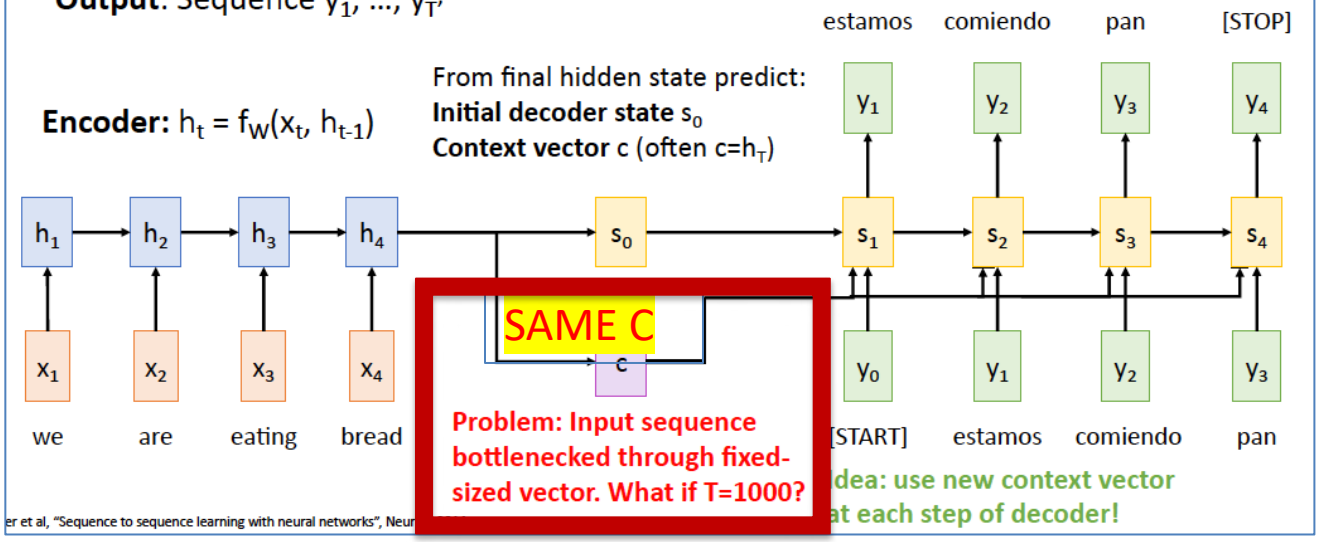
Input: Sequence $x_1, \dots x_T$

Output: Sequence $y_1, \dots, y_{T'}$

Encoder: $h_t = f_W(x_t, h_{t-1})$

Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

From final hidden state predict:
Initial decoder state s_0
Context vector c (often $c=h_T$)



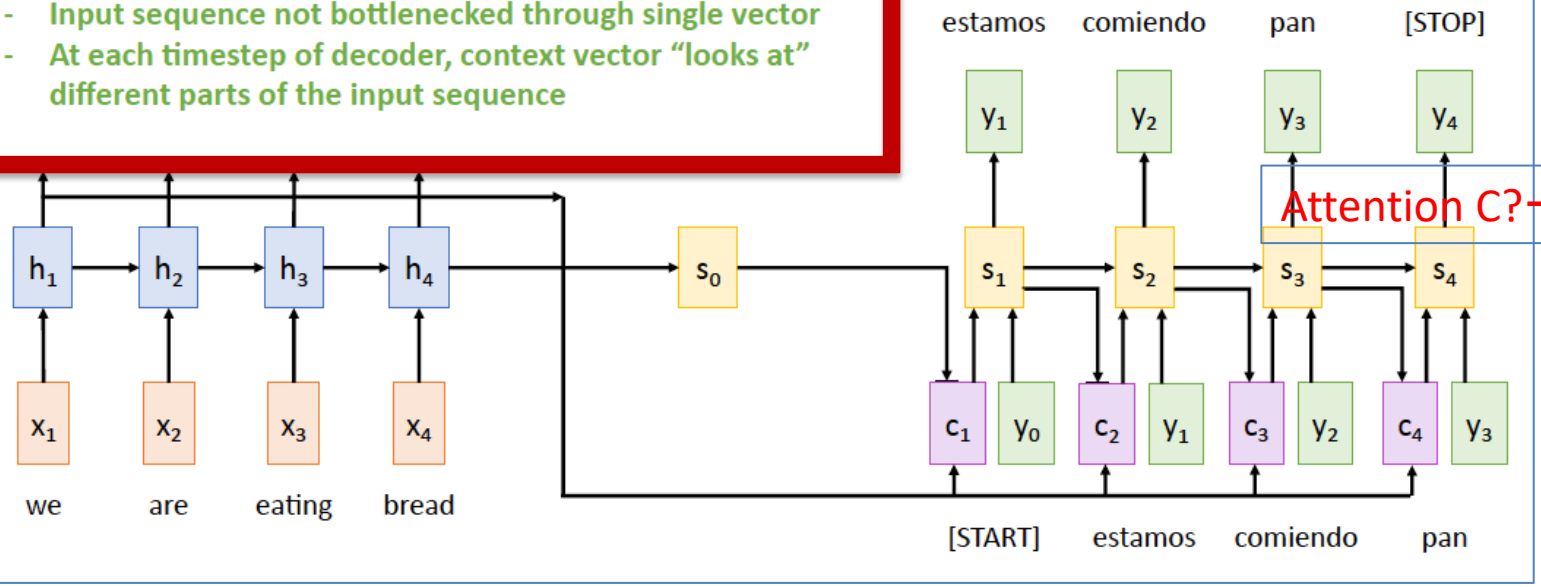
Problem: Input sequence bottlenecked through fixed-sized vector. What if T=1000?

Idea: use new context vector at each step of decoder!

er et al, "Sequence to sequence learning with neural networks", Neur

Use a different context vector in each timestep of decoder
- Input sequence not bottlenecked through single vector
- At each timestep of decoder, context vector "looks at" different parts of the input sequence

Seq-to-Seq with RNNs and Attention



Attention C? → Decoder ?

Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Summary of Various Versions of Attention Algorithms

$e_{t,i,j} = f_{att}(s_{t-1}, h_{i,j})$
$a_{t,:,:} = \text{softmax}(e_{t,:,:})$
$c_t = \sum_{i,j} a_{t,i,j} h_{i,j}$

Bahdanau et al (2015)

Alignment Scores eij:
fatt : Use MLP to train W and b
E=eij = tanh{W X(s,h) + b}

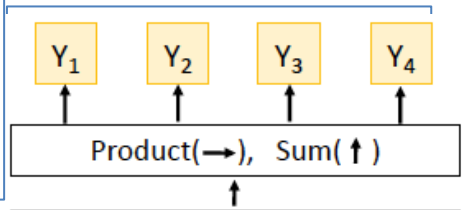
Inputs:
Query vector: q (Shape: D_Q)
Input vectors: X (Shape: N_X x D_X)
Similarity function: f_{att}

Computation:
Similarities: e (Shape: N_X) $e_i = f_{att}(q, X_i)$
Attention weights: a = softmax(e) (Shape: N_X)
Output vector: $y = \sum_i a_i X_i$ (Shape: D_X)

Generalize idea
Single Query Vector
Q = S; X = h

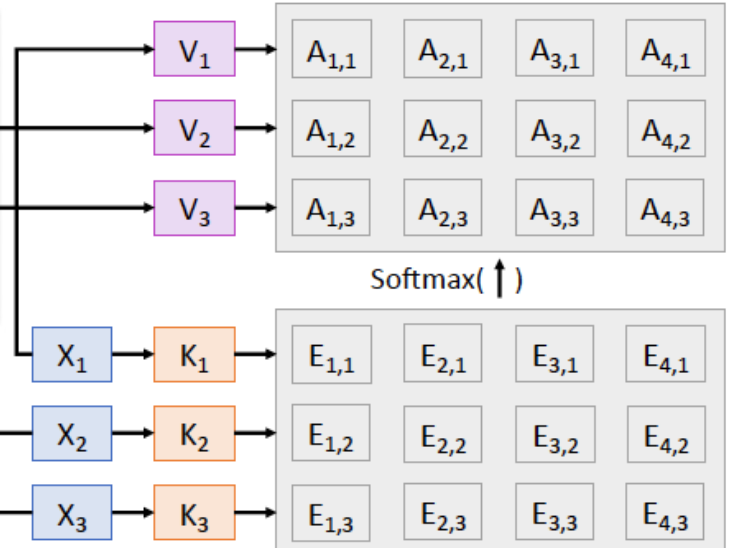
Inputs:
Query vector: q (Shape: D_Q)
Input vectors: X (Shape: N_X x D_Q)
Similarity function: scaled dot product

Computation:
Similarities: e (Shape: N_X) $e_i = q \cdot X_i / \text{sqrt}(D_Q)$
Attention weights: a = softmax(e) (Shape: N_X)
Output vector: $y = \sum_i a_i X_i$ (Shape: D_X)



Inputs:
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)

Separate Key and Value with learnable weights, Wk and Wv
Multiple Query vectors
Block Data :MM BLAS3



Computation:
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value Vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T / \sqrt{D_Q}$ (Shape: N_Q x N_X) $E_{i,j} = (Q_i \cdot K_j) / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = AV (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} V_j$

Lots of MM BLAS3

Self Attention, a special case case of the general attention layer : One query one input

"Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence."

Inputs:

Input vectors: X (Shape: N_X x D_X)

Key matrix: W_K (Shape: D_X x D_Q)

Value matrix: W_V (Shape: D_X x D_V)

Query matrix: W_Q (Shape: D_X x D_Q)

Input : A set of vector (Matrix)
Output : a set of vector (Matrix)
Add a query layer with Wq
Nonlinear Transformation :
Score for each input vector with another input vector in the input set

Computation:

Query vectors: $Q = XW_Q$

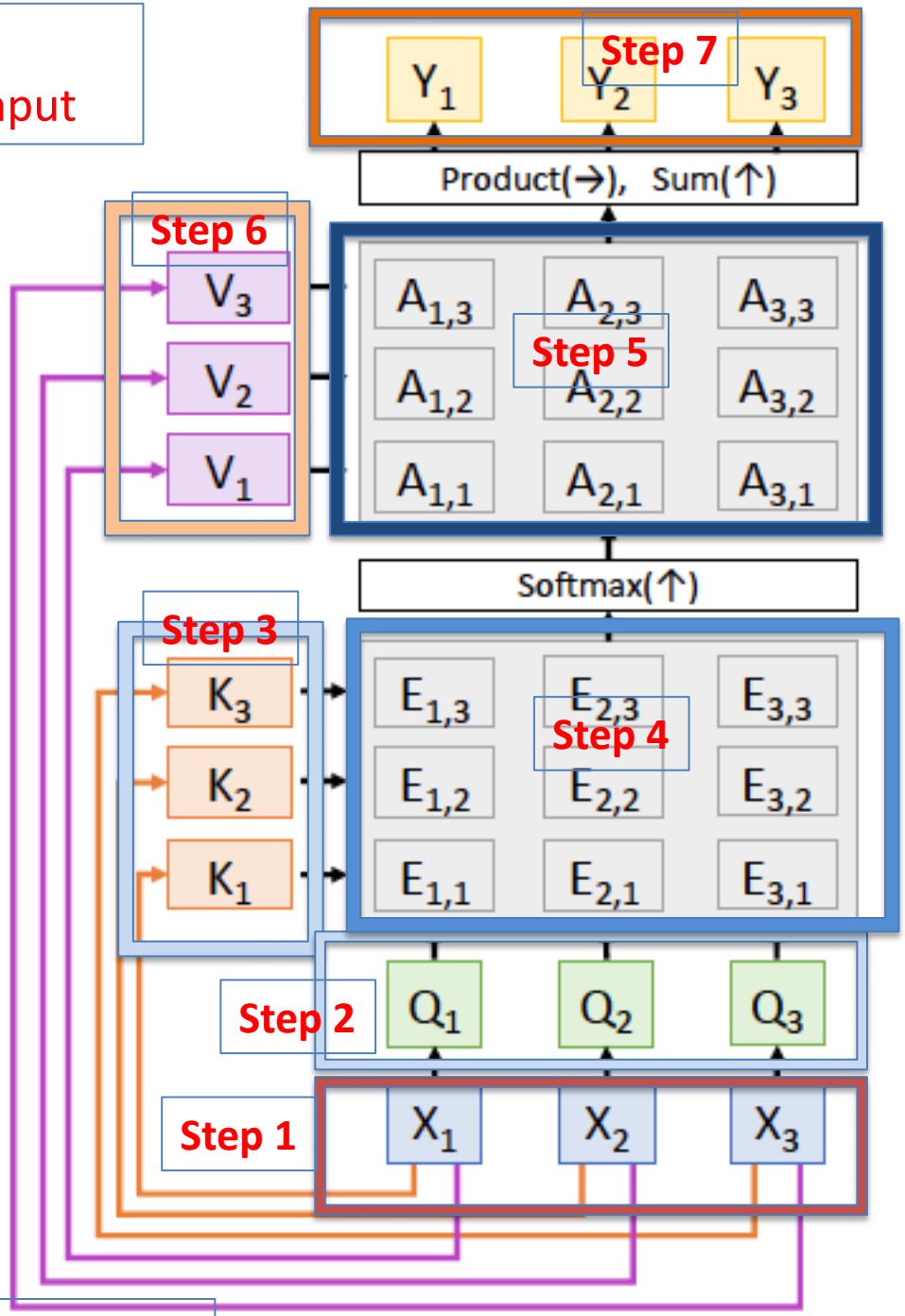
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)

Value Vectors: $V = XW_V$ (Shape: N_X x D_V)

Similarities: $E = QK^T / \sqrt{D_Q}$ (Shape: N_X x N_X) $E_{i,j} = (Q_i \cdot K_j) / \sqrt{D_Q}$

Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)

Output vectors: Y = AV (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



Multi-head Self-Attention

LOTS of MM (BLAS3), Parallel !!

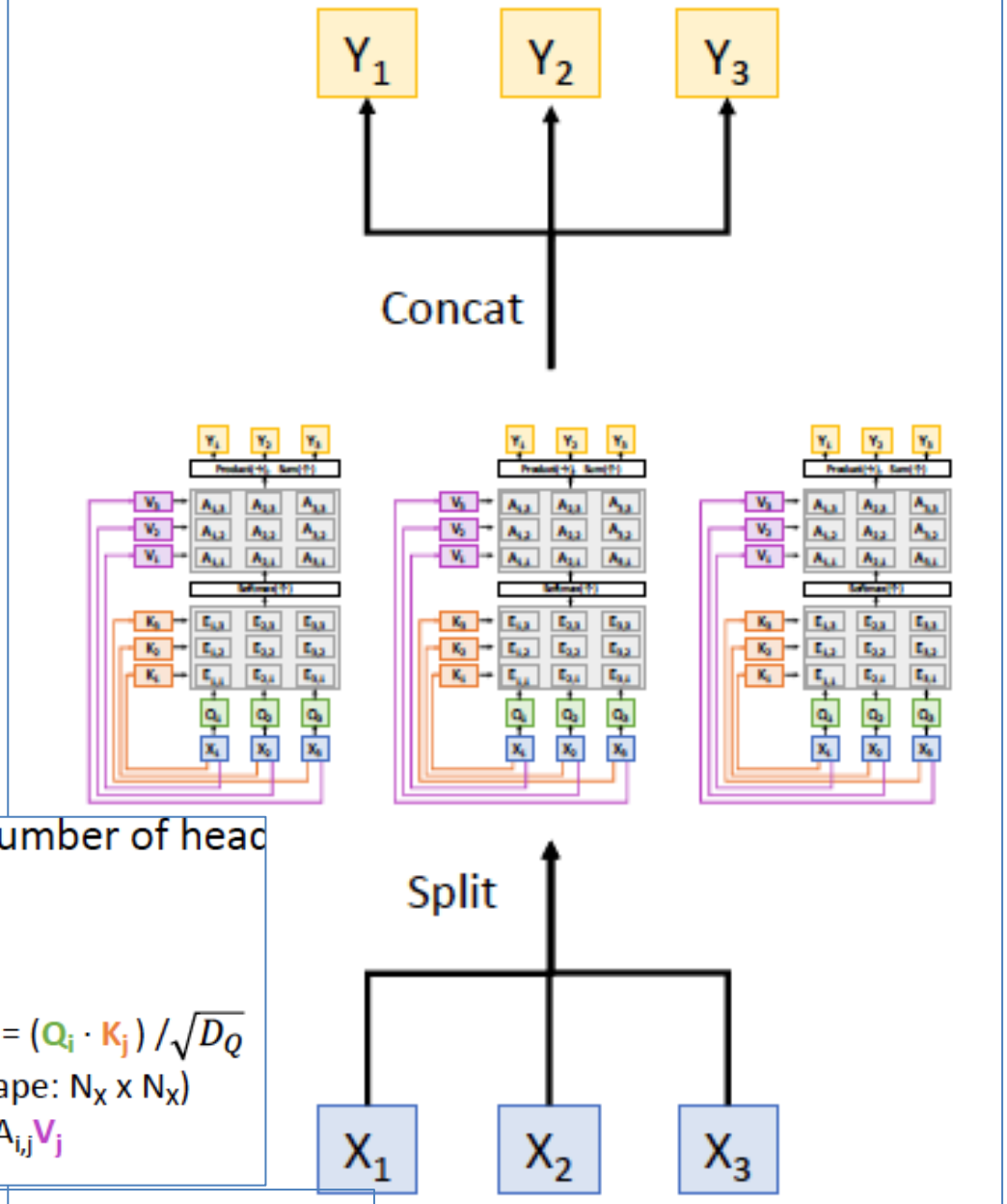
The text is split into a certain number of segments or chunks before being fed into the system as input

Multi-head Self-Attention Layer
Use H independent "Attention Heads" in parallel

Hyperparameters:
Query dimension Dq and
Number of heads, H

Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

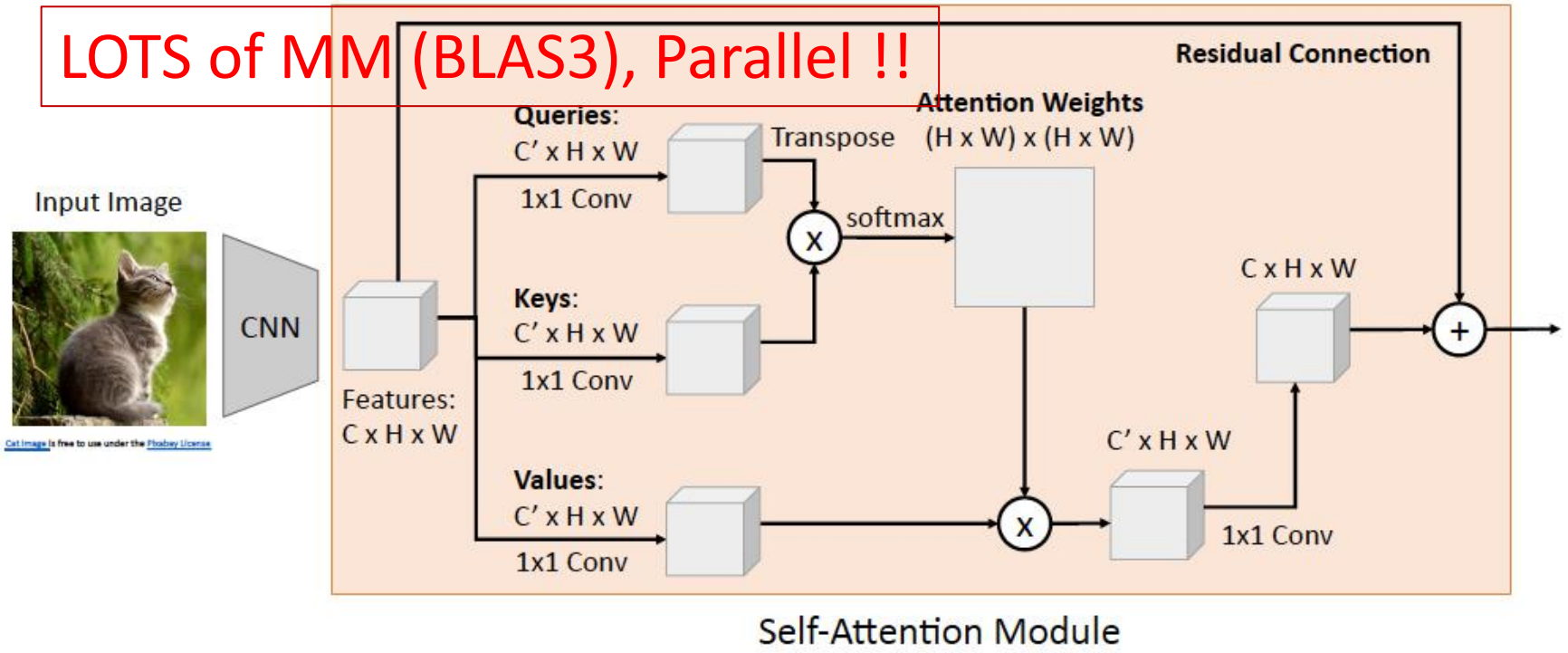
Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value Vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T / \sqrt{D_Q}$ (Shape: N_X x N_X) $E_{i,j} = (Q_i \cdot K_j) / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: Y = AV (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



Justin Johnson of the University of Michigan : Deep Learning for Computer Vision

Attention is ubiquities in Deep Learning

LOTS of MM (BLAS3), Parallel !!



Implicit Attention (feature) → Explicit Attention (algorithmic) → Soft Attention (continuous)

Attention vector →Bahdanau Attention (MLP) → General Attention (Q, K, V → Matrix)

General Attention (Q, K, V → W matrix) → Self Attention (X → Q) → Multihead Attention

Encoder → Multihead Attention → Decoder → Transformer for NLP

https://www.tensorflow.org/text/tutorials/nmt_with_attention

LOTS of MM (BLAS3), Parallel !!

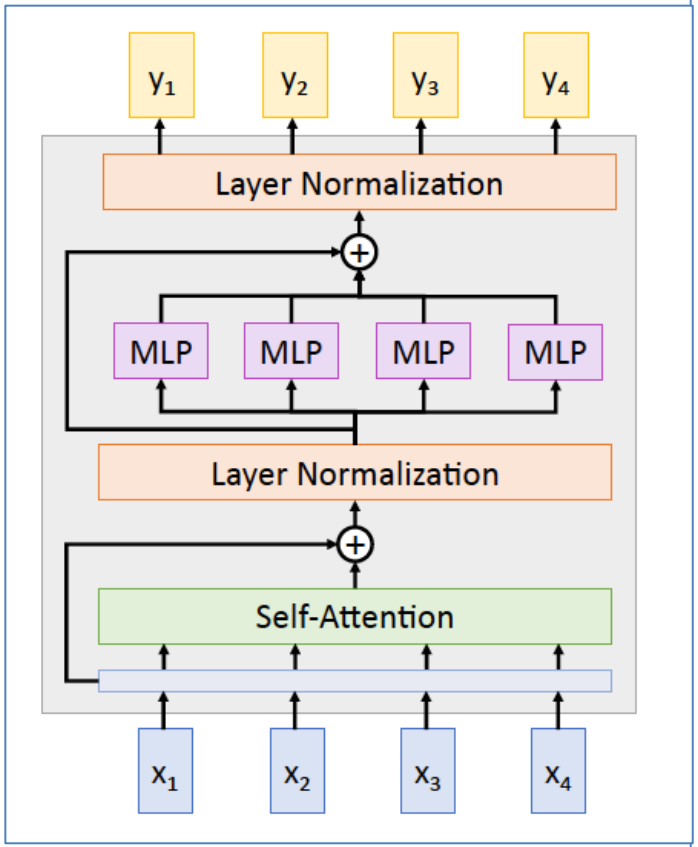
Transformer

Transformer Block:
Input: Set of vectors x
Output: Set of vectors y

Self-attention is the only interaction between vectors!

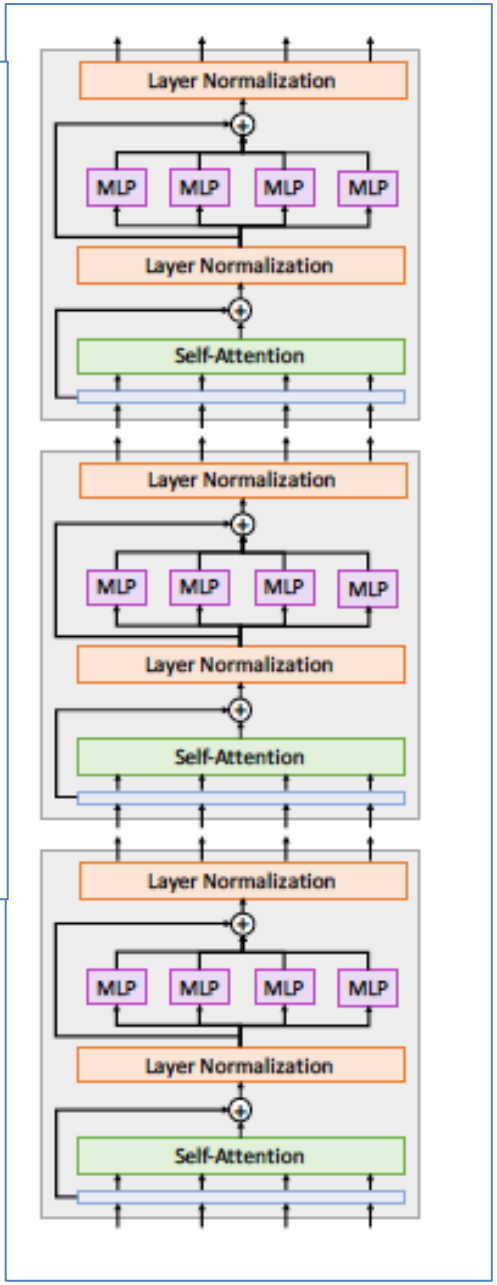
Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable



A **Transformer** is a sequence of transformer blocks

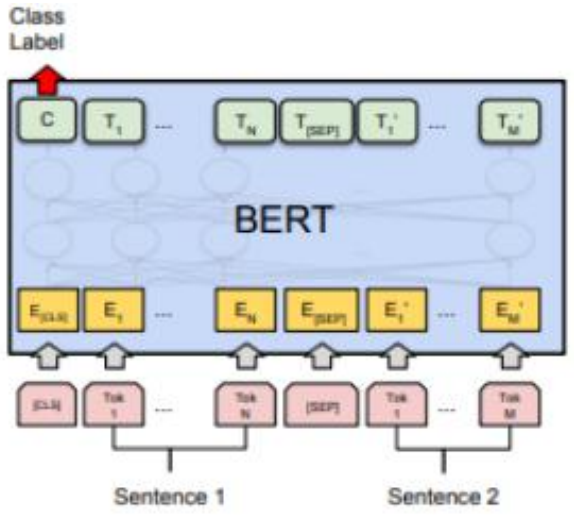
Vaswani et al:
12 blocks, D_Q=512, 6 heads



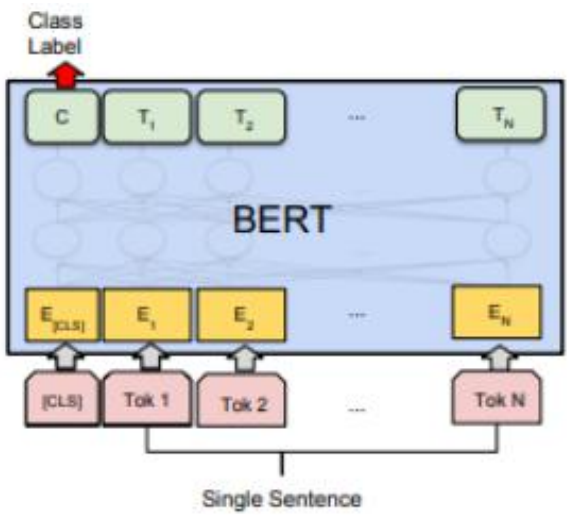
Google's BERT (Bidirectional Encoder Representations from Transformers)

The BERT framework, a new language representation model from Google AI, uses pre-training and fine-tuning to create state-of-the-art models for a wide range of tasks. These tasks include question answering systems, sentiment analysis, and language inference. Use the pre-trained model in transfer learning for NLP.

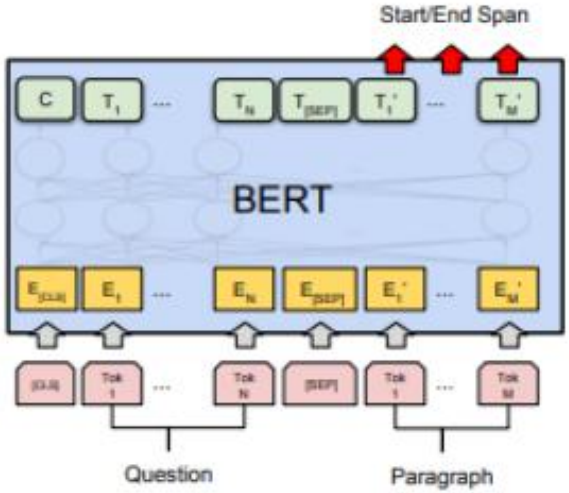
Source: https://arxiv.org/abs/1810.04805



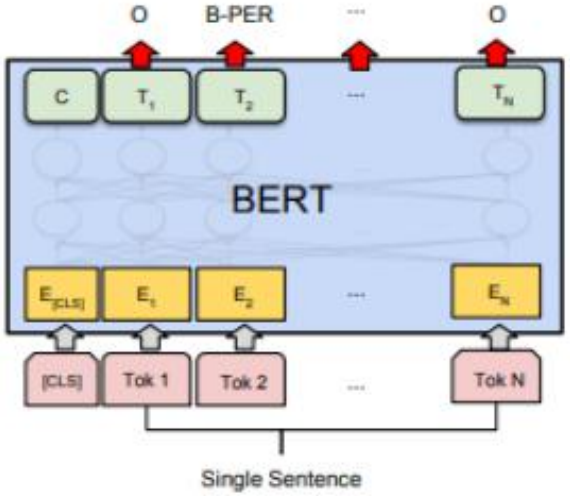
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA



(c) Question Answering Tasks: SQuAD v1.1

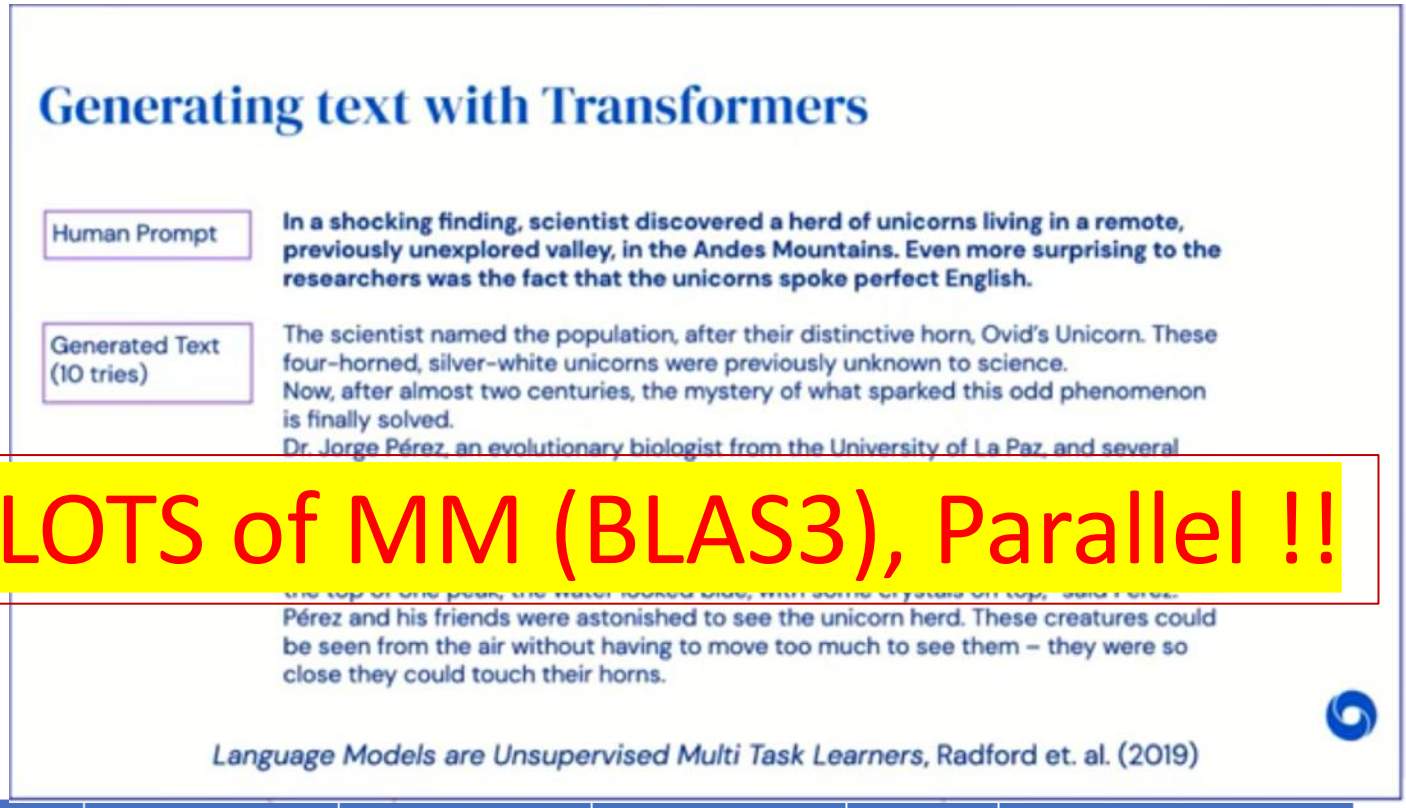


(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

https://www.analyticsvidhya.com/blog/2019/06/understanding-transformers-nlp-state-of-the-art-models/?utm_source=blog&utm_medium=comprehensive-guide-attention-mechanism-deep-learning

NLP Performance

Scaling to Bigger and larger transformer blocks from : Google, Openai, Nividia, FB,…



LOTS of MM (BLAS3), Parallel !!

Model	Layers	Width	Heads	Params	Data	Training
Transformer-Base	12	512	8	65M		8x P100 (12 hours)
Transformer-Large	12	1024	16	213M		8x P100 (3.5 days)
BERT-Base	12	768	12	110M	13 GB	
BERT-Large	24	1024	16	340M	13 GB	
XLNet-Large	24	1024	16	~340M	126 GB	512x TPU-v3 (2.5 days)
RoBERTa	24	1024	16	355M	160 GB	1024x V100 GPU (1 day)
GPT-2	48	1600	?	1.5B	40 GB	
Megatron-LM	72	3072	32	8.3B	174 GB	512x V100 GPU (9 days)
Turing-NLG	78	4256	28	17B	?	256x V100 GPU
GPT-3	96	12288	96	175B	694GB	?

Justin Johnson of the University of Michigan : Deep Learning for Computer Vision

DNN Model

Applications

+

Data Ensemble + Input

+

It's all about

linear algebra calculations

DNN in particular

+

Algorithms, Software

+

Output + Data Analysis

+

Hardware

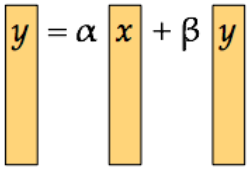
LOTS of MM (BLAS3), Parallel !!

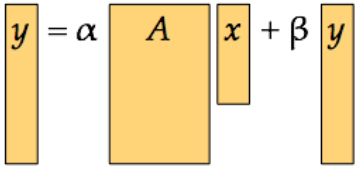
Basic Linear Algebra Subprograms (BLAS)

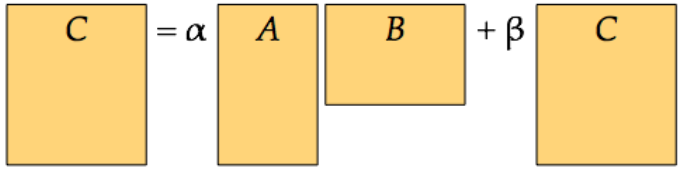
- **BLAS is a library of standardized basic linear algebra computational kernels created to perform efficiently on serial computers taking into account the memory hierarchy of modern processors.**
- **BLAS1 does vectors-vectors operations.**
  - **Saxpy = y(i) = a* x(i) + y(i), ddot= $\sum$x(i) *y(i)**
- **BLAS2 does matrices - vectors operations.**
  - **MV : y = A x + b (dgemv)**
- **BLAS3 operates on pairs or triples of matrices.**
  - **MM : C = αAB + βC, Triangular Solve : X = $\alpha T^{-1}X$**
- **Level3 BLAS is created to take full advantage of the fast cache memory. Matrix computations are arranged to operate in block fashion. Data residing in cache are reused by small blocks of matrices. dgemm**
- **Atlas, openBLAS, MKL, ESSL, libsci, ACML, cuBLAS, BLIS**

Computational Intensity = FLOPS/ Memory Access

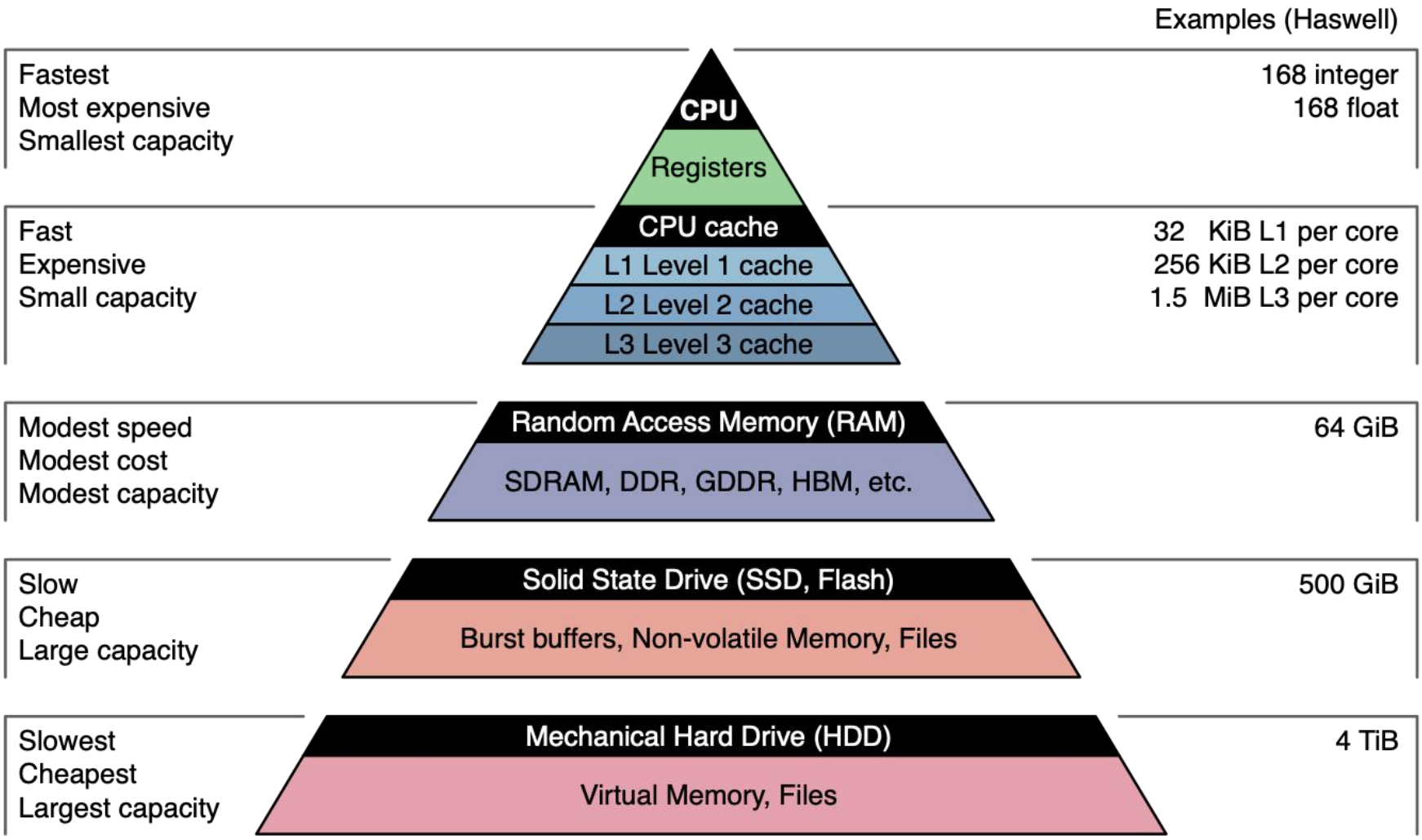
- ✓ Level 1 BLAS — vector operations
  - ✓ $O(n)$ data and flops (floating point operations)
  - ✓ Memory bound:
    $O(1)$ flops per memory access
- ✓ Level 2 BLAS — matrix-vector operations
  - ✓ $O(n^2)$ data and flops
  - ✓ Memory bound:
    $O(1)$ flops per memory access
- ✓ Level 3 BLAS — matrix-matrix operations
  - ✓ $O(n^2)$ data, $O(n^3)$ flops
  - ✓ Surface-to-volume effect
  - ✓ Compute bound:
    $O(n)$ flops per memory access







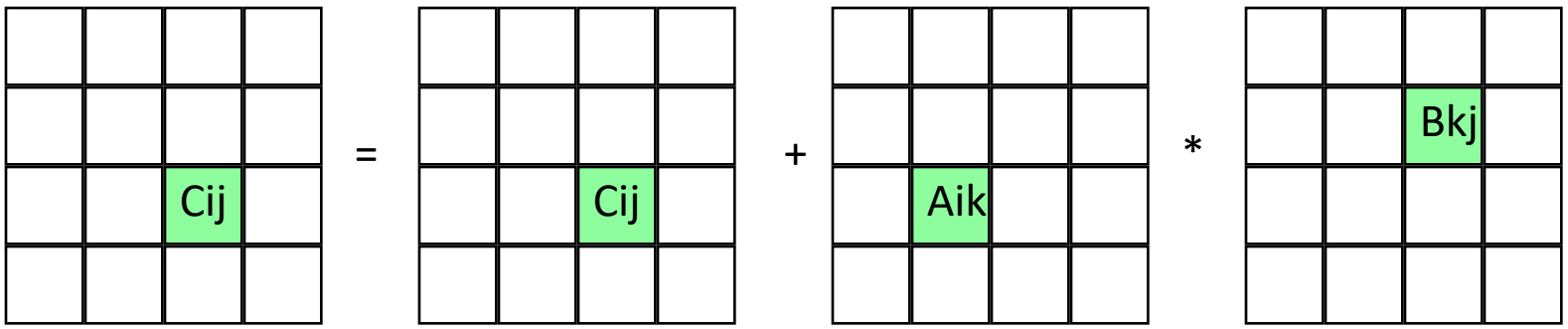
Memory hierarchy



Adapted from illustration by Ryan Leng

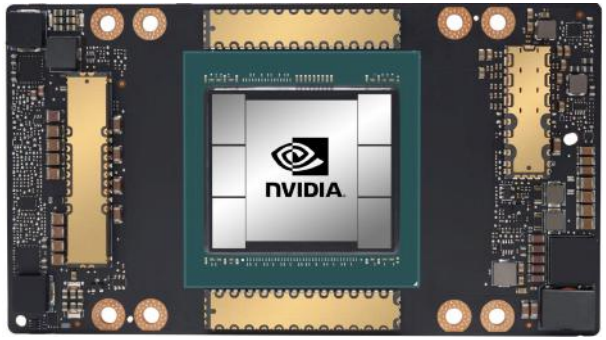
It's about
Block matrix access

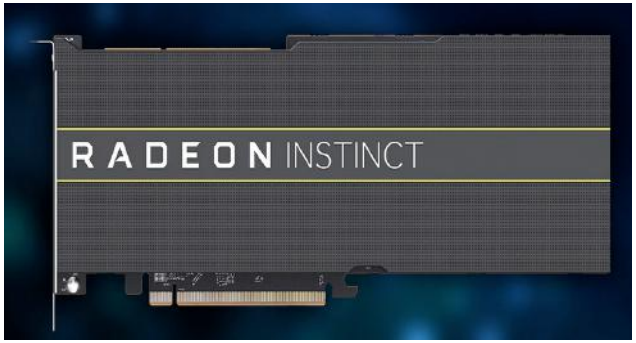
- q = f/m = (2*n^3) / ((2*N + 2) * n^2) ~ n / N
- If N is equal to 1, the algorithm is ideal. However, N is bounded by the amount of fast cache memory. However, N can be taken independently to the size of matrix, n.
- The optimal value of N = sqrt (size of fast memory / 3)



$$Cij = Cij + \sum_{k=1}^{n} Aik * Bkj$$

Software Stack : Single Precision Computation





nvcc, openACC, CUDA
cuBLAS, cuDNN, NSIGHT,
grid, block, thread, warp

hipcc, openMP, HIP, rocprofiler
hipBLAS, Tensile (GEMM), ROCm
Grid, block, thread, wavefront

Intel, openMP,DPC++, oneAPI,
Vtune, sycl, oneMKL, , oneDNN.

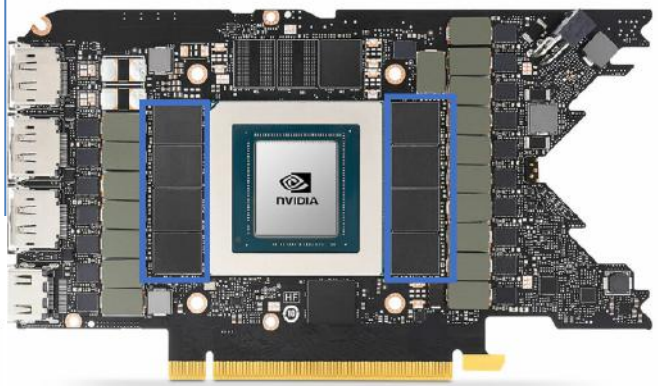
	bits	precision		epsilon (2 × unit roundoff)	underflow (min normal)	overflow (max)
bfloat16	16	8 bits	≈ 2 digits	7.81×10^{-3}	1.18×10^{-38}	3.40×10^{38}
half	16	11 bits	≈ 3 digits	9.77×10^{-4}	6.10×10^{-5}	65504
single (float)	**32**	**24 bits**	**≈ 7 digits**	$\mathbf{1.19 \times 10^{-7}}$	$\mathbf{1.18 \times 10^{-38}}$	$\mathbf{3.40 \times 10^{38}}$
double	**64**	**53 bits**	**≈ 16 digits**	$\mathbf{2.22 \times 10^{-16}}$	$\mathbf{2.23 \times 10^{-308}}$	$\mathbf{1.79 \times 10^{308}}$
double-double	64x2	107 bits	≈ 32 digits	1.23×10^{-32}	2.23×10^{-308}	1.79×10^{308}
quad	128	113 bits	≈ 34 digits	1.93×10^{-34}	3.36×10^{-4932}	1.19×10^{4932}

Google TPU implements bfloat16, NVIDIA implements half, quad is double double

Inside a computer

CPU: "Central Processing Unit"

GPU: "Graphics Processing Unit"



CPU vs GPU

Nvidia RTX3090 GPU system

	Cores	Clock Speed (GHz)	Memory	Price	TFLOP/sec
CPU Ryzen Threadripper 3970X	64 (128 threads with hyperthreading)	3.7 (4.5 boost)	System RAM	$1999	~6.9 FP32
GPU NVIDIA RTX 3090	10496	1.4 (1.7 boost)	24 GB GDDR6X	$1499	~35.6 FP32 ~142 TFLOP with Tensor core

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: More cores, but each core is much slower and "dumber"; great for parallel tasks

https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2020/schedule.html

Nvidia DGX A100 (H100) Box: A GPU cluster

Single Box = DP 9.7x8=77.6 TF (155.3 TF TC),
TF32 TC: 312x8=2.5PFLOPS; FP16 TC : 2.5 PF x2 – 5 PF AI

GPU : 32bit ML : Best performance / cost

SYSTEM SPECIFICATIONS

GPUs	8x NVIDIA A100 Tensor Core GPUs
GPU Memory	320 GB total
Performance	5 petaFLOPS AI 10 petaOPS INT8
NVIDIA NVSwitches	6
System Power Usage	6.5kW max
CPU	Dual AMD Rome 7742, 128 cores total, 2.25 GHz (base), 3.4 GHz (max boost)
System Memory	1TB
Networking	8x Single-Port Mellanox ConnectX-6 VPI 200Gb/s HDR InfiniBand 1x Dual-Port Mellanox ConnectX-6 VPI 10/25/50/100/200Gb/s Ethernet
Storage	OS: 2x 1.92TB M.2 NVME drives Internal Storage: 15TB (4x 3.84TB) U.2 NVME drives
Software	Ubuntu Linux OS
System Weight	271 lbs (123 kgs)
Packaged System Weight	315 lbs (143kgs)
System Dimensions	Height: 10.4 in (264.0 mm) Width: 19.0 in (482.3 mm) MAX Length: 35.3 in (897.1 mm) MAX
Operating Temperature Range	5°C to 30°C (41°F to 86°F)

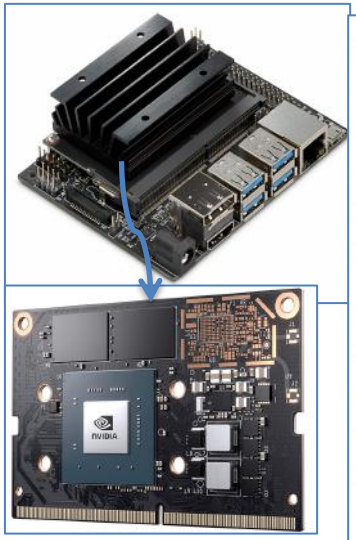
GPU on supercomputers

NVIDIA Volta100 on Summit

- ✓ Number of Nodes: 4,608
- ✓ Node performance: 42 TF
- ✓ Memory per Node: 512 GB DDR4 + 96 GB HBM2
- ✓ NV memory per Node: 1600 GB
- ✓ Processors:
  - ✓ 2 IBM POWER9™ 9,216 CPUs
  - ✓ 6 NVIDIA Volta(27,648 GPUs)

GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS
Memory Bandwidth	900GB/sec	

GPU architectures



DEVELOPER KIT	
GPU	128-core Maxwell
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	microSD (not included)
Video Encoder	4K @ 30 | 4x 1080p @ 30 | 9x 720p @ 30 (H.264/H.265)
Video Decoder	4K @ 60 | 2x 4K @ 30 | 8x 1080p @ 30 | 18x 720p @ 30| (H.264/H.265)
Camera	1x MIPI CSI-2 DPHY lanes
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI 2.0 and eDP 1.4
USB	4x USB 3.0, USB 2.0 Micro-B
Others	GPIO, I²C, I²S, SPI, UART
Mechanical	100 mm x 80 mm x 29 mm

Jetson Nano Developer Kit

Nsight Systems	2019.3
Nsight Graphics	2018.7
Nsight Compute	1.0
Jetson GPIO	1.0
Jetson OS	Ubuntu 18.04
CUDA	10.0.166
cuDNN	7.3.1.28
TensorRT	5.0.6.3

Other NVIDIA GPUs used in this workshop: GTX 1650 , K80, P100, V100, A100, ..

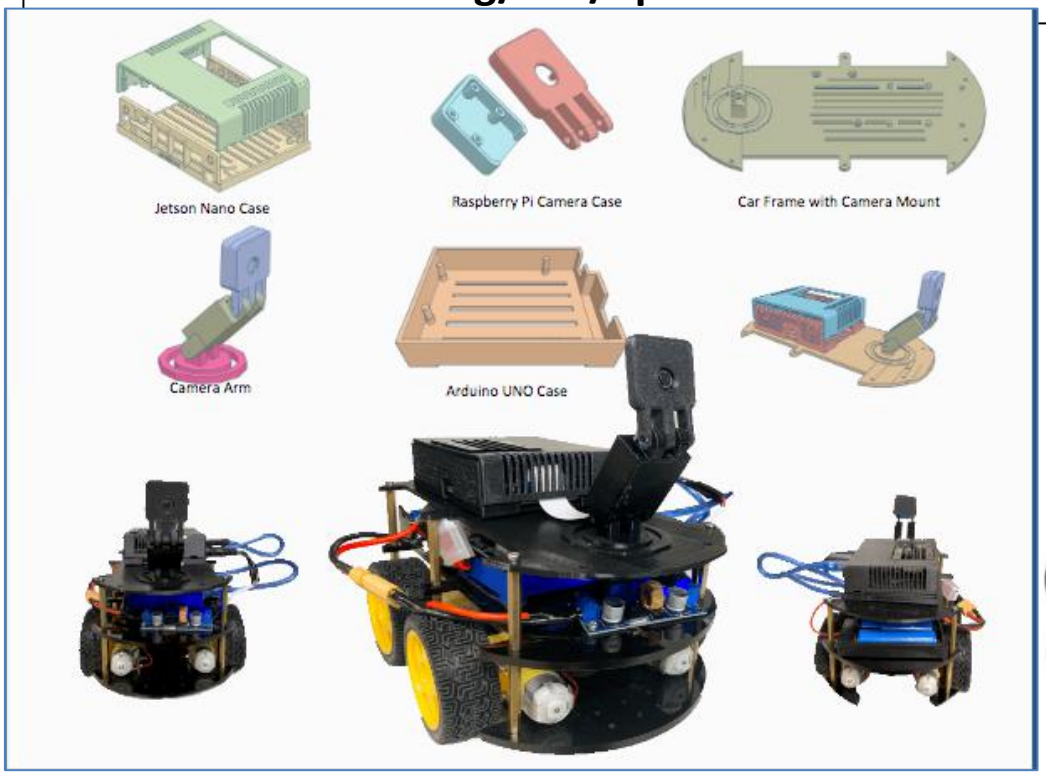
PRODUCT SPECIFICATIONS	
NVIDIA® CUDA Cores	896
Clock Speed	1485 MHz
Boost Speed	1725 MHz
Memory Speed (Gbps)	8
Memory Size	4GB GDDR5
Memory Interface	128-bit
Memory Bandwidth (Gbps)	128

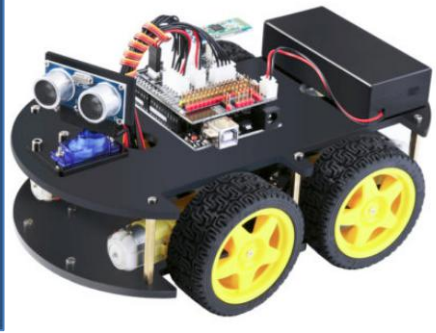
NVIDIA Tesla K80

MEMORY SIZE PER BOARD	24 GB GDDR5 (12 GB per GPU)
MEMORY INTERFACE	384-bit
MEMORY BANDWIDTH	480 Gb/s
CUDA CORES	4992
PEAK DOUBLE PRECISION FLOATING POINT PERFORMANCE	2.91 Tflops (GPU Boost Clocks) 1.87 Tflops (Base Clocks)
PEAK SINGLE PRECISION FLOATING POINT PERFORMANCE	8.74 Tflops (GPU Boost Clocks) 5.6 Tflops (Base Clocks)

Desktop Computers and GPU

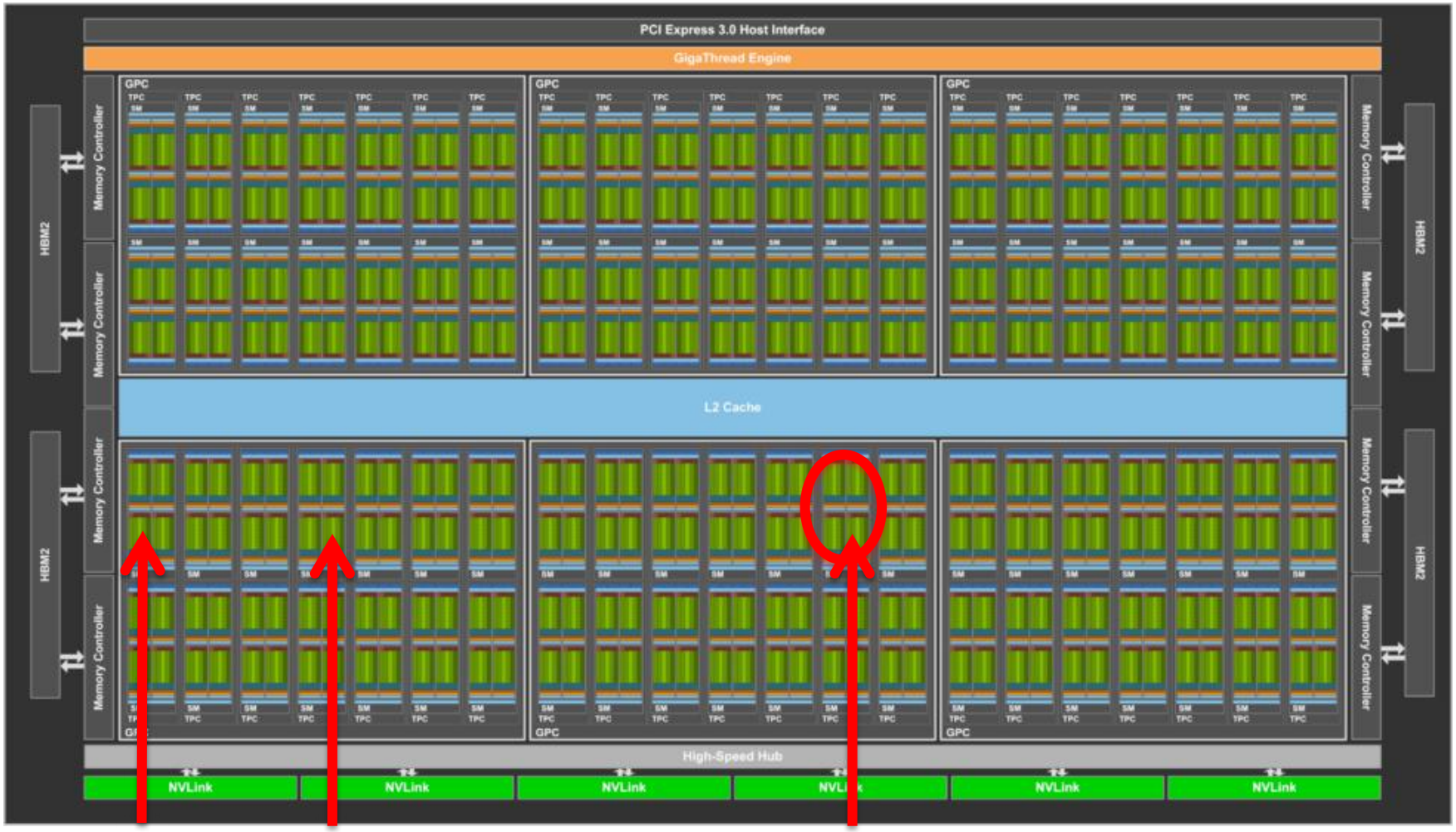
- ✓ **Computer Access, needs of each team, Desktop, laptop, Jetson Nano, Car**
- ✓ **Desktops configuration and deliveries, mid September**
- ✓ **Dell Server Box 12 cores, 48G Ram, 1TB Disk, 1650 super GPU card**
- ✓ **Ubuntu OS, anaconda individual, https://www.anaconda.com/products/individual**
- ✓ **Download, Linux,** 64-Bit (x86) Installer (550 MB), python 3.8
- ✓ **Jetson Nano card and car kit, late October**
- ✓ **www.bitbucket.org/cfdl/opendnnwheel**





GPU architectures

nvidia-tesla-v100-gpu-details



Streaming Multiprocessors (SMs) **32 cores**

https://devblogs.nvidia.com/inside-volta/image3-3/

Streaming Multiprocessor (SM) - V100

SM is the heart of the GPU architecture in which GPU uses as the basic building blocks. It consists of 4 sub-cores.

- A FP32 core is the execution unit that performs single precision floating point arithmetic (floats).
- A FP64 core performs double precision arithmetic (doubles).
- A INT32 Core performs integer arithmetic.
- The warp scheduler selects which warp (group of 32 threads) to send to which execution units.
- 64 KB Register File –lots of transistors used for vary fast memory.
- 128KB of configurable L1 (data) cache or shared memory
- Shared memory is a used managed cache.
- LD/ST units load and store data to/from cores.
- SFU–special function units compute things like transcendentals.

In Volta 100, each SM is partitioned into four processing blocks, each with:

- ✓ 16 FP32 Cores;
- ✓ 8 FP64 Cores;
- ✓ 16 INT32 Cores;
- ✓ 128KB L1 / Shared memory;
- ✓ 64 KB Register File.

Ampere A100

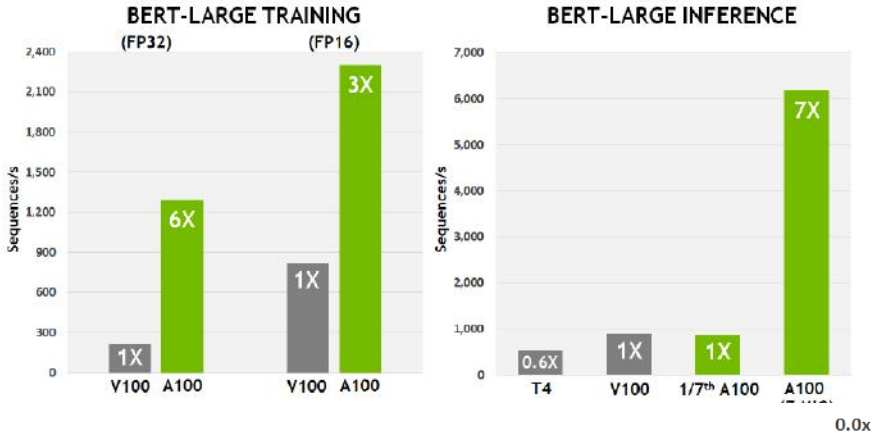
54B transistors
826 mm²

108 SM
6912 CUDA Cores
432 Tensor Cores

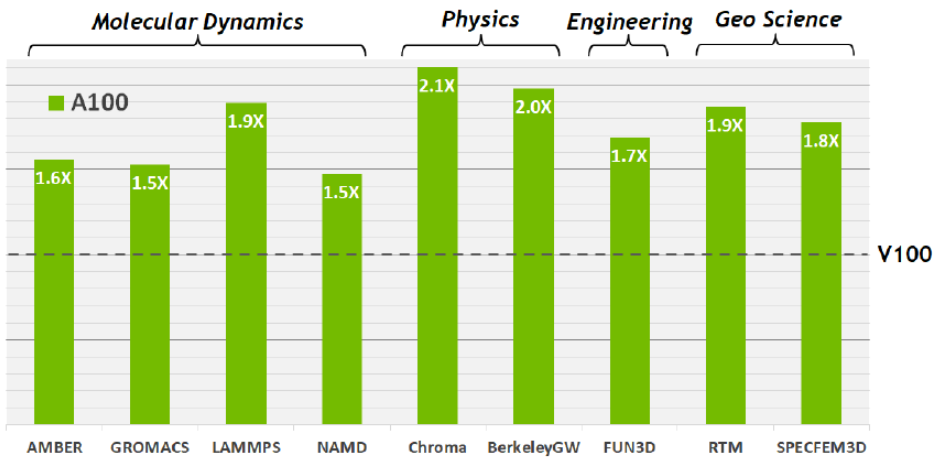
40 GB HBM2
1555 GB/s HBM2
600 GB/s NVLink

7 NVIDIA.

UNIFIED AI ACCELERATION



ACCELERATING HPC



https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

A100 SM GPU

FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	4
Register File	256 KB
Unified L1/Shared memory	192 KB
Active Threads	2048

The full implementation of the GA100 GPU includes the following units:

- **8 GPCs, 8 TPCs/GPC, 2 SMs/TPC, 16 SMs/GPC, 128 SMs per full GPU**
- **64 FP32 CUDA Cores/SM, 8192 FP32 CUDA Cores per full GPU**
- **4 Third-generation Tensor Cores/SM, 512 Third-generation Tensor Cores per full GPU**
- **6 HBM2 stacks, 12 512-bit Memory Controllers**

The NVIDIA A100 Tensor Core GPU of the GA100 GPU includes the following units:

- **7 GPCs, 7 or 8 TPCs/GPC, 2 SMs/TPC, up to 16 SMs/GPC, 108 SMs**
- **64 FP32 CUDA Cores/SM, 6912 FP32 CUDA Cores per GPU**
- **4 Third-generation Tensor Cores/SM, 432 Third-generation Tensor Cores per GPU**
- **5 HBM2 stacks, 10 512-bit Memory Controllers**

Tensor Core in A100

Specialized hardware execution units

- ✓ Perform matrix and convolution operations, which represent most fundamental and time-consuming operations for most DL workloads

Scalar vs matrix instructions

- ✓ FP32 cores perform scalar instructions: multiplication of an element of A with an element of B
- ✓ Tensor Cores perform matrix instructions: multiplication between vectors/matrix of elements at a time

Compared to scalar FP32 operations, Tensor Cores are:

- ✓ 8-16x faster (up to 32x faster with sparsity) and more energy efficient

TensorFloat(TF32) model for single-precision training (A100):

- ✓ Accelerates only math-limited operations
- ✓ Compared to FP32 training, **8x** higher math throughput, same memory bandwidth pressure
- ✓ Does not require any changes to training scripts

Default math mode for single-precision training on NVIDIA Ampere GPU Architecture

16-bit formats for mixed-precision training (V100 or A100):

- ✓ Fastest option: accelerate math-and memory-limited operations
- ✓ Compared to FP32 training: **16x**higher math throughput, **0.5x**memory bandwidth pressure
- ✓ Requires some changes to training scripts: fp32 master weights, layer selection, loss-scaling, Automatic Mixed Precision (AMP) reduces these changes to just a few lines (TF, PyT, MxNet)

Training Neural Networks with Tensor Cores - Dusan Stosic, NVIDIA

https://nvlabs.github.io/eccv2020-mixed-precision-tutorial/

https://www.youtube.com/watch?v=jF4-_ZK_tyc

https://www.youtube.com/watch?v=xjjN9q2ym6s

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7959640/pdf/peerj-cs-07-330.pdf

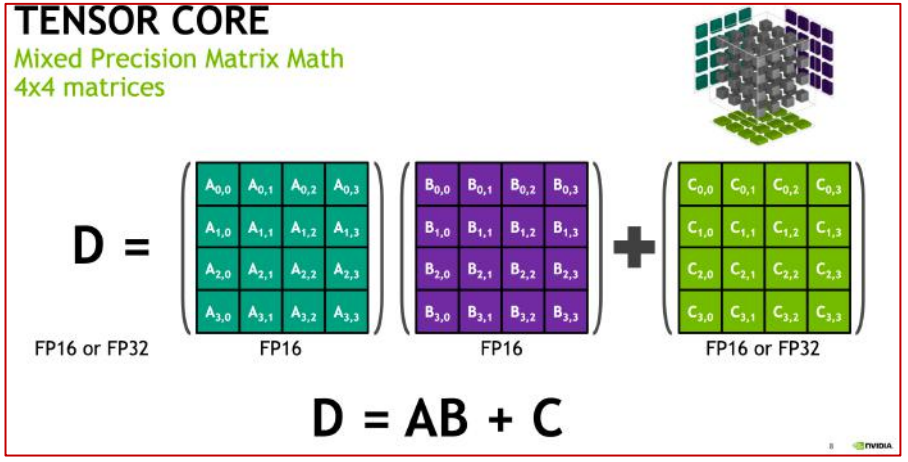
All About MM in SM V100, (A100)

General Matrix-Matrix Multiplication (GEMM) operations are at the core of neural network training and inference, and are used to multiply large matrices of input data and weights in various layers. The GEMM operation computes the matrix product *D = A * B + C*, where *C* and *D* are *m*-by-*n* matrices, *A* is an *m*-by-*k* matrix, and *B* is a *k*-by-*n* matrix. The problem size of such GEMM operations running on Tensor Cores is defined by the matrix sizes, and typically denoted as *m*-by-*n*-by-*k*, (4x4x4) or (8x4x8).

FMA = Fused Multiple-add, compute 2 floating point

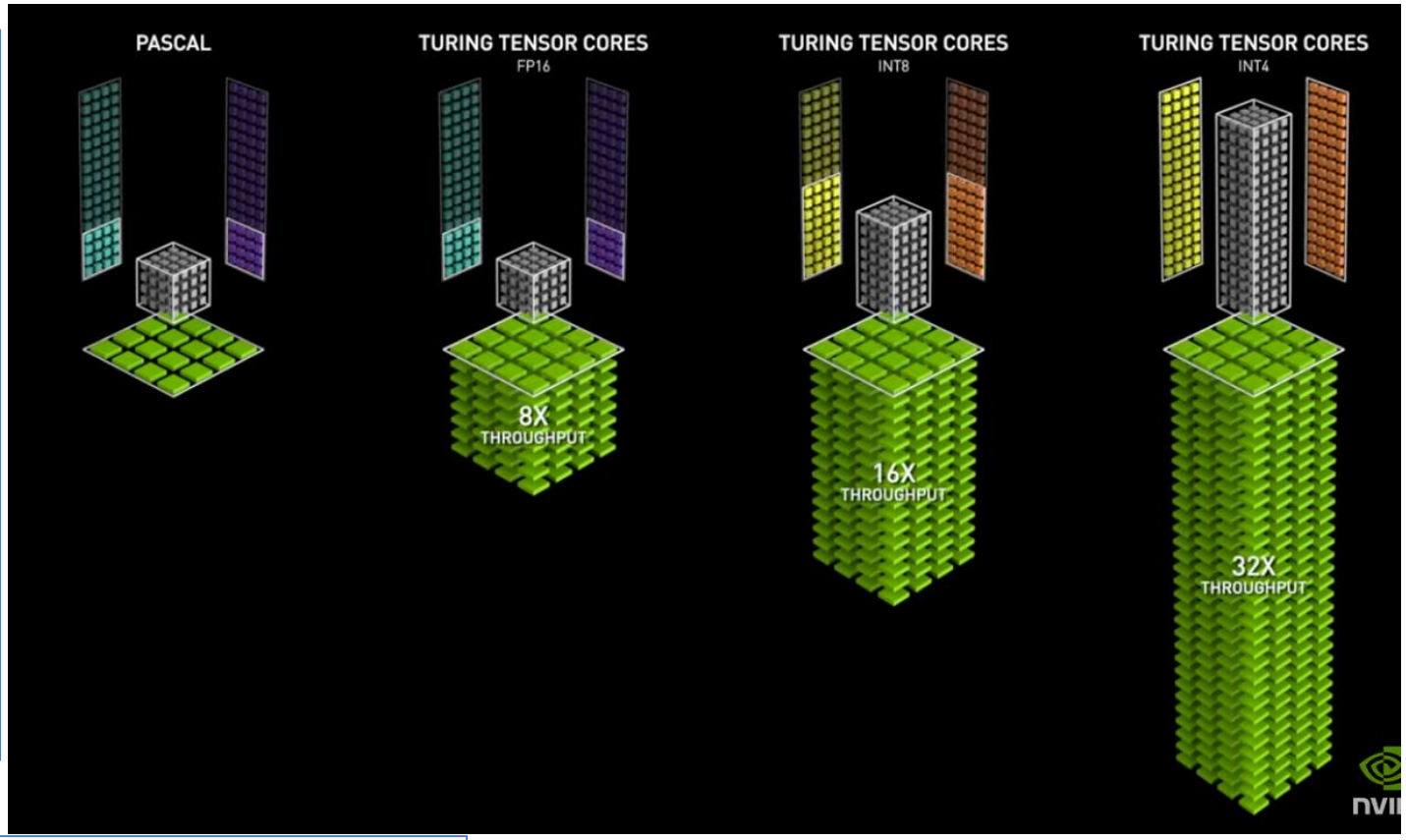
	V100	A100	A100 Sparsity[1]	A100 Speedup	A100 Speedup with Sparsity
A100 FP16 vs V100 FP16	31.4 TFLOPS	78 TFLOPS	NA	2.5x	NA
A100 FP16 TC vs V100 FP16 TC	125 TFLOPS	312 TFLOPS	624 TFLOPS	2.5x	5x
A100 BF16 TC vs V100 FP16 TC	125 TFLOPS	312 TFLOPS	624 TFLOPS	2.5x	5x
A100 FP32 vs V100 FP32	15.7 TFLOPS	19.5 TFLOPS	NA	1.25x	NA
A100 TF32 TC vs V100 FP32	15.7 TFLOPS	156 TFLOPS	312 TFLOPS	10x	20x
A100 FP64 vs V100 FP64	7.8 TFLOPS	9.7 TFLOPS	NA	1.25x	NA
A100 FP64 TC vs V100 FP64	7.8 TFLOPS	19.5 TFLOPS	NA	2.5x	NA
A100 INT8 TC vs V100 INT8	62 TOPS	624 TOPS	1248 TOPS	10x	20x
A100 INT4 TC	NA	1248 TOPS	2496 TOPS	NA	NA
A100 Binary TC	NA	4992 TOPS	NA	NA	NA

Tensor core is a new type of processing core that performs a type of specialized matrix math, suitable for deep learning and certain types of HPC. Tensor cores perform a fused multiply add, where two 4 x 4 FP16 matrices are multiplied and then the result added to a 4 x 4 FP16 or FP32 matrix. The result is a 4 x 4 FP16 or FP32 matrix; NVIDIA refers to tensor cores as performing mixed precision math, because the inputted matrices are in half precision but the product can be in full precision.



Tensor Core MM, V100, A100 Performance

- ✓ A100 : 1 SM has 4 TC,
- ✓ Each TC does 256 FP16 FMA unit
- ✓ Imply each TC can do a 8x4x8 (multiplying 8×4 matrix with 4×8 matrix) mixed-precision matrix multiplication per clock – (256)
- ✓ So 1 SM can do 4 x 256 = 1024 FP16 FMA or 2048 FP16 operations per clock cycle
- ✓ 108 x 2048 x 1.4 = 310 TFLOPS
- ✓ Same for FP32 → FP64



NVIDIA A100 Tensor Core GPU with its 108 SMs includes a total of 432 Tensor Cores delivers up to 312 TFLOPS of dense mixed-precision FP16/FP32 performance.

That equates to 2.5x the mixed-precision Tensor Core performance of the entire Tesla V100 GPU, and 20x V100's standard FP32 (FMA operations running on traditional FP32 CUDA cores) throughput.

FMA = Fused Multiple-add, compute 2 floating point

- ✓ V100 : 1 SM has 8 TC,
- ✓ Each TC does 64 FP16
- ✓ Imply each TC can do a 4x4x4 matrix – 64 FMA
- ✓ So 1 SM can do 8 x 64 = 512 FP16 FMA or 1024 FP16 operations per clock cycle
- ✓ 80SM x 1024 FP16 / C x 1.53 GHz = 125 TFLOPS

LAB 2
Problem 4

Write a python code to compute C=AXB and plot a curve of the FLOPS against the matrix size N (take N=2000, 4000, 6000) using single precision when the computation is done on a CPU. DO the same on the GPU

```
import numpy as np
import time

A = np.random.rand(2000, 2000).astype('float32')
B = np.random.rand(2000, 2000).astype('float32')
%timeit np.dot(A,B)
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm2000 = timeend - timestt
gf2000 = 2*2*2*2/tmm2000
print('time = ',tmm2000)
print('GFLOPS = ', gf2000)

A = np.random.rand(4000, 4000).astype('float32')
B = np.random.rand(4000, 4000).astype('float32')
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm4000 = timeend - timestt
gf4000 = 2*4*4*4/tmm4000
print('time = ',tmm4000)
print('GFLOPS = ', gf4000)
```

```
A = np.random.rand(6000, 6000).astype('float32')
B = np.random.rand(6000, 6000).astype('float32')
timestt = time.time()
C=np.dot(A,B)
timeend = time.time()
tmm6000 = timeend - timestt
gf6000 = 2*6*6*6/tmm6000
print('time = ',tmm6000)
print('GFLOPS = ', gf6000)

import torch
A = torch.randn(6000, 6000).cuda()
B = torch.randn(6000, 6000).cuda()
timestt = time.time()
C=torch.matmul(A,B)
timeend = time.time()
tmm6000 = timeend - timestt
gf6000 = 2*6*6*6/tmm6000
print('time = ',tmm6000)
print('GFLOPS = ', gf6000)
```

```
1 loop, best of 5: 228 ms per loop
time =  0.2360849380493164
GFLOPS =  67.7722184744277
time =  1.8578176498413086
GFLOPS =  68.8980428251037
time =  6.155170440673828
GFLOPS =  70.18489644824643
```

```
1 loop, best of 5: 214 ms per loop
time =  0.22870469093322754
GFLOPS =  69.95921218192831
time =  1.8288803100585938
GFLOPS =  69.98817762759944
time =  6.098201036453247
GFLOPS =  70.84056386754575
```

```
time =  6.029452800750732
GFLOPS =  71.6482928510878
time =  0.06104087829589844
GFLOPS =  7077.224510202169
```

GPU Performance architectures

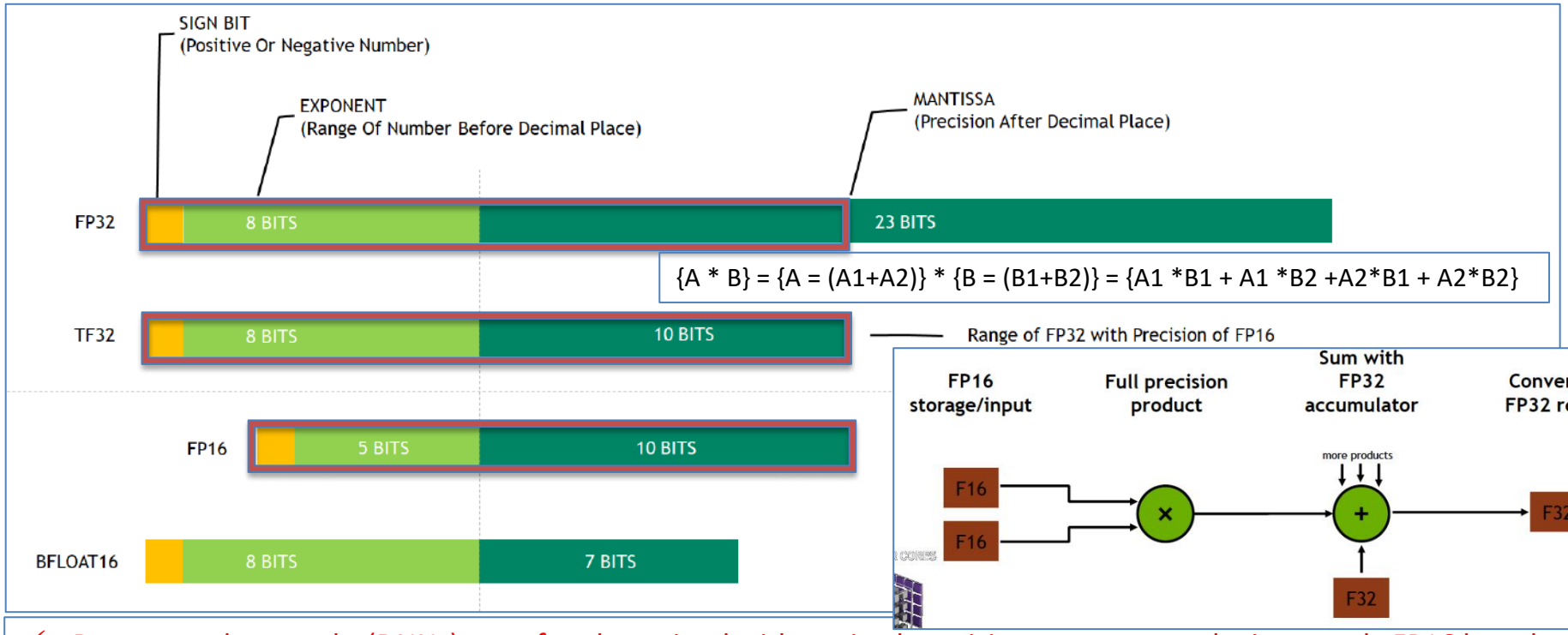
GPU Features	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100
GPU Codename	GP100	GV100	GA100
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
GPU Board Form Factor	SXM	SXM2	SXM4
SMs	56	80	108
TPCs	28	40	54
FP32 Cores / SM	64	64	64
FP32 Cores / GPU	3584	5120	6912
FP64 Cores / SM (excl. Tensor)	32	32	32
FP64 Cores / GPU (excl. Tensor)	1792	2560	3456
INT32 Cores / SM	NA	64	64
INT32 Cores / GPU	NA	5120	6912
Tensor Cores / SM	NA	8	4[2]
Tensor Cores / GPU	NA	640	432
GPU Boost Clock	1480 MHz	1530 MHz	1410 MHz
Peak FP16 Tensor TFLOPS with FP16 Accumulate[1]	NA	125	312/624[3]
Peak FP16 Tensor TFLOPS with FP32 Accumulate[1]	NA	125	312/624[3]
Peak BF16 Tensor TFLOPS with FP32 Accumulate[1]	NA	NA	312/624[3]
Peak TF32 Tensor TFLOPS[1]	NA	NA	156/312[3]
Peak FP64 Tensor TFLOPS[1]	NA	NA	19.5
Peak INT8 Tensor TOPS[1]	NA	NA	624/1248[3]
Peak INT4 Tensor TOPS[1]	NA	NA	1248/2496[3]
Peak FP16 TFLOPS[1] (non-Tensor)	21.2	31.4	78
Peak BF16 TFLOPS[1] (non-Tensor)	NA	NA	39
Peak FP32 TFLOPS[1] (non-Tensor)	10.6	15.7	19.5
Peak FP64 TFLOPS[1] (non-Tensor)	5.3	7.8	9.7
Peak INT32 TOPS[1,4]	NA	15.7	19.5
Texture Units	224	320	432
Memory Interface	4096-bit HBM2	4096-bit HBM2	5120-bit HBM2
Memory Size	16 GB	32 GB / 16 GB	40 GB
Memory Data Rate	703 MHz DDR	877.5 MHz DDR	1215 MHz DDR

Peak FP64[1]	9.7 TFLOPS
Peak FP64 Tensor Core[1]	19.5 TFLOPS
Peak FP32[1]	19.5 TFLOPS
Peak FP16[1]	78 TFLOPS
Peak BF16[1]	39 TFLOPS
Peak TF32 Tensor Core[1]	156 TFLOPS | 312 TFLOPS[2]
Peak FP16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak BF16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak INT8 Tensor Core[1]	624 TOPS | 1,248 TOPS[2]
Peak INT4 Tensor Core[1]	1,248 TOPS | 2,496 TOPS[2]

Memory Bandwidth	720 GB/sec	900 GB/sec	1555 GB/sec
L2 Cache Size	4096 KB	6144 KB	40960 KB
Shared Memory Size / SM	64 KB	Configurable up to 96 KB	Configurable up to 164 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	14336 KB	20480 KB	27648 KB
TDP	300 Watts	300 Watts	400 Watts
Transistors	15.3 billion	21.1 billion	54.2 billion
GPU Die Size	610 mm²	815 mm²	826 mm2
TSMC Manufacturing Process	16 nm FinFET+	12 nm FFN	7 nm N7

1. Peak rates are based on GPU Boost Clock.
2. Four Tensor Cores in an A100 SM have 2x the raw FMA computational power of eight Tensor Cores in a GV100 SM.
3. Effective TOPS / TFLOPS using the new Sparsity Feature
4. TOPS = IMAD-based integer math

Numerical precisions supported by NVIDIA A100 for Deep Learning



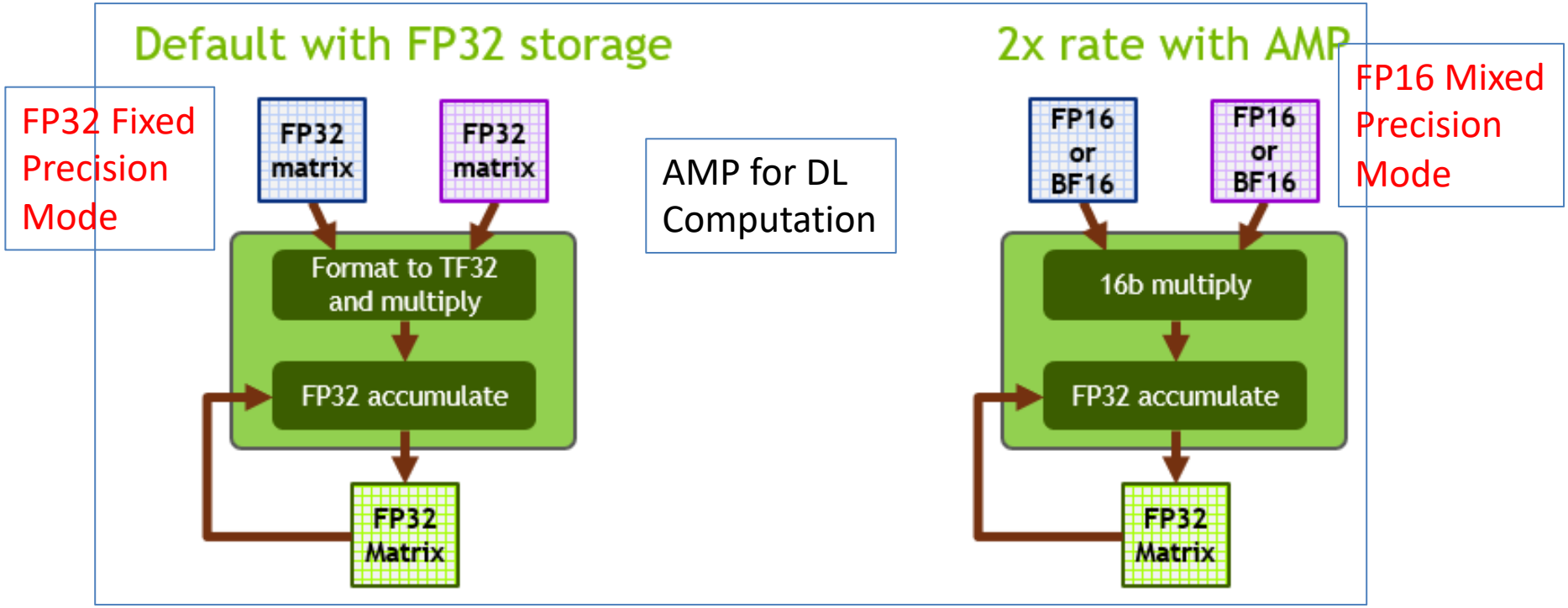
- ✓ Deep neural networks (DNNs) can often be trained with a mixed precision strategy, employing mostly FP16 but also FP32 precision when necessary. This strategy results in a significant reduction in computation, memory, and memory bandwidth requirements while most often converging to the similar final accuracy.
- ✓ NVIDIA Tensor Cores are specialized arithmetic units on NVIDIA Volta and newer generation GPUs. They can carry out a complete matrix multiplication and accumulation operation (MMA) in a single clock cycle. On Volta and Turing, the inputs are two matrices of size 4×4 in FP16 format, while the accumulator is in FP32.
- ✓ The third-generation Tensor Cores on Ampere support a novel math mode: TF32 is a hybrid format defined to handle the work of FP32 with greater efficiency. Specifically, TF32 uses the same 10-bit mantissa as FP16 to ensure accuracy while sporting the same range as FP32, thanks to using an 8-bit exponent.
- ✓ A wider representable range matching FP32 eliminates the need of a loss-scaling operation when using TF32, thus simplifying the mixed precision training workflow.

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7959640/pdf/peerj-cs-07-330.pdf

https://developer.nvidia.com/blog/accelerating-tensorflow-on-a100-gpus/

Software : Automatic Mixed Precision (AMP) : A100 : 2 Modes

- ✓ 1st mode (FP32 Single Precision): On Ampere Tensor Cores, TF32 is the default math mode for Deep Learning workloads, as opposed to FP32 on Volta/Turing GPUs. Internally, when operating in TF32 mode, Ampere Tensor Cores accept two FP32 matrices as inputs but internally carry out matrix multiplication in TF32 format. The result is accumulated in an FP32 matrix.
- ✓ 2nd Mode (Mixed Precision) : When operating in FP16/BF16 mode, Ampere Tensor Cores accept FP16/BF16 matrices instead, and accumulate in an FP32 matrix. FP16/BF16 mode on Ampere provides 2x throughput compared to TF32.



- ✓ TF32 is designed to bring the processing power of NVIDIA Tensor Cores technologies to all DL workloads without any required code changes.
- ✓ Neural network models that can be trained successfully to convergence with AMP on FP16/BF16 can also be trained to convergence with TF32.

https://developer.nvidia.com/blog/accelerating-tensorflow-on-a100-gpus/

TF32 Single Precision - Tensor Core mode

TF32 is a Tensor Core mode, not a type

- ✓ Only convolutions and matrix multiplies convert inputs to TF32, all other operations remain completely FP32
- ✓ All storage in memory remains FP32
- ✓ Consequently, it's only exposed as a Tensor Core operation mode, contrast with fp16/bfloat16 types that provide: storage, various math operators, etc

Operation:

- ✓ Read FP32 inputs from memory, round inputs to TF32 prior to Tensor Core operation
- ✓ Multiply inputs without loss of precision, accumulate products in FP32, write FP32 product to memory

SAMPLING OF NETWORKS

Classification Tasks

Architecture	Network	Top-1 Accuracy	
		FP32	TF32
ResNet	RN18	70.43	70.58
	RN32	74.03	74.08
	RN50	76.78	76.73
	RN101	77.57	77.57
ResNext	RNX50	77.51	77.62
	RNX101	79.10	79.30
WideResNet	WRN50	77.99	78.11
	WRN101	78.61	78.62
DenseNet	DN121	75.57	75.57
	DN169	76.75	76.69
VGG	V11-BN	68.47	68.44
	V16-BN	71.54	71.51
	V19-BN	72.54	72.68
	V19	71.75	71.60
GoogleNet	InceptionV3	77.20	77.34
	Xception	79.09	79.31
Dilated RN	DRN A 50	78.24	78.16
ShuffleNet	V2-X1	68.62	68.87
	V2-X2	73.02	72.88
MNASNet	V1.0	71.62	71.49
SqueezeNet	V1_1	60.90	60.85
MobileNet	MN-V2	71.64	71.76
Stacked UNet	SUN64	69.53	69.62
EfficientNet	B0	76.79	76.72

Dataset is ISLVRC 2012

Detection & Segmentation Tasks

Architecture	Network	Metric	Model Accuracy	
			FP32	TF32
Faster RCNN	RN50 FPN 1X	mAP	37.81	37.95
	RN101 FPN 3X	mAP	40.04	40.19
	RN50 FPN 3X	mAP	42.05	42.14
Mask RCNN	TorchVision	mAP	37.89	37.89
		mIOU	34.65	34.69
	RN50 FPN 1X	mAP	38.45	38.63
		mIOU	35.16	35.25
	RN50 FPN 3X	mAP	41.04	40.93
		mIOU	37.15	37.23
	RN101 FPN 3X	mAP	42.99	43.08
		mIOU	38.72	38.73
Retina Net	RN50 FPN 1X	mAP	36.46	36.49
	RN50 FPN 3X	mAP	38.04	38.19
	RN101 FPN 3X	mAP	39.75	39.82
RPN	RN50 FPN 1X	mAP	58.02	58.11
Single-Shot Detector (SSD)	RN18	mAP	19.13	19.18
	RN50	mAP	24.91	24.85

Dataset is MS COCO 2017

Language Tasks

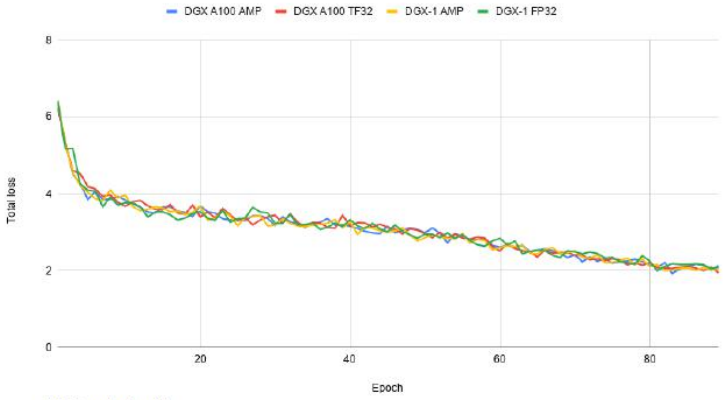
Architecture	Network	Dataset	Metric	Model Accuracy	
				FP32	TF32
Transformer	Vaswani Base	WMT	BLEU	27.18	27.10
	Vaswani Large	WMT	BLEU	28.63	28.62
	Levenshtein	WMT	Loss	6.16	6.16
Convolutional	Light Conv Base	WMT	BLEU	28.55	28.74
	Light Conv Large	WMT	BLEU	30.10	30.20
	Dynamic Conv Base	WMT	BLEU	28.34	28.42
	Dynamic Conv Large	WMT	BLEU	30.10	30.31
	FairSeq Conv	WMT	BLEU	24.83	24.86
Recurrent	GNMT	WMT	BLEU	24.53	24.80
Convolutional	Fairseq Dauphin	WikiText	PPL	35.89	35.80
Transformer	XL Standard	WikiText	PPL	22.89	22.80
BERT	Base Pre-train	Wikipedia	LM Loss	1.34	1.34
	Base Downstream	SQUAD v1	F1	87.95	87.66
		SQUAD v2	F1	76.68	75.67

No hyperparameter changes
Differences in accuracy are within typical bounds of run-to-run variation (different random seeds, etc.)

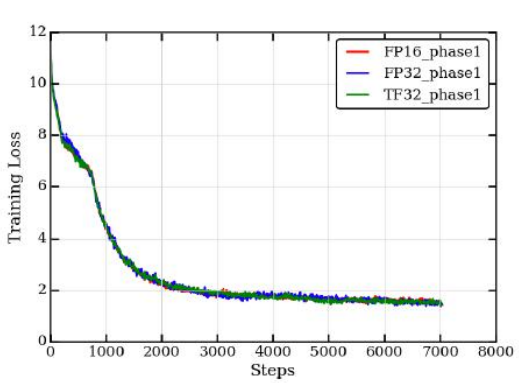
13 NVIDIA

Training Neural Networks with Tensor Cores - Dusan Stosic, NVIDIA
https://www.youtube.com/watch?v=jF4-_ZK_tyc

DL Performance of TF32 Single Precision mode



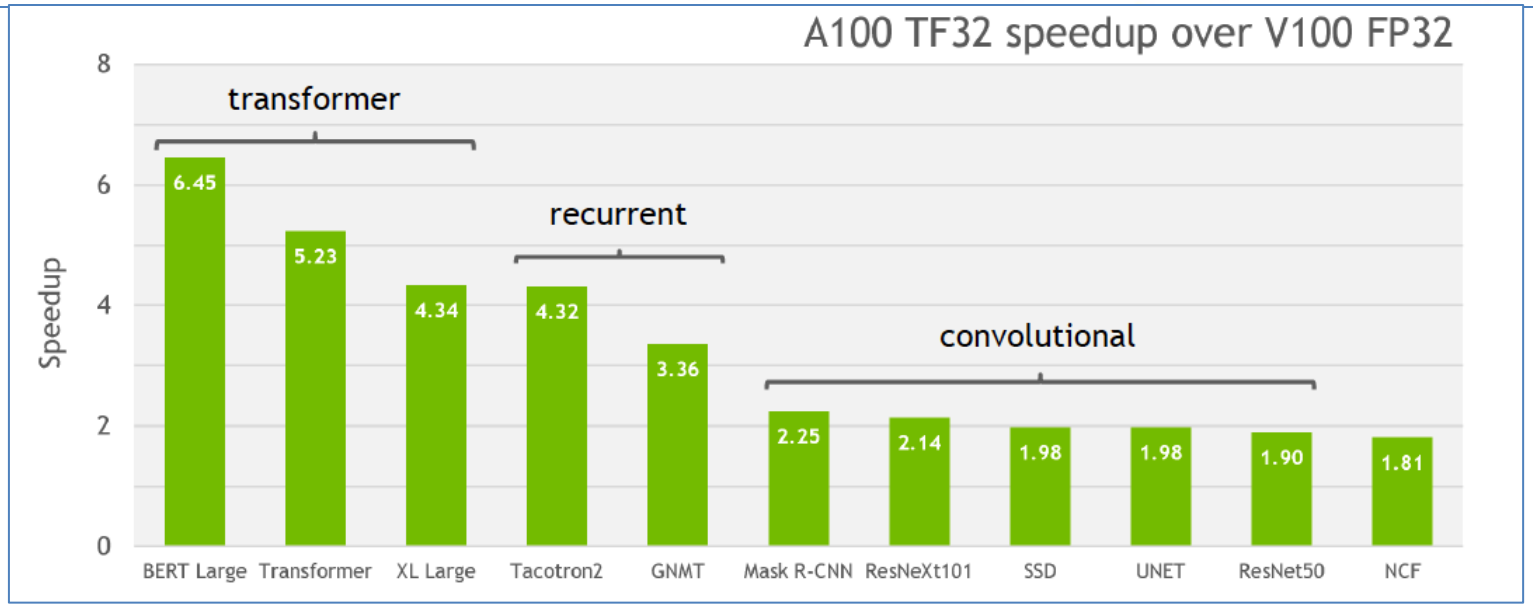
Model can be found at:
https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Classification/ConvNets/resnet50v1.5
Data collected on NVIDIA A100
Results can be reproduced with Tensorflow 1.15 in NGC container tensorflow:20.06-tf1-py3



Model can be found at:
https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/BERT
Data collected on NVIDIA A100
Performance can be reproduced with PyTorch 1.6 in NGC pytorch:20.06-py3 container

Results are easily reproducible using NGC containers and Deep Learning Examples

17 NVIDIA



Training Neural Networks with Tensor Cores - Dusan Stosic, NVIDIA

https://www.youtube.com/watch?v=jF4-_ZK_tyc

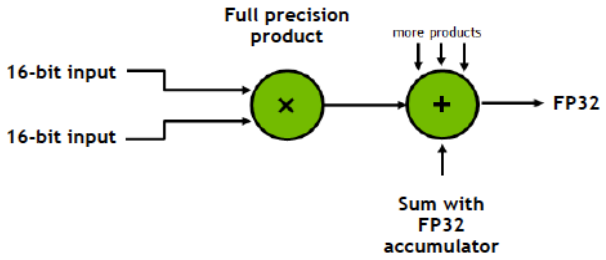
TENSOR CORES FOR 16-BIT FORMATS

Fastest way to train networks

Operation:

$\{A * B\} = \{A = (A1+A2)\} * \{B = (B1+B2)\} = \{A1 * B1 + A1 * B2 + A2 * B1 + A2 * B2\}$

- Multiply and add FP16 or BF16 tensors
- Products are computed without loss of precision, accumulated in FP32
- Final FP32 output is rounded to FP16/BF16 before writing to memory



NVIDIA Ampere Architecture enhancements:

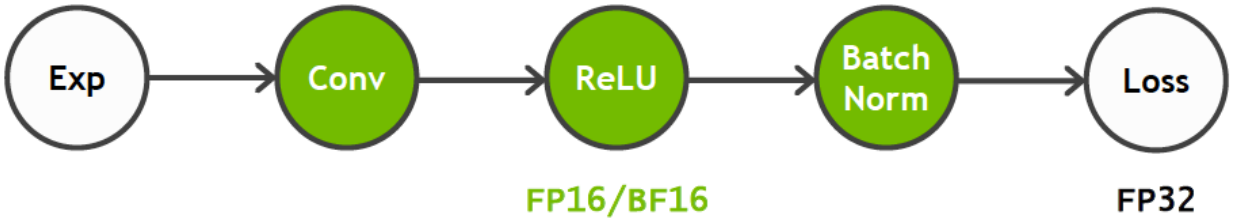
- New tensor core design: 2.5x throughput for dense operations (A100 vs V100)
- Sparsity support: additional 2x throughput for sparse operations
- BFloat16 (BF16): Same rate as FP16

Combines single-precision (FP32) with lower precision (e.g. FP16) when training a network

✓ Use lower precision where applicable (e.g. convolutions, matrix multiplies)

✓ Keep certain operations in FP32

Achieves the same accuracy as FP32 training using all the same hyper-parameters



Training Neural Networks with Tensor Cores - Dusan Stosic, NVIDIA

https://www.youtube.com/watch?v=jF4-_ZK_tyc

Mixed Precision - Tensor Core mode

LAYER SELECTION

Decide which operations to compute in FP32/16-bits

WEIGHT STORAGE

Keep model weights and updates in FP32

LOSS SCALING

Retain small gradient magnitudes for FP16

8-16x acceleration from FP16/BF16 Tensor Cores

2x acceleration with 16-bit formats (but should not sacrifice accuracy)

Matrix Multiplications	**Reductions**	**Loss Functions**	**Pointwise**
linear, matmul, bmm, conv	batch norm, layer norm, sum, softmax	cross entropy, l2 loss, weight decay	relu, sigmoid, tanh, exp, log

Operations that can use 16-bit storage (FP16/BF16)

✓ Matrix multiplications. Most pointwise operations (e.g. relu, tanh, add, sub, mul)

Operations that need more precision (FP32/FP16)

✓ Adding small values to large sums can lead to rounding errors, Reduction operations (e.g. sum, softmax, normalization)

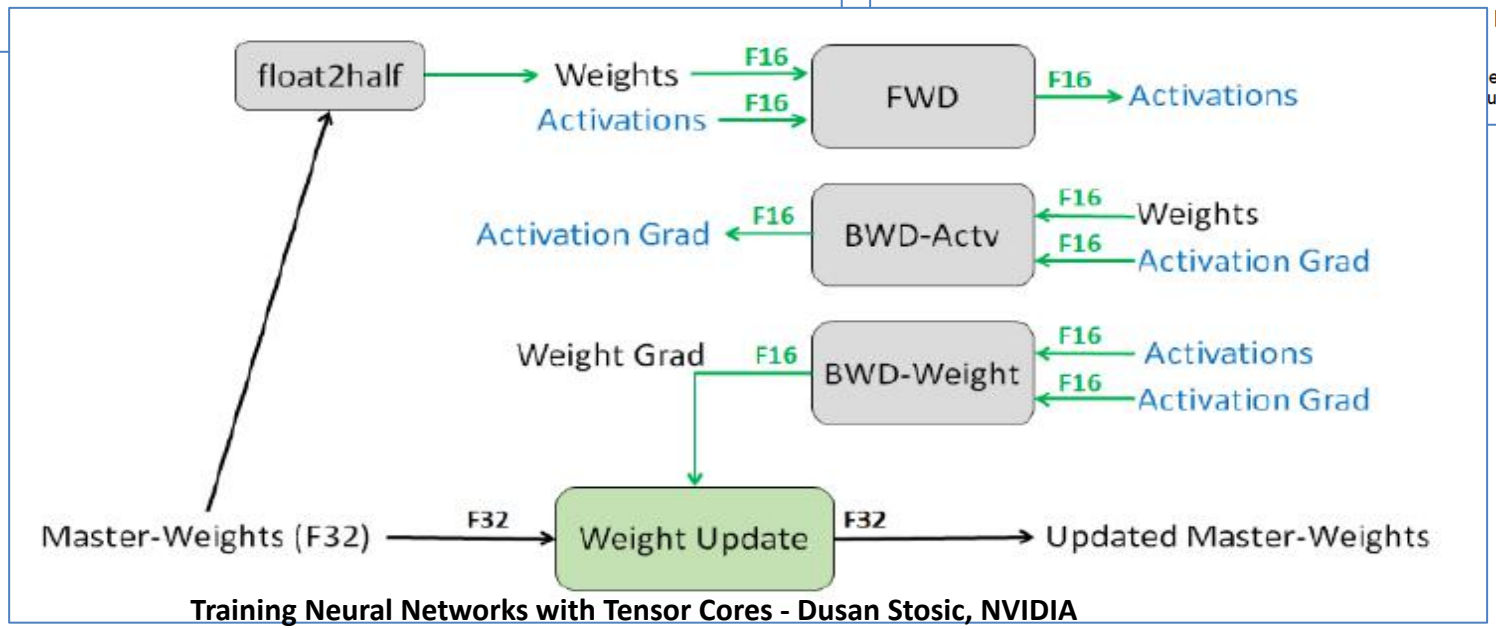
Operations that need more range (FP32/BF16)

✓ Pointwise operations where $f(x) \gg |x|$ (e.g. exp, log, pow), Loss functions

cuDNN >= 8.0	cuBLAS >= 11.0
Convolutions	Linear algebra operations
TF32 is the default math	Default math mode is FP32 because of HPC
TF32 kernels selected when operating on 32-bit data	TF32 enabled when math mode set to CUBLAS_TF32_TENSOR_OP_MATH *

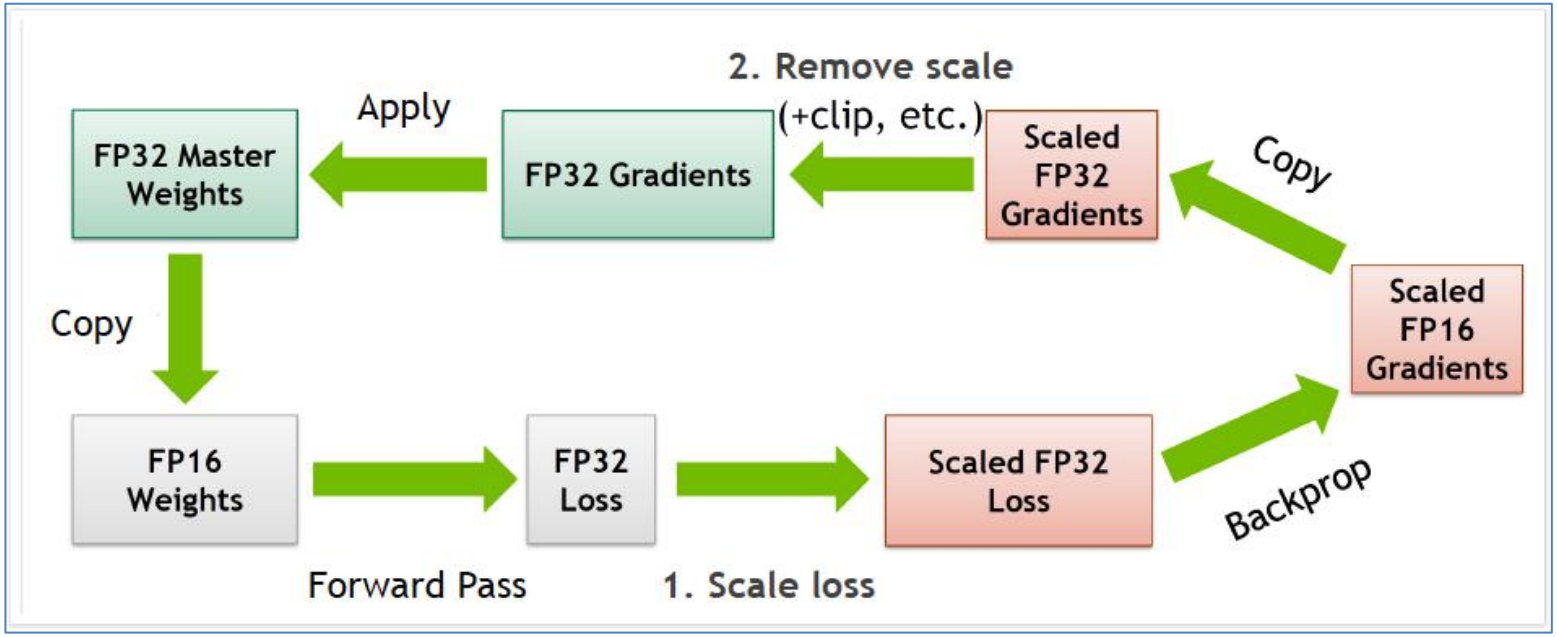
keep math in FP32

eration
uBLAS state



Training Neural Networks with Tensor Cores - Dusan Stosic, NVIDIA

Tensor Core Deep Learning Operations



Conservative default : keep weights in FP32 so that small updates accumulate across iterations

A100 introduces the next generation of Tensor Cores for DL acceleration

- ✓ TF32 is the default math mode on A100
- ✓ Accelerates single-precision training, 10x more math throughput that Volta single-precision

FP16 and BF16 formats for maximum speed

- ✓ FP16 and BF16 Tensor Cores provide 16x more math throughput than FP32 (2x faster than TF32)
- ✓ AMP makes FP16 training easy in all major frameworks
- ✓ Training results match those of single-precision, require no changes to hyper-parameters
- ✓ Also reduce memory consumption, enabling larger batches, larger models, etc

Sparsity support for a further 2x math throughput

- ✓ Accelerates DL inference

NVIDIA cuDNN : Developer Guide I NVIDIA Docs

NVIDIA CUDA Deep Neural Network LIbrary (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines arising frequently in DNN applications:

- Convolution forward and backward, including cross-correlation
- Matrix multiplication
- Pooling forward and backward
- Softmax forward and backward
- Neuron activations forward and backward: relu, tanh, sigmoid, elu, gelu, softplus, swish
- Arithmetic, mathematical, relational and logical pointwise operations
- Tensor transformation functions
- LRN, LCN and batch normalization forward and backward

cuDNN convolution routines aim for a performance that is competitive with the fastest GEMM (matrix multiply)-based implementations of such routines while using significantly less memory.

cuDNN features include customizable data layouts, supporting flexible dimension ordering, striding, and subregions for the 4D tensors used as inputs and outputs to all of its routines. This flexibility allows easy integration into any neural network implementation and avoids the input/output transposition steps sometimes necessary with GEMM-based convolutions. cuDNN offers a context-based API that allows for easy multithreading and (optional) interoperability with NVIDIA CUDA streams.

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

Chapter 2. Tensor Descriptor

The cuDNN library describes data holding images, videos and any other data with contents with a generic n-D tensor defined with the following parameters:

- a dimension nbDims from 3 to 8
- a data type (32-bit floating-point, 64 bit-floating point, 16-bit floating-point...)
- dimA integer array defining the size of each dimension
- strideA integer array defining the stride of each dimension (for example, the number of elements to add to reach the next element from the same dimension)

The first dimension of the tensor defines the batch size n, and the second dimension defines the number of features maps c. This tensor definition allows, for example, to have some dimensions overlapping each other within the same tensor by having the stride of one dimension smaller than the product of the dimension and the stride of the next dimension. In cuDNN, unless specified otherwise, all routines will support tensors with overlapping dimensions for forward-pass input tensors, however, dimensions of the output tensors cannot overlap. Even though this tensor format supports negative strides (which can be useful for data mirroring), cuDNN routines do not support tensors with negative strides unless specified otherwise.

A 4-D tensor descriptor is used to define the format for batches of 2D images with 4 letters: N,C,H,W for respectively the batch size, the number of feature maps, the height and the width. The letters are sorted in decreasing order of the strides. The commonly used 4-D tensor formats are: Tensor Descriptor, NVIDIA cuDNN PG-06702-001_v8.3.3 | 6

- 4-D - NCHW, NHWC, CHWN
- 5-D - NCDHW, NDHWC, CDHWN

Figure 1. Example with N=1, C=64, H=5, W=4.

EXAMPLE

N = 1

C = 64

H = 5

W = 4

c = 0

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

c = 1

20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39

c = 2

40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59

. . .

c = 30

600	601	602	603
604	605	606	607
608	609	610	611
612	613	614	615
616	617	618	619

c = 31

620	621	622	623
624	625	626	627
628	629	630	631
632	633	634	635
636	637	638	639

c = 32

640	641	642	643
644	645	646	647
648	649	650	651
652	653	654	655
656	657	658	659

. . .

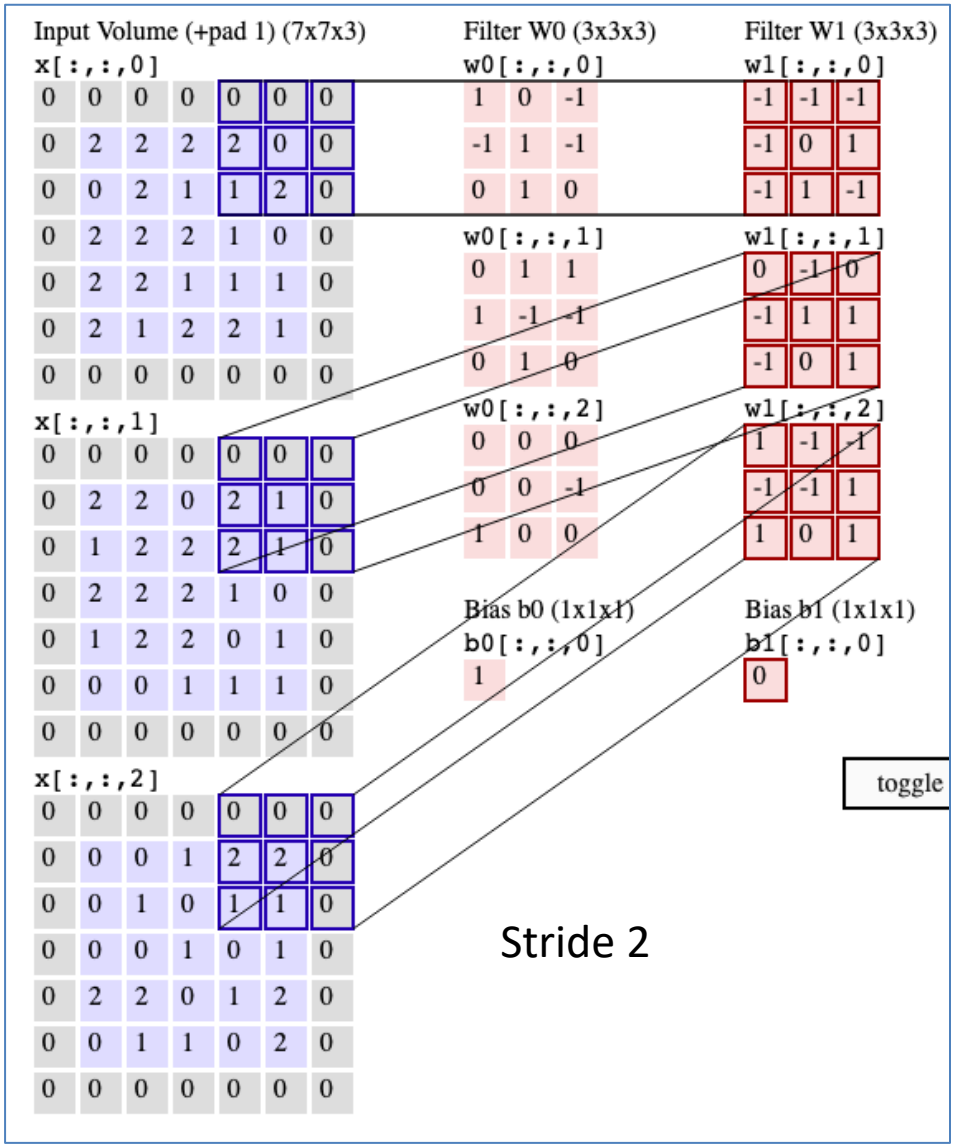
c = 62

1240	1241	1242	1243
1244	1245	1246	1247
1248	1249	1250	1251
1252	1253	1254	1255
1256	1257	1258	1259

c = 63

1260	1261	1262	1263
1264	1265	1266	1267
1268	1269	1270	1271
1272	1273	1274	1275
1276	1277	1278	1279

- **N** is the batch size; 1.
- **C** is the number of feature maps (i.e., number of channels); 64.
- **H** is the image height; 5.
- **W** is the image width; 4.



Input Tensor
= N xH x W x C
= 1 x 7 x 7 x 3

Input Tensor
H(i), W(j), C(k)
For N=1,1
for i =1, 7
for j = 1, 7
for k=1, 3
imat(n,i,j,k)

Filter Tensor
= H x W x C x M
= 3 x 3 x 3 x 2

Filter Tensor
H(i), W(j), C(k), M(m)
for H =1, 3
for j = 1, 3
for k=1, 3
for m =1,2
fmat(i,j,k,m)

Overlapping max pooling of 2x2, stride 1

Flatten Fully Connected layer of neurons

Fully Connected layer of 2 softmax neurons

Dimension of Input = (1, 7, 7, 3)
Dimension of Filter = (3, 3, 3, 2)

The output values after the convolution step are:
Dimension of qa = (1, 3, 3, 2)
[[[[-2. 7.] [1. 1.] [4. -7.]]
[[1. 6.] [9. -8.] [6. -6.]]
[[2. -7.] [1. -4.] [2. -8.]].]]

The output values after the ReLU operation are
Dimension of qb = (1, 3, 3, 2)
[[[[0. 7.] [1. 1.] [4. 0.]]
[[1. 6.] [9. 0.] [6. 0.]]
[[2. 0.] [1. 0.] [2. 0.]]]]

The output values after the max pooling operation are
Dimension of qc = (1, 2, 2, 2)
[[[[9. 7.] [9. 1.]]
[[9. 6.] [9. 0.]].]]

The values of the vector after it is flattened based on the row major counting system are
Dimension of qd = (1, 8)
[[9 9 9 9 7 1 6 0]]

The total number of the trainable parameters is 146

The outputs after the softmax function are
Dimension of output = (8,)
[2.3891544e-01 2.3891544e-01 2.3891544e-01 2.3891544e-01
3.2333687e-02 8.0147205e-05 1.1894900e-02 2.9484507e-05]

Input Tensor
= N xH x W x C
= 1 x 7 x 7 x 3

Filter Tensor
= H x W x C x M
= 3 x 3 x 3 x 2

qa, qb tensor
= N x H x W x M
= 1 x 3 x 3 x 2

qc tensor
= N x H x W x M
= 1 x 2 x 2 x 2

qd matrix
= 1 x (2 x 2 x 2) = 1 x 8

softmax vector
= (2 x 2 x 2) = 8

Chapter 6. Tensor Core Operations

Tensor Core operations accelerate matrix math operations; cuDNN uses Tensor Core operations that accumulate into FP16, FP32, and INT32 values. Setting the math mode to CUDNN_TENSOR_OP_MATH via the cudnnMathType_t enumerator indicates that the library will use Tensor Core operations. This enumerator specifies the available options to enable the Tensor Core and should be applied on a per-routine basis.

The default math mode is CUDNN_DEFAULT_MATH, which indicates that the Tensor Core operations will be avoided by the library. Because the CUDNN_TENSOR_OP_MATH mode uses the Tensor Cores, it is possible that these two modes generate slightly different numerical results due to different sequencing of the floating-point operations.

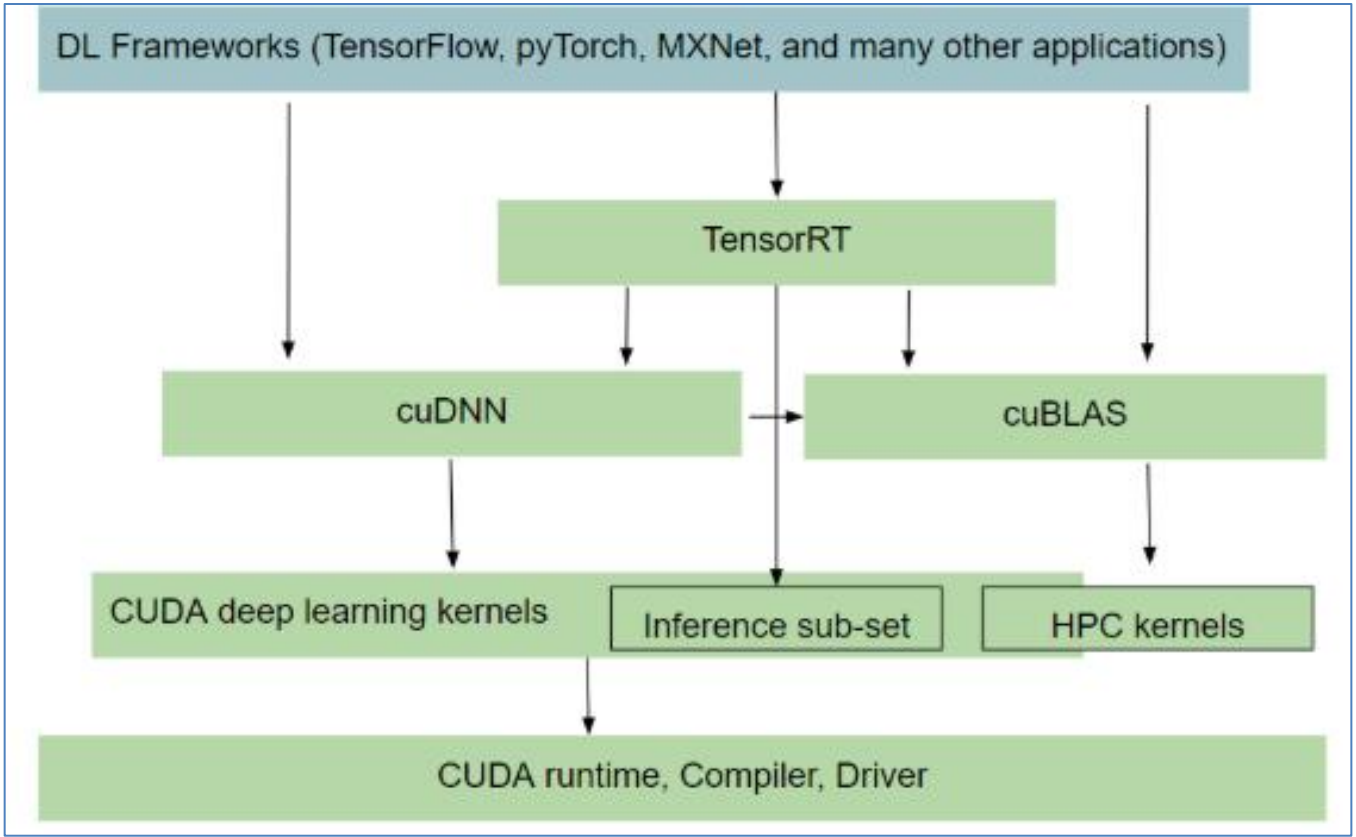
For example, the result of multiplying two matrices using Tensor Core operations is very close, but not always identical, to the result achieved using a sequence of scalar floating-point operations. For this reason, the cuDNN library requires an explicit user opt-in before enabling the use of Tensor Core operations.

However, experiments with training common deep learning models show negligible differences between using Tensor Core operations and scalar floating point paths, as measured by both the final network accuracy and the iteration count to convergence. Consequently, the cuDNN library treats both modes of operation as functionally indistinguishable and allows for the scalar paths to serve as legitimate fallbacks for cases in which the use of Tensor Core operations is unsuitable. Kernels using Tensor Core operations are available for:

- Convolutions
- RNNs
- Multi-Head Attention

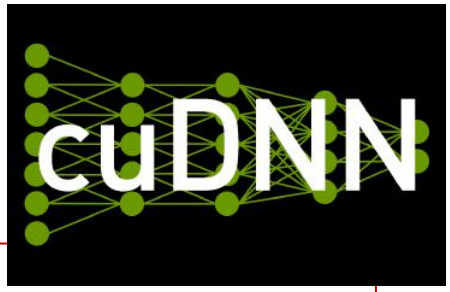
Convolution Functions

Supported Convolution Function	Supported Algos
cudnnConvolutionForward	CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED
cudnnConvolutionBackwardData	CUDNN_CONVOLUTION_BWD_DATA_ALGO_1 CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED
cudnnConvolutionBackwardFilter	CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1 CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAD_NONFUSED



Introduction to CuDNN

- CuDNN is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.



- Basic programming model consists of:
  - Defining data in terms of tensors, including their shape and memory layout; describing the algorithms to perform; allocating memory for cuDNN to operate on (a workspace) and finally executing the actual operations.
  - Using those tensors as arguments to cuDNN's built-in functions for both the forwards and backwards passes through a neural network

- More details about everything we discussed in NVIDIA's official cuDNN developer guide

Key Features of CuDNN

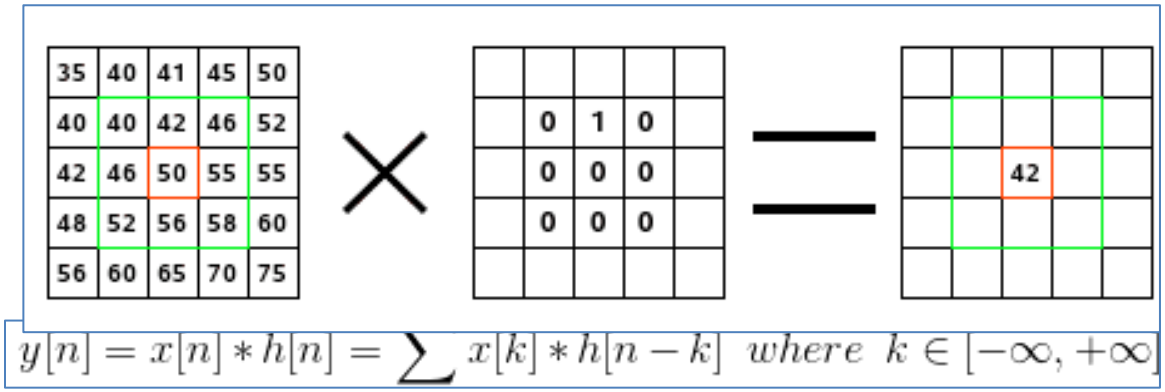
- Grouped convolutions support NHWC inputs/outputs and FP16/FP32 compute for models such as ResNet and Xception
- Dilated convolutions using mixed precision Tensor Core operations for applications such as semantic segmentation, image super-resolution, denoising, etc
- TensorCore acceleration with FP32 inputs and outputs (previously restricted to FP16 input)
- RNN cells support multiple use cases with options for cell clipping and padding masks
- Automatically select the best RNN implementation with RNN search API
- Arbitrary dimension ordering, striding, and sub-regions for 4d tensors means easy integration into any neural net implementation

http://www.goldsborough.me/cuda/ml/cudnn/c++/2017/10/01/14-37-23-convolutions_with_cudnn/
https://devblogs.nvidia.com/tag/deep-neural-networks/

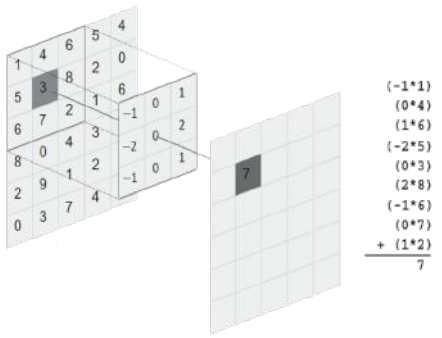
CuDNN examples: Convolutions with cuDNN

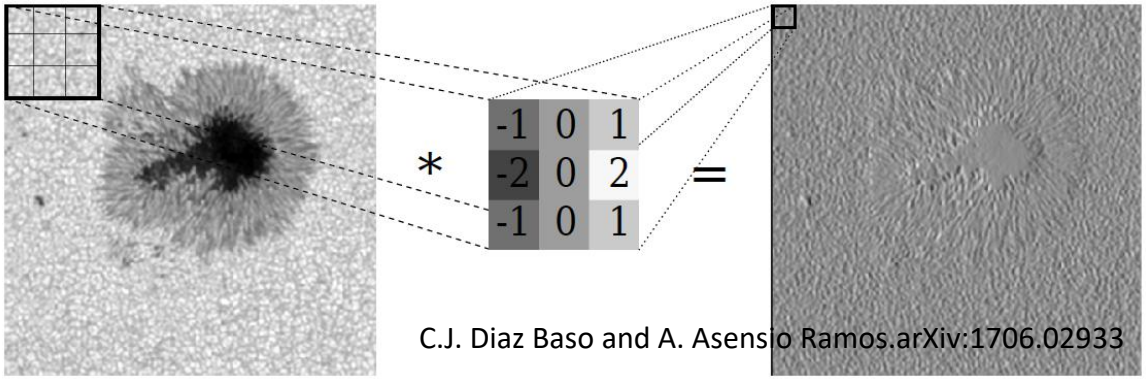
- Convolution is the treatment of a matrix by another one which is called “kernel”.

multiplies together two arrays of numbers to produce a third array of numbers of the same dimensionality



- One of the fundamental building blocks in image processing





An example of a convolution with a filter. A vertical border-locating kernel is convolved with the input image of the Sun. A resulting feature map of size (N-2)X(N- 2) is generated from the convolution

https://docs.gimp.org/2.8/en/plug-in-convmatrix.html, Heterogeneous Computing with OpenCL 2.0

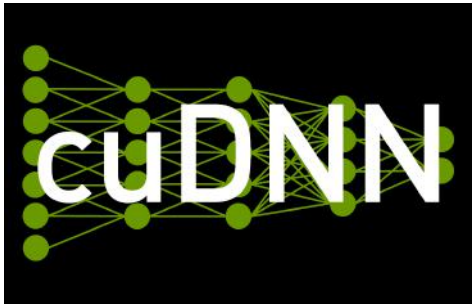
CuDNN examples: Convolutions with cuDNN

Implementation

- begin by including the necessary header and creating an object of cudnnHandle_t type, which will serve as a sort of *context* object, connecting the various operations we need to piece together for our convolution kernel:

```
#include <cudnn.h>

int main(int argc, char const *argv[]) {
  cudnnHandle_t cudnn;
  cudnnCreate(&cudnn);
}
```

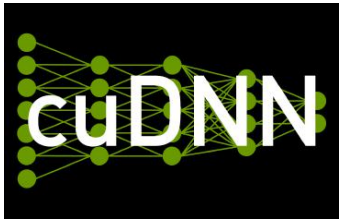
cudnnCreate, like all other cuDNN routines, returns an error code of cudnnStatus_t type.




```
cudnnFilterDescriptor_t kernel_descriptor;
checkCUDNN(cudnnCreateFilterDescriptor(&kernel_descriptor));
checkCUDNN(cudnnSetFilter4dDescriptor(kernel_descriptor,
                                      /*dataType=*/CUDNN_DATA_FLOAT,
                                      /*format=*/CUDNN_TENSOR_NCHW,
                                      /*out_channels=*/3,
                                      /*in_channels=*/3,
                                      /*kernel_height=*/3,
                                      /*kernel_width=*/3));
```

http://www.goldsborough.me/cuda/ml/cudnn/c++/2017/10/01/14-37-23-convolutions_with_cudnn/

CuDNN examples: Convolutions with cuDNN



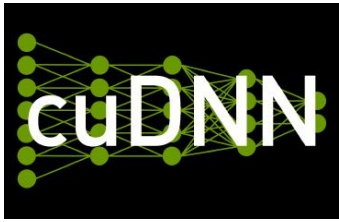
Implementation

- **# Describing the Convolution Kernel**: declare and configure a *descriptor*, which, you may notice, is an overarching pattern in cuDNN code

```
cudnnConvolutionDescriptor_t convolution_descriptor;
checkCUDNN(cudnnCreateConvolutionDescriptor(&convolution_descriptor));
checkCUDNN(cudnnSetConvolution2dDescriptor(convolution_descriptor,
                                           /*pad_height=*/1,
                                           /*pad_width=*/1,
                                           /*vertical_stride=*/1,
                                           /*horizontal_stride=*/1,
                                           /*dilation_height=*/1,
                                           /*dilation_width=*/1,
                                           /*mode=*/CUDNN_CROSS_CORRELATION,
                                           /*computeType=*/CUDNN_DATA_FLOAT));
```

http://www.goldsborough.me/cuda/ml/cudnn/c++/2017/10/01/14-37-23-convolutions_with_cudnn/

CuDNN examples: Convolutions with cuDNN



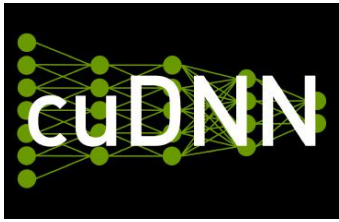
Implementation

- **# Describing the Convolution Kernel**: description of the convolution algorithm we want to use

```
cudnnConvolutionFwdAlgo_t convolution_algorithm;
checkCUDNN(
   cudnnGetConvolutionForwardAlgorithm(cudnn,
                                       input_descriptor,
                                       kernel_descriptor,
                                       convolution_descriptor,
                                       output_descriptor,

CUDNN_CONVOLUTION_FWD_PREFER_FASTEST,
                                       /*memoryLimitInBytes=*/0,
                                       &convolution_algorithm));
```

http://www.goldsborough.me/cuda/ml/cudnn/c++/2017/10/01/14-37-23-convolutions_with_cudnn/

CuDNN examples: Convolutions with cuDNN



Implementation

- **# Describing the Convolution Kernel**: declare the physical memory to operate on

```
size_t workspace_bytes = 0;
checkCUDNN(cudnnGetConvolutionForwardWorkspaceSize(cudnn,
                                                   input_descriptor,
                                                                kernel_descriptor,
                                                       convolution_descriptor,
                                                                output_descriptor,
                                                                convolution_algorithm,
                                                           &workspace_bytes));
std::cerr << "Workspace size: " << (workspace_bytes / 1048576.0) << "MB" << std::endl;
```

http://www.goldsborough.me/cuda/ml/cudnn/c++/2017/10/01/14-37-23-convolutions_with_cudnn/

CuDNN examples: Convolutions with cuDNN

Implementation
- **# Allocating Memory1**:

```
void* d_workspace{nullptr};
cudaMalloc(&d_workspace, workspace_bytes);

int image_bytes = batch_size * channels *
height * width * sizeof(float);

float* d_input{nullptr};
cudaMalloc(&d_input, image_bytes);
cudaMemcpy(d_input, image.ptr<float>(0),
image_bytes, cudaMemcpyHostToDevice);

float* d_output{nullptr};
cudaMalloc(&d_output, image_bytes);
cudaMemset(d_output, 0, image_bytes);
```

Implementation
- **# Allocating Memory2**:

```
// Mystery kernel
const float kernel_template[3][3] = {
 {1,  1, 1},
 {1, -8, 1},
 {1,  1, 1}
};

float h_kernel[3][3][3][3];
for (int kernel = 0; kernel < 3; ++kernel) {
 for (int channel = 0; channel < 3; ++channel) {
  for (int row = 0; row < 3; ++row) {
   for (int column = 0; column < 3; ++column) {
    h_kernel[kernel][channel][row][column] =
kernel_template[row][column];
   }
  }
 }
}

float* d_kernel{nullptr};
cudaMalloc(&d_kernel, sizeof(h_kernel));
cudaMemcpy(d_kernel, h_kernel, sizeof(h_kernel),
cudaMemcpyHostToDevice);
```

http://www.goldsborough.me/cuda/ml/cudnn/c++/2017/10/01/14-37-23-convolutions_with_cudnn/

CuDNN examples: Convolutions with cuDNN

Implementation

- # perform the actual convolution operation1:

```
const float alpha = 1, beta = 0;
checkCUDNN(cudnnConvolutionForward(cudnn,
                                   &alpha,
                                   input_descriptor,
                                   d_input,
                                   kernel_descriptor,
                                   d_kernel,
                                   convolution_descriptor,
                                   convolution_algorithm,
                                   d_workspace,
                                   workspace_bytes,
                                   &beta,
                                   output_descriptor,
                                   d_output));
```

http://www.goldsborough.me/cuda/ml/cudnn/c++/2017/10/01/14-37-23-convolutions_with_cudnn/

Proven to match FP32 results across a wide range of **tasks**, **problem domains**, **deep neural network architectures**

Image Classification
AlexNet
DenseNet
Inception
MobileNet
EfficientNet
ResNet
ResNeXt
ShuffleNet
SqueezeNet
VGG
Xception
Dilated ResNet
Stacked U-Net

Detection / Segmentation
DeepLab
Faster R-CNN
Mask R-CNN
SSD
NVIDIA Automotive
RetinaNet
UNET
DETR

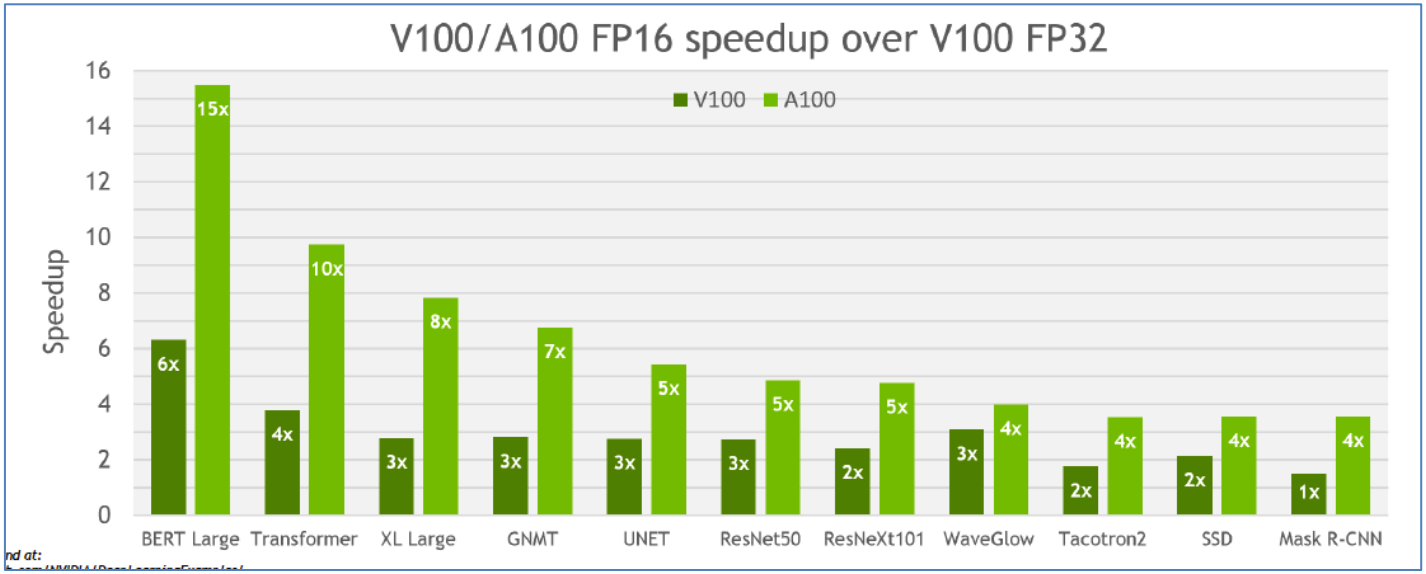
Recommendation
DeepRecommender
DLRM
NCF

Generative Models (Images)
DLSS
Vid2vid
GauGAN
Partial Image Inpainting
Progress GAN
Pix2Pix

Speech
Deep Speech 2
Jasper
Tacotron
Wave2vec
WaveNet
WaveGlow

Language Modeling
BERT
GPT
TrellisNet
Gated Convolutions
BigLSTM/mLSTM
RoBERTa
Transformer XL

Translation
Convolutional Seq2Seq
Dynamic Convolutions
GNMT (RNN)
Levenshtein Transformer
Transformer (Self-Attention)



MAGMA Overview

COMPUTING LABORATORY

MAGMA

MAGMA

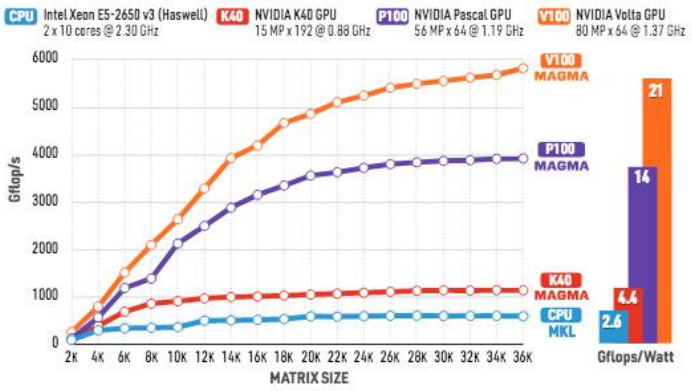
MAGMA (Matrix Algebra on GPU and Multicore Architectures) is a collection of next generation linear algebra libraries for heterogeneous architectures. MAGMA is designed and implemented by the team that developed LAPACK and ScaLAPACK, incorporating the latest developments in hybrid synchronization- and communication-avoiding algorithms, as well as dynamic runtime systems. Interfaces for the current LAPACK and BLAS standards are supported to allow computational scientists to seamlessly port any linear algebra reliant software components to heterogeneous architectures. MAGMA allows applications to fully exploit the power of current heterogeneous systems of multi/many-core CPUs and multi-GPUs to deliver the fastest possible time to accurate solution within given energy constraints. FIND OUT MORE AT **http://icl.utk.edu/magma**

HYBRID ALGORITHMS

MAGMA uses a hybridization methodology where algorithms of interest are split into tasks of varying granularity and their execution scheduled over the available hardware components. Scheduling can be static or dynamic. In either case, small non-parallelizable tasks, often on the critical path, are scheduled on the CPU, and larger more parallelizable ones, often Level 3 BLAS, are scheduled on the GPU.

PERFORMANCE & ENERGY EFFICIENCY

MAGMA LU factorization in double precision arithmetic



FEATURES AND SUPPORT

- **MAGMA 2.3** FOR CUDA
- **clMAGMA 1.4** FOR OpenCL
- **MAGMA MIC 1.4** FOR Intel Xeon Phi

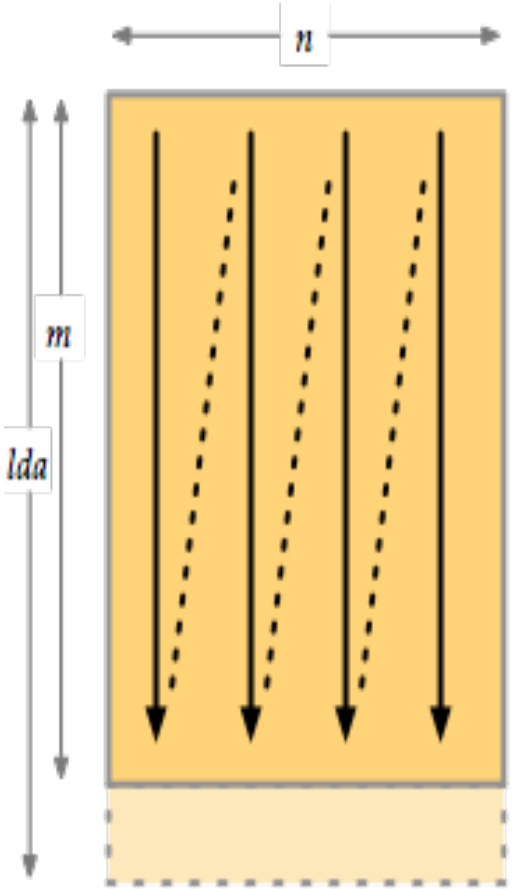
CUDA	OpenCL	Intel Xeon Phi	
●	●	●	Linear system solvers
●	●	●	Eigenvalue problem solvers
●	●		Auxiliary BLAS
●			Batched LA
●		●	Sparse LA
●	●	●	CPU/GPU Interface
●	●	●	Multiple precision support
●			Non-GPU-resident factorizations
●	●	●	Multicore and multi-GPU support
NEW			MAGMA Analytics/DNN
●	●	●	LAPACK testing
●	●	●	Linux
●	●		Windows
●	●		Mac OS

MAGMA 2.3 DRIVER ROUTINES

	MATRIX	OPERATION	ROUTINE	INTERFACES CPU	INTERFACES GPU
LINEAR EQUATIONS	GE	Solve using LU	{sdcz}gesv	✓	✓
		Solve using MP	{zc,ds}gesv		✓
	SPD/HPD	Solve using Cholesky	{sdcz}posv	✓	✓
		Solve using MP	{zc,ds}posv		✓
LEAST SQUARES	GE	Solve LLS using QR	{sdcz}gels		✓
		Solve using MP	{zc,ds}geqrsv		✓
STANDARD EVP	GE	Compute e-values, optionally e-vectors	{sdcz}geev	✓	
	SY/HE	Computes all e-values, optionally e-vectors	{sd}syevd	✓	✓
			{cz}heevd	✓	✓
		Range (D&C)	{cz}heevdx		✓
		Range (B&I It.)	{cz}heevx	✓	✓
		Range (MRRR)	{cz}heevr	✓	✓
STAND. SVP	GE	Compute SVD, optionally s-vectors	{sdcz}gesvd	✓	
			{sdcz}gesdd	✓	
GENERALIZED EVP	SPD/HPD	Compute all e-values, optionally e-vectors	{sd}sygvd	✓	
			{cz}hegvd	✓	
		Range (D&C)	{cz}hegvdx	✓	
		Range (B&I It.)	{cz}hegvx	✓	
		Range (MRRR)	{cz}hegvr	✓	

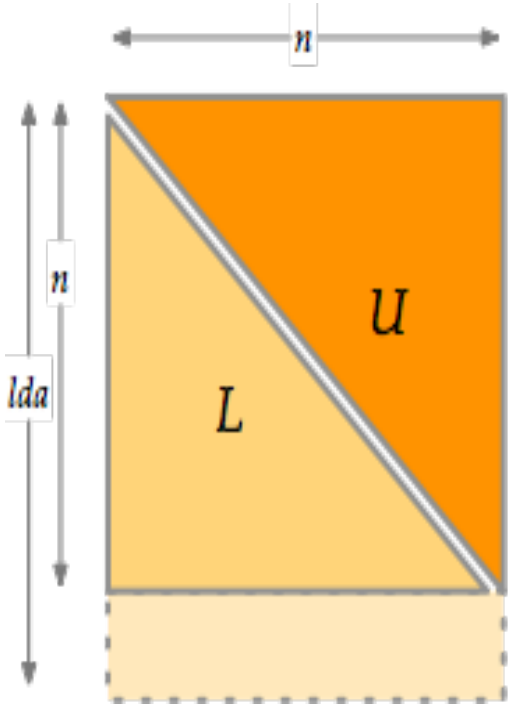
Matrix layout

General m-by-n matrix, LAPACK column-major layout



Symmetric / Hermitian / Triangular n-by-n matrix

- uplo = Lower or Upper
- Entries in opposite triangle ignored



Simple example

- Solve $AX = B$
  - **D**ouble precision, **GE**neral matrix, **S**ol**V**e (**DGESV**)
- Traditional LAPACK call
- Complete codes available at bit.ly/magma-tutorial

```
// tutorial0_lapack.cc
#include "magma_lapack.h"

int main( int argc, char** argv )
{
    int n = 100, nrhs = 10;
    int lda = n, ldx = n;
    double *A = new double[ lda*n ];
    double *X = new double[ ldx*nrhs ];
    int* ipiv = new int[ n ];

    // ... fill in A and X with your
data
    // A[ i + j*lda ] = A_ij
    // X[ i + j*ldx ] = X_ij

    // solve AX = B where B is in X
    int info;
    lapackf77_dgesv( &n, &nrhs,
                     A, &lda, ipiv,
                     X, &ldx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in X

    delete[] A;
    delete[] X;
    delete[] ipiv;
}
```

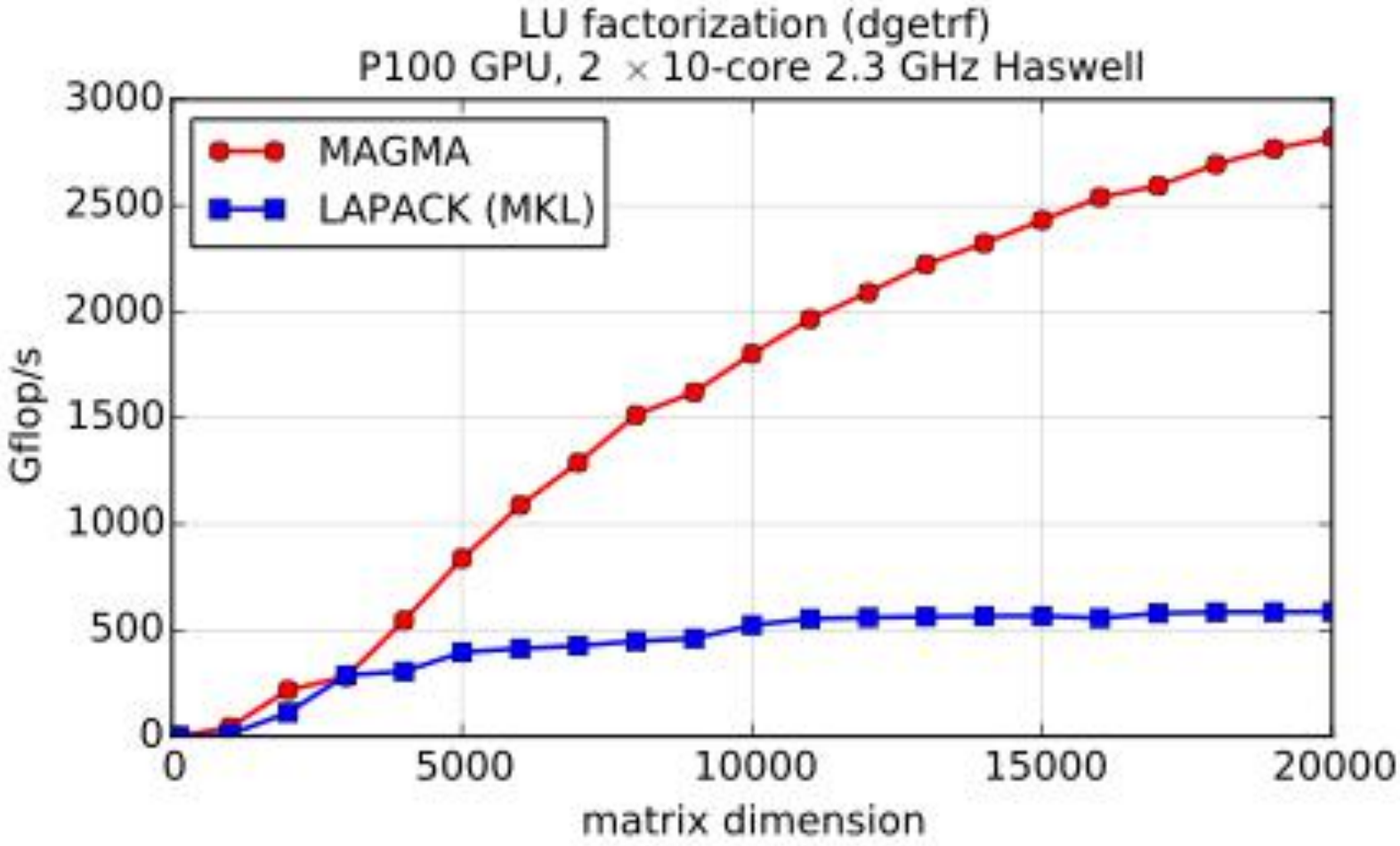
Simple example

- MAGMA CPU interface
  - Input & output matrices in CPU host memory
- Add MAGMA init & finalize
- MAGMA call direct replacement for LAPACK call

```
// tutorial1_cpu_interface.cc
#include "magma_v2.h"

int main( int argc, char** argv )
{
    magma_init();

    int n = 100, nrhs = 10;
    int lda = n, ldx = n;
    double *A = new double[ lda*n ];
    double *X = new double[ ldx*nrhs ];
    int* ipiv = new int[ n ];

    // ... fill in A and X with your
data
    // A[ i + j*lda ] = A_ij
    // X[ i + j*ldx ] = X_ij

    // solve AX = B where B is in X
    int info;
    magma_dgesv( n, nrhs,
                 A, lda, ipiv,
                 X, ldx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in X

    delete[] A;
    delete[] X;
    delete[] ipiv;

    magma_finalize();
}
```

```
magma> cd testing
magma/testing> ./testing_dgetrf -n 123 -n 1000:20000:1000 --
lapack --check
% MAGMA 2.2.0  compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% CUDA runtime 8000, driver 9000. OpenMP threads 20. MKL 2017.0.1, MKL threads 20.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16276.2 MiB memory, capability 6.0

%   M     N   CPU Gflop/s (sec)   GPU Gflop/s (sec)   |PA-LU|/(N*|A|)
%===========================================================================
  123   123      0.20 (   0.01)      0.40 (   0.00)   3.59e-18   ok       # warmup run
 1000  1000     10.40 (   0.06)     43.50 (   0.02)   2.76e-18   ok
 2000  2000    111.64 (   0.05)    218.26 (   0.02)   2.68e-18   ok
 3000  3000    288.38 (   0.06)    280.28 (   0.06)   2.65e-18   ok
 4000  4000    305.58 (   0.14)    545.90 (   0.08)   2.81e-18   ok
 5000  5000    396.16 (   0.21)    838.09 (   0.10)   2.71e-18   ok
 6000  6000    413.37 (   0.35)   1088.14 (   0.13)   2.71e-18   ok
 7000  7000    426.71 (   0.54)   1288.60 (   0.18)   2.67e-18   ok
 8000  8000    447.85 (   0.76)   1514.43 (   0.23)   2.66e-18   ok
 9000  9000    461.05 (   1.05)   1621.29 (   0.30)   2.87e-18   ok
10000 10000    524.06 (   1.27)   1802.39 (   0.37)   2.84e-18   ok
11000 11000    554.16 (   1.60)   1965.85 (   0.45)   2.84e-18   ok
12000 12000    559.33 (   2.06)   2090.42 (   0.55)   2.82e-18   ok
13000 13000    563.56 (   2.60)   2223.62 (   0.66)   2.80e-18   ok
14000 14000    566.58 (   3.23)   2323.04 (   0.79)   2.78e-18   ok
15000 15000    567.17 (   3.97)   2431.59 (   0.93)   2.77e-18   ok
16000 16000    556.86 (   4.90)   2539.66 (   1.08)   2.79e-18   ok
17000 17000    579.82 (   5.65)   2593.40 (   1.26)   2.75e-18   ok
18000 18000    584.93 (   6.65)   2694.57 (   1.44)   2.76e-18   ok
19000 19000    585.78 (   7.81)   2768.67 (   1.65)   2.75e-18   ok
20000 20000    587.08 (   9.08)   2821.48 (   1.89)   2.74e-18   ok
```

Testers: LU factorization (dgetrf)



Magma/2.5.2

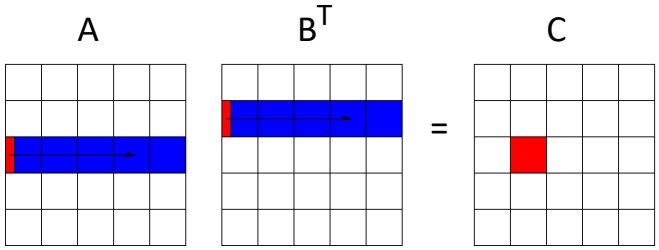
```
[wongk@gpu046 ~]$ nvidia-smi
Fri Oct  2 14:44:57 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  On   | 00000000:87:00.0 Off |                    0 |
| N/A   33C    P0    27W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
[wongk@gpu046 ~]$ module list
Currently Loaded Modulefiles:
  1) psc_path/1.1    2) slurm/default   3) intel/19.5      4) xdusage/2.1-1
[wongk@gpu046 ~]$ module load Magma/2.5.2
[wongk@gpu046 ~]$ module list
Currently Loaded Modulefiles:
  1) psc_path/1.1    2) slurm/default   3) intel/19.5      4) xdusage/2.1-1   5) Magma/2.5.2
[wongk@gpu046 ~]$ module show Magma/2.5.2
-------------------------------------------------------------------
/opt/modulefiles/Magma/2.5.2:


module-whatis Magma 2.5.2
prepend-path LD_LIBRARY_PATH /opt/packages/Magma/magma-2.5.2/lib
-------------------------------------------------------------------
```

Magma/2.5.2 on bridges GPU

```
[wongk@gpu046 ~]$ module load cuda/10.1
[wongk@gpu046 ~]$ module list
Currently Loaded Modulefiles:
  1) psc_path/1.1    2) slurm/default   3) intel/19.5      4) xdusage/2.1-1   5) Magma/2.5.2     6) cuda/10.1
[wongk@gpu046 ~]$ cd /opt/packages/Magma/magma-2.5.2/testing

[wongk@gpu046 testing]$ ./testing_dgemm -N 5000:10000:500 --lapack
% MAGMA 2.5.2  compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% CUDA runtime 10010, driver 10020. OpenMP threads 16. MKL 2019.0.5, MKL threads 16.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16280.9 MiB memory, capability 6.0
% Fri Oct  2 15:04:44 2020
% Usage: ./testing_dgemm [options] [-h|--help]
% If running lapack (option --lapack), MAGMA and cuBLAS error are both computed
% relative to CPU BLAS result. Else, MAGMA error is computed relative to cuBLAS result.
% transA = No transpose, transB = No transpose
%   M     N     K   MAGMA Gflop/s (ms)  cuBLAS Gflop/s (ms)   CPU Gflop/s (ms)  MAGMA error  cuBLAS error
%=================================================================================================================
 5000  5000  5000   3646.66 (  68.56)    4299.59 (  58.15)    515.86 ( 484.63)    2.16e-17     2.16e-17   ok
 5500  5500  5500   3746.67 (  88.81)    4479.56 (  74.28)    531.21 ( 626.40)    2.16e-17     2.16e-17   ok
 6000  6000  6000   3715.05 ( 116.28)    4536.77 (  95.22)    520.63 ( 829.77)    2.12e-17     2.12e-17   ok
 6500  6500  6500   3693.40 ( 148.71)    4524.19 ( 121.40)    531.53 (1033.33)    2.06e-17     2.06e-17   ok
 7000  7000  7000   3673.32 ( 186.75)    4486.80 ( 152.89)    528.31 (1298.47)    1.99e-17     1.99e-17   ok
 7500  7500  7500   3669.31 ( 229.95)    4487.55 ( 188.02)    527.02 (1600.97)    1.92e-17     1.92e-17   ok
 8000  8000  8000   3704.85 ( 276.39)    4527.77 ( 226.16)    532.93 (1921.45)    1.85e-17     1.85e-17   ok
 8500  8500  8500   3699.27 ( 332.03)    4512.24 ( 272.20)    524.18 (2343.21)    1.94e-17     1.94e-17   ok
 9000  9000  9000   3687.36 ( 395.40)    4515.05 ( 322.92)    523.77 (2783.68)    2.06e-17     2.06e-17   ok
 9500  9500  9500   3669.67 ( 467.28)    4502.94 ( 380.81)    521.39 (3288.80)    2.12e-17     2.12e-17   ok
10000 10000 10000   3660.58 ( 546.36)    4491.71 ( 445.27)    515.41 (3880.44)    2.16e-17     2.16e-17   ok
```

SGEMM Example

2008. Volkov and Demmel. ***Benchmarking GPUs to tune dense linear algebra***, SC08
http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf



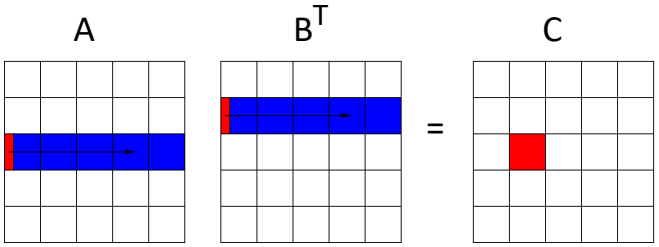
* **Small** red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads

```
// GPU kernel: compute C = alpha A B' + beta C
__global__ void sgemmNT( const float *A, int lda,
                         const float *B, int ldb,
                         float* C, int ldc, int k,
                         float alpha, float beta  )
{
    int inx = threadIdx.x;
    int iny = threadIdx.y;
    int ibx = blockIdx.x * 32;
    int iby = blockIdx.y * 32;

    A += ibx + inx + __mul24( iny, lda );
    B += iby + inx + __mul24( iny, ldb );
    C += ibx + inx + __mul24( iby + iny, ldc );

    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    for( int i = 0; i < k; i += 4 )
    {
        __syncthreads();
        __shared__ float a[4][32];
        __shared__ float b[4][32];

        a[iny][inx] = A[i*lda];
        a[iny+2][inx] = A[(i+2)*lda];

        b[iny][inx]   = B[i*ldb];
        b[iny+2][inx] = B[(i+2)*ldb];
        __syncthreads();

        for( int j = 0; j < 4; j++ )
        {
            float _a = a[j][inx];
            float *_b = &b[j][0] + iny;

            c[0] += _a*_b[0];
            c[1] += _a*_b[2];
            c[2] += _a*_b[4];
            c[3] += _a*_b[6];
            c[4] += _a*_b[8];
            c[5] += _a*_b[10];
            c[6] += _a*_b[12];
            c[7] += _a*_b[14];
            c[8] += _a*_b[16];
            c[9] += _a*_b[18];
            c[10] += _a*_b[20];
            c[11] += _a*_b[22];
            c[12] += _a*_b[24];
            c[13] += _a*_b[26];
            c[14] += _a*_b[28];
            c[15] += _a*_b[30];
        }
    }

    for( int i = 0; i < 16; i++, C += 2*ldc )
        C[0] = alpha * c[i] + beta * C[0];
}


void ourSgemm (char transa, char transb,
               int m, int n, int k,
               float alpha,
               const float *A, int lda,
               const float *B, int ldb,
               float beta,
               float *C, int ldc)
{
    assert( (transa == 'N' || transa == 'n') &&
            (transb == 'T' || transb == 't') &&
            ((m|n|k|lda|ldb)&31) == 0,
            "unsupported parameters in ourSgemm()" );

    dim3 grid( m/32, n/32, 1 );
    dim3 threads2( 32, 2, 1 );
    sgemmNT<<<grid, threads2>>>( A, lda,
                                 B, ldb,
                                 C, ldc,
                                 k, alpha, beta );
}
```

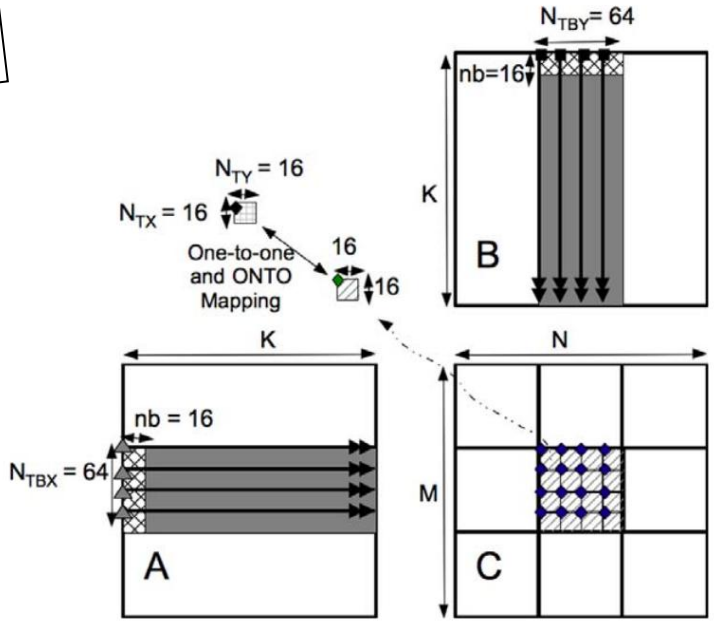
GEMM in MAGMA

2010. R. Nath, S. Tomov and J. Dongarra. ***An improved MAGMA GEMM for Fermi Graphics Processing Units***, The International Journal of High Performance Computing Applications.
http://www.netlib.org/utk/people/JackDongarra/journals/208_2010_an-improved-magma-gemm-for-fermi-gpus.pdf

- **Add register blocking**
- **Parameterized for autotuning for particular size GEMMs and portability across GPUs**



* **Small** red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads

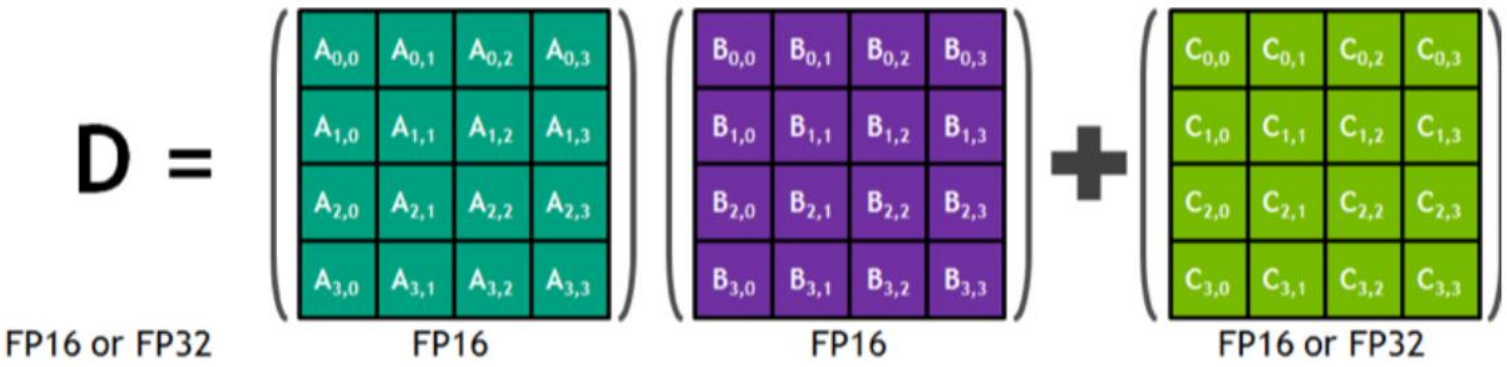


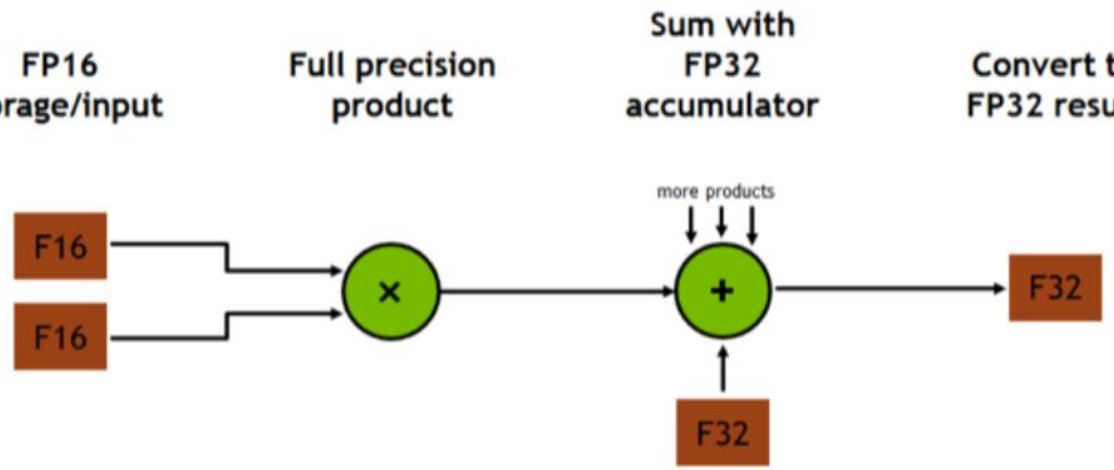
A thread computes part of a row (16 values) of the C block

A thread computes a block of C (4 x 4 values in this case)

GPU Tensor Cores – accelerated FP16 matrix-multiply-and-accumulate units

- Up to 120 teraFLOP/s on NVIDIA Volta V100 GPUs





Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel

```
for (int i=0; i<batchcount; i++)
  la_computation(Ai, …)
```

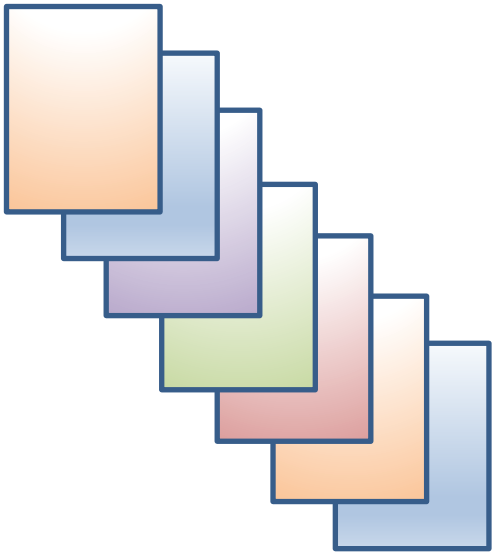
Call a numerical library (e.g., BLAS, LAPACK, etc.) to compute A_i in parallel

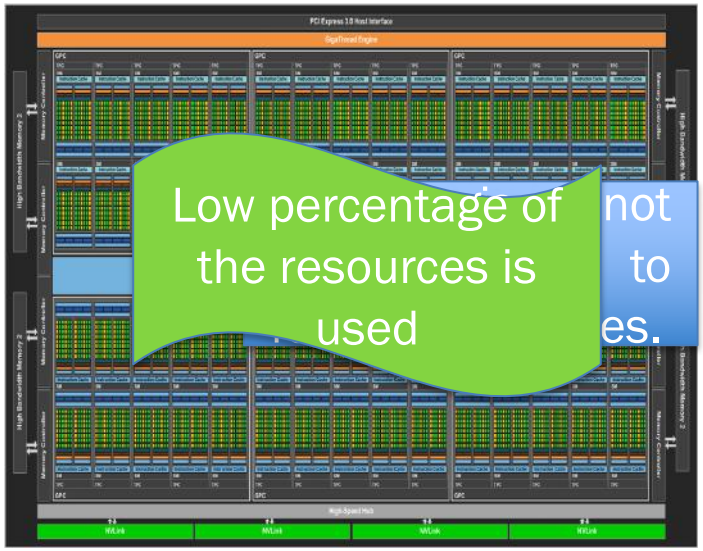
Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

```
for (int i=0; i<batchcount; i++)
  dgemm(…)
```





Batched Linear Algebra Computations

Non-batched computation

- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resourc** by assigning a matrix to each group of core/TB to opera on it independently

```
for (int i=0; i<batchcount; i++)
  la_computation(A_i, …)
```

```
Batched_la_computation(A,batchcount,…)
```

Batched Linear Algebra Computations

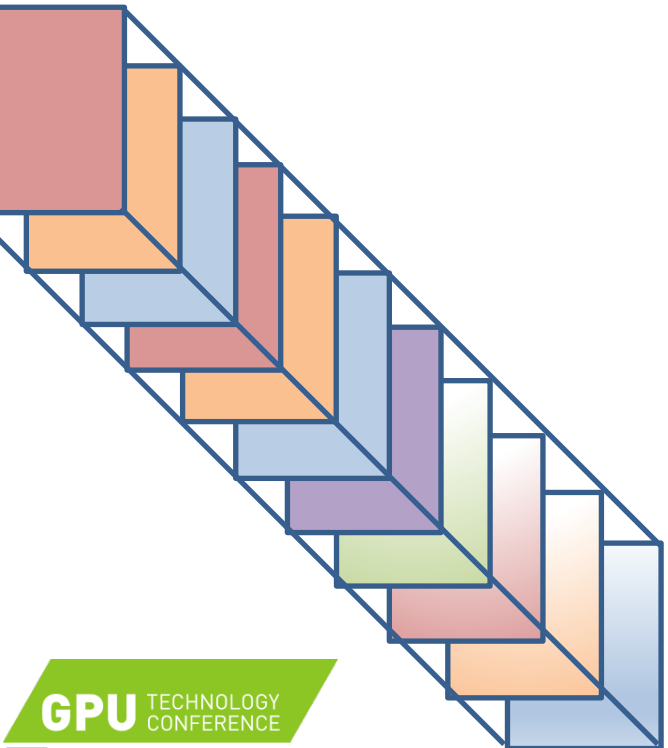
Non-batched computation

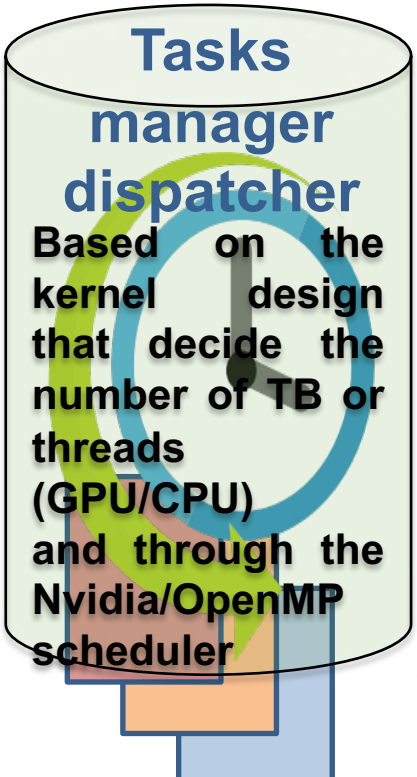
- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

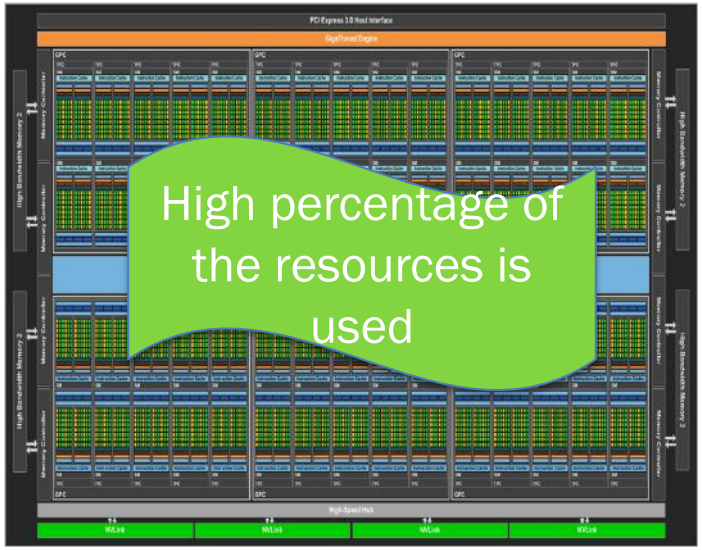
Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resourc** by assigning a matrix to each group of core/TB to opera on it independently

`Batched_dgemm(...)`







Batched Linear Algebra Computations

Non-batched computation

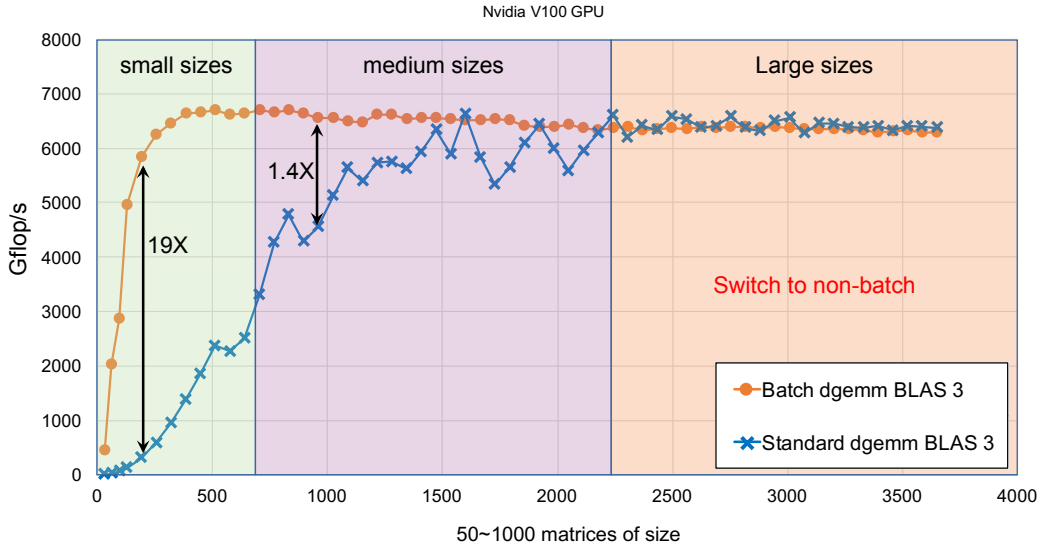
- **Data parallel computation** (e.g., over independent matrices A_i)
- **Loop over the matrices one by one** and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation

- **Given an interface as a single Batched routine**
- **Distribute all the matrices over the available resourc** by assigning a matrix to each group of core/TB to opera on it independently

```
for (int i=0; i<batchcount; i++)
  la_computation(A_i, …)
```

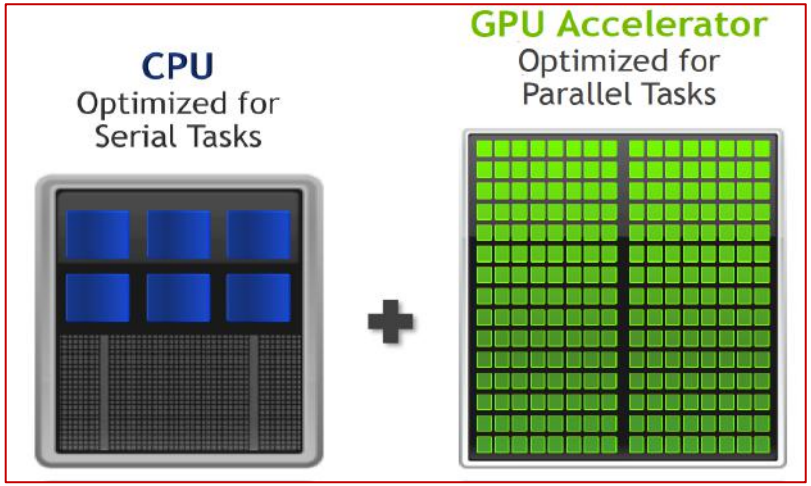
```
Batched_la_computation(A,batchcount,…)
```

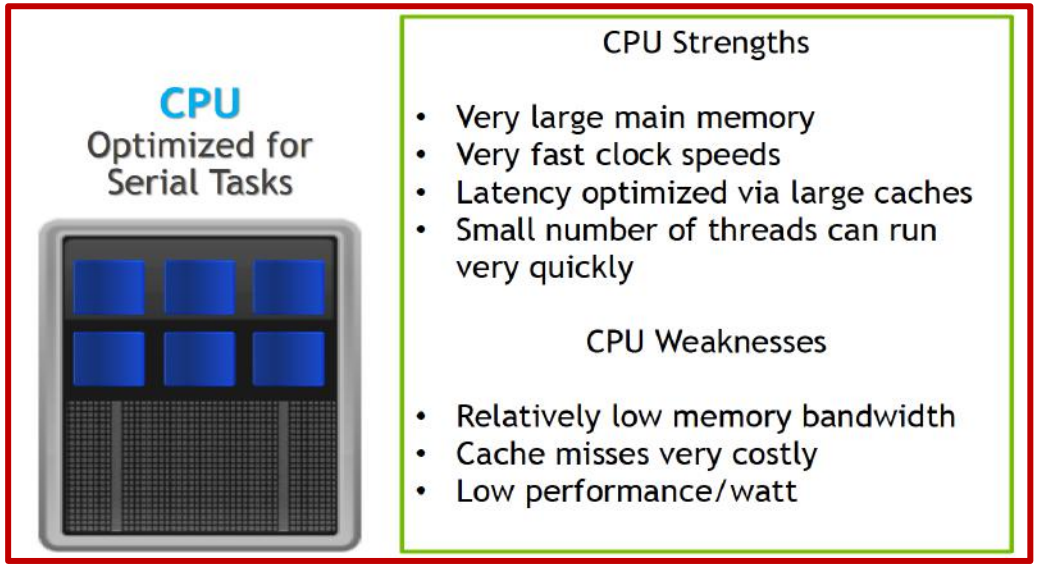


Matrix sizes (fixed) in the batch

Batch size 1,000 Batch size 300 Batch size 50

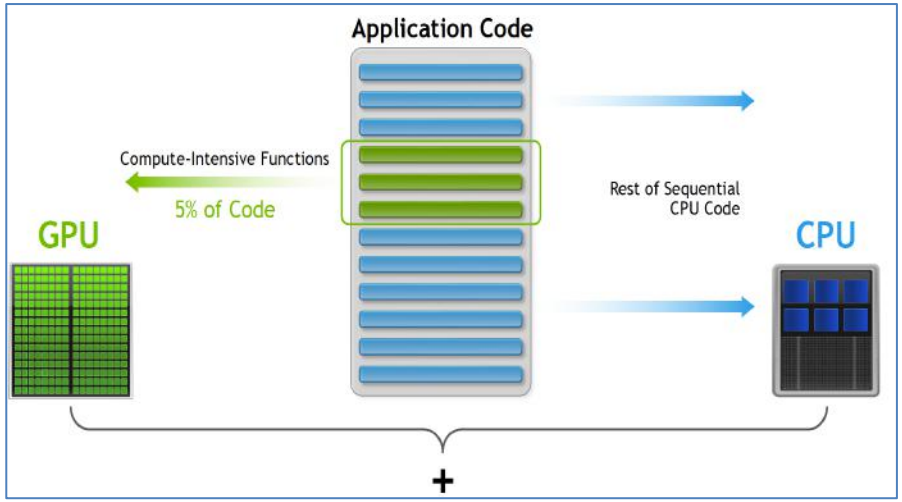
Programming style of GPU

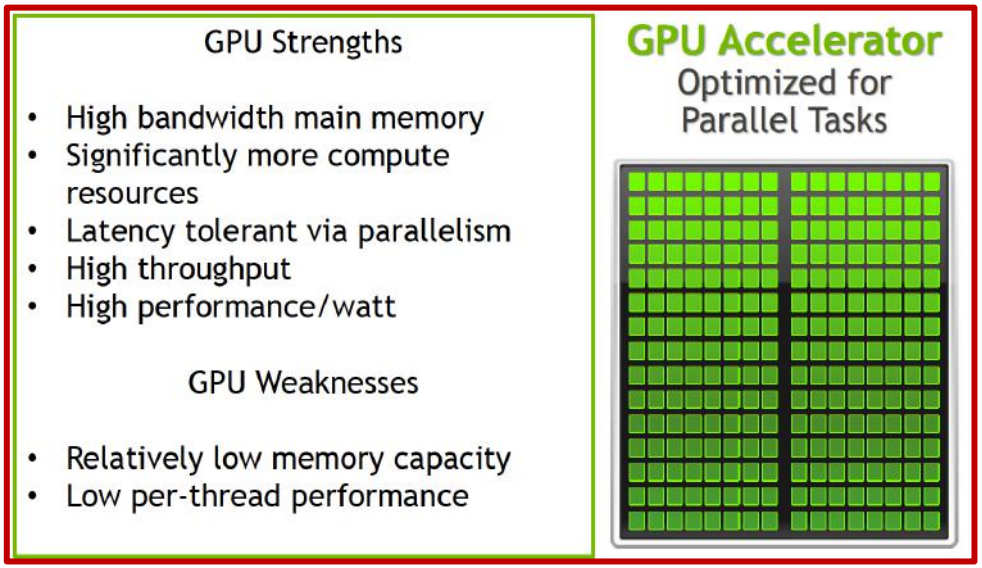




Individual GPU cores are not optimized like CPU cores for serial processing
Latency hidden by switching between threads/warps
Maximum number of threads in flight on A100: 221,184 (108SM x 2048 threads/SM)

CPU is a Latency Reducing Architecture

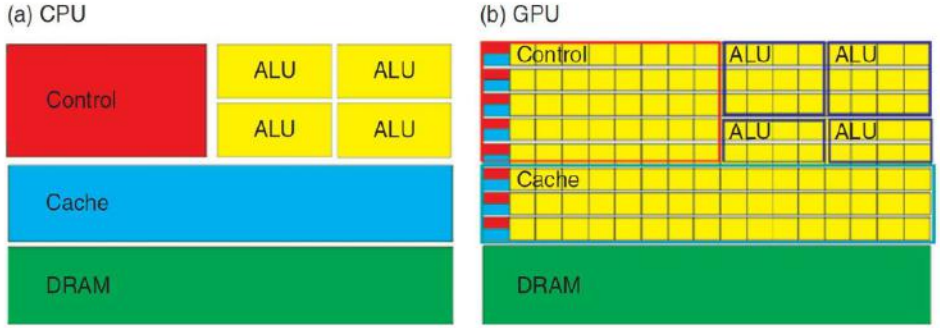




GPU is all about hiding Latency

Programming flow of GPU

GPU vs CPU architecture

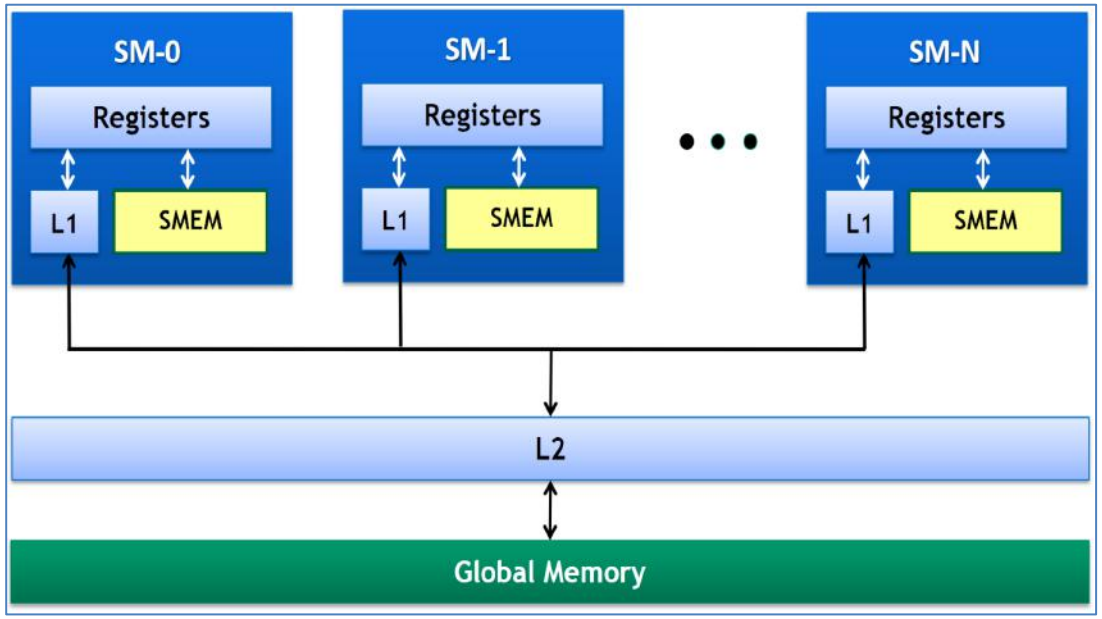


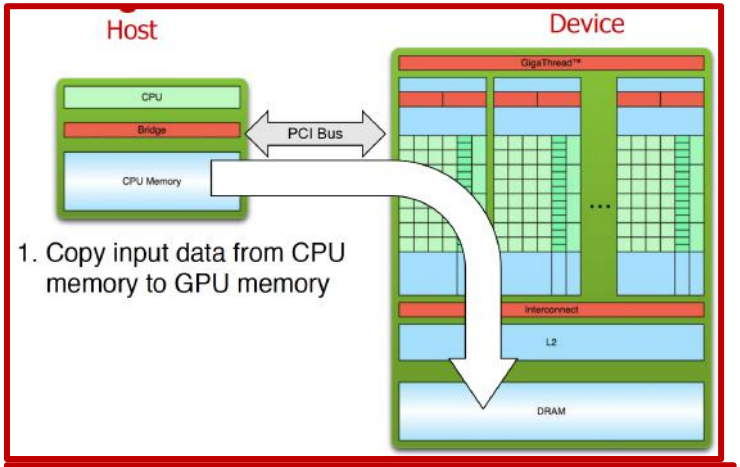
CPU

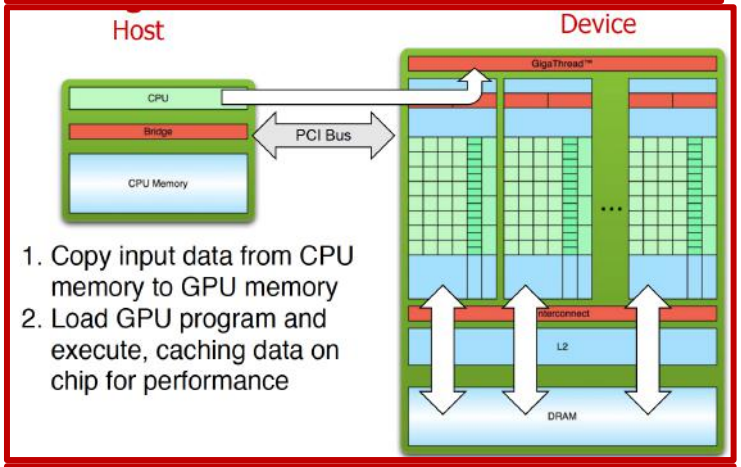
- Few processing cores with sophisticated hardware
- Multi-level caching
- Prefetching
- Branch prediction

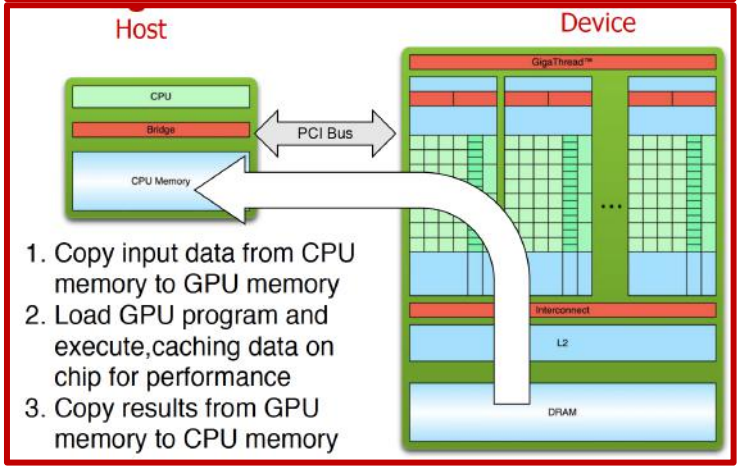
GPU

- Thousands of simplistic compute cores (packaged into a few multiprocessors)
- Operate in lock-step
- Vectorized loads/stores to memory
- Need to manage memory hierarchy



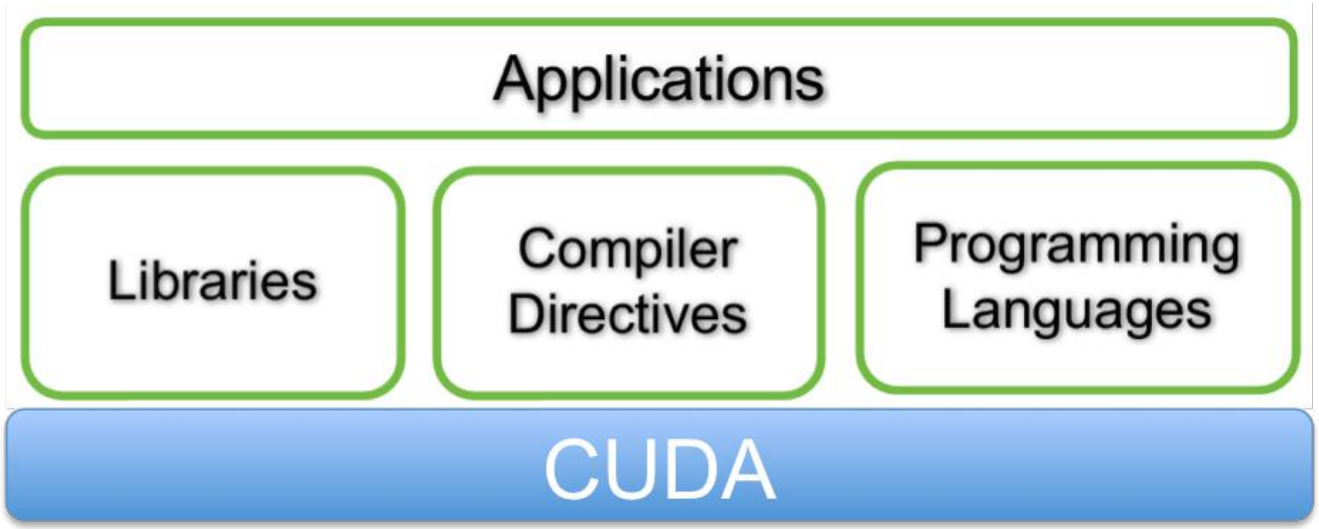






Introduction to CUDA libraries

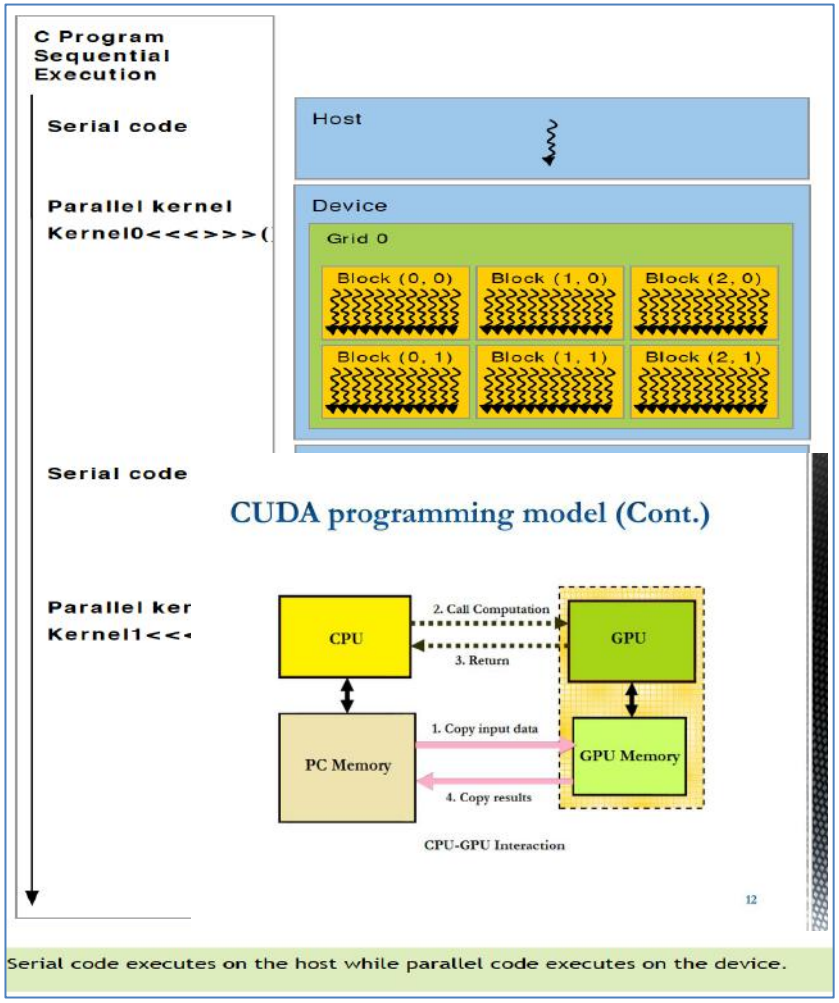
CUDA Libraries to accelerate the application development



Three Ways to Accelerate Applications by CUDA

- ✓ Ease to use: Using libraries enables GPU acceleration without in depth knowledge of GPU programming
- ✓ “Drop-in”: Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- ✓ Quality: Libraries offer high-quality implementations of functions encountered in a broad range of applications

CUDA Code Implementation (hello World)




```
#include <stdio.h>
/*This program prints "Hello World from GPU! from 10 CUDA
threads running on the GPU. – hello.cu */
__global__ void helloFromGPU()
{
   printf("Hello World from GPU!\n"); }
```

```
int main(int argc, char **argv)
{
   printf("Hello World from CPU!\n");
   helloFromGPU<<<1, 10>>>();
   CHECK(cudaDeviceReset());
   return 0;}
```

__global__ : A qualifier added to standard C. This alerts the compiler that a function should be compiled to run on a device (GPU) instead of host (CPU).
2.Function helloFromGPU() is called from host code.

```
$ ./a.out
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
…
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
```

<<<grid, thread>>>;
<<<1, 10>>>;
<<<10, 1>>>;
<<<10, 10>>>;

```
nvcc –o <exe> <hello.cu>
-I/cudaIncludePath –L/CudaLibPath –lcuda –lcudart
```

directories for #include files

directories for libraries

libraries to be linked

Device functions (helloFromGPU()) processed by NVIDIA nvcc compiler
Host functions (e.g. main()) processed by standard host compiler like gcc

CUDA program Skeleton

What are the key components in a CUDA C program?

Example P0-0_hellofromGPU.cu

```
#include <stdio.h>
/*A simple introduction to programming in
CUDA. This program prints "Hello World
from GPU! from 10 CUDA threads running on
the GPU. */
__global__ void helloFromGPU()
{
   printf("Hello World from GPU!\n");
}
```

```
int main(int argc, char **argv)
{
   printf("Hello World from CPU!\n");
   helloFromGPU<<<1, 10>>>();
   CHECK(cudaDeviceReset());
   return 0;
}
```

"`__global__`" declares a function:

- Called "Kernel"
- Indicates the kernel code will be run on device(GPU)

- ✓ A CUDA program has two parts: host code executing on host and interfaces to device; Kernel code which runs on device.
- ✓ Host code starts running, and when first encounter device kernel, device code is sent to device and function is launched on device

"helloFromGPU<<<1, 10>>>()":

- Called "Kernel launch"
- Triple angle brackets mark a call from host code to device code
- Numbers inside triple-angle-brackets defines kernel's execution configuration on device

CUDA code Scalar Addition

/*Code example:

scalar addition on GPU*/

- ✓ The kernel “add()” will execute on the device and will be called from the host
- ✓ add() runs on the device, so a, b and c must point to device memory

- CUDA source Code P0-1_ScalarAdd.cu

- Compiling the code

```
nvcc –o <exe> <source_file>
-I/cudaIncludePath
–L/CudaLibPath –lcuda –lcudart
```

- Executing a CUDA Program

```
$ ./a.out
2+7=9
```

```
#include <stdio.h>
/*
* This program performs scalar addition "a+b=c" on the GPU.
*/
__global__ void add(int *a, int *b, int *c)
{ /**/
*c = *a + *b;}
```

```
int main(void)
{
int a, b, c; // host copies of a, b, c
int *d_a, *d_b, *d_c; // memory locations of device copies of a, b, c
int size = sizeof(int);
// Allocate space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
// Setup input values
a = 2;
b = 7;
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
add<<<1, 1>>>(d_a, d_b, d_c);
// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
// Cleanup
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
printf("%i + %i = %i\n", a, b, c);
return 0;
}
```

CUDA program structure

```
__global__ void some_kernel(...){
...
}
int main (void){
  // Declare all variables.
  ...
  // Allocate host memory and device memory .
  ...
  // Write data to host memory
  ...
  //copy them to device memory.
  ...
  // Kernel invocation code to launch kernel on device.
  ...
  // Copy GPU results in device memory back to host memory.
  ...
  // Free allocated host memory and device memory
  ...
}
```

Skeleton of a CUDA program

- Write and launch a kernel for device
- Parameters and data transfer between host and device

How to run a code in parallel on device?

CUDA Example Vector Addition

/*Example-1-2: Vector Addition by GPU*/

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad \Rightarrow \quad c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{bmatrix}$$

- Get the CUDA source Code: P0-2_vectorAdd.cu

- **Compiling the Code**

nvcc –o <exe> <source_file> -I/cudaIncludePath –L/CudaLibPath –lcuda –lcudart

directories for #include files

directories for libraries

libraries to be linked

Vector Addition by GPU

```
#include <stdio.h>

/**
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C. The 3 vectors
have the same
 * number of elements numElements.
 */
__global__ void vectorAdd(const float *A, const float *B, float
*C,
                          int numElements)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;

  if (i < numElements) {
    C[i] = A[i] + B[i];
  }
}
```

```
/**
 * Host main routine
 */
int main(void)
{
// Print the vector length to be used, and compute its size
  int numElements = 50000;
  size_t size = numElements * sizeof(float);

  // Allocate the host input vector A
  float *h_A = (float *)malloc(size);

  // Allocate the host input vector B
  float *h_B = (float *)malloc(size);

  // Allocate the host output vector C
  float *h_C = (float *)malloc(size);

  // Initialize the host input vectors
  for (int i = 0; i < numElements; ++i) {
    h_A[i] = rand() / (float)RAND_MAX;
    h_B[i] = rand() / (float)RAND_MAX;

  // Allocate the device input vector A
  float *d_A = NULL;
  err = cudaMalloc((void **)&d_A, size);

  // Allocate the device input vector B
  float *d_B = NULL;
  err = cudaMalloc((void **)&d_B, size);

  // Allocate the device output vector C
  float *d_C = NULL;
  err = cudaMalloc((void **)&d_C, size);
```

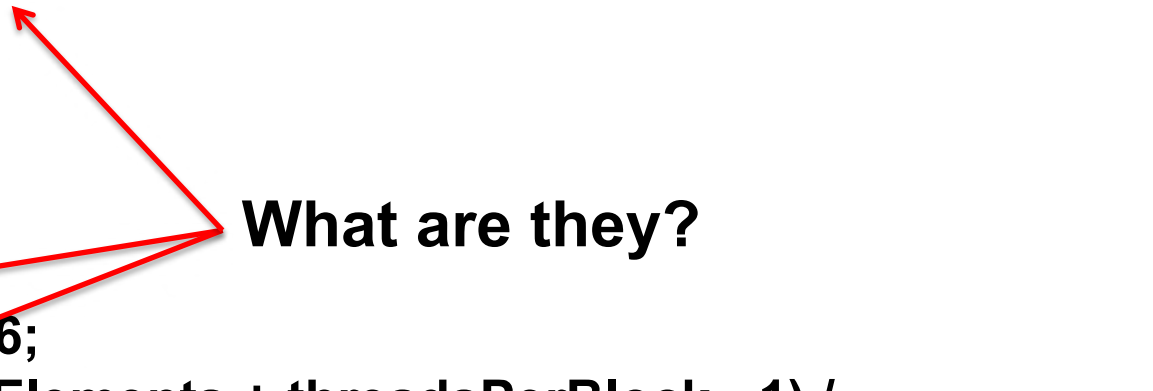
```
// Copy the host input vectors A and B in host memory to the device input vectors in device memory
  printf("Copy input data from the host memory to the CUDA device\n");
  err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  // Launch the Vector Add CUDA Kernel
  int threadsPerBlock = 256;
  int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;
  printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid, threadsPerBlock);
  vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
  err = cudaGetLastError();

  // Copy the device result vector in device memory to the host result vector in host memory.
  printf("Copy output data from the CUDA device to the host memory\n");
  err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

  // Verify that the result vector is correct
  for (int i = 0; i < numElements; ++i) {
    printf("%.5f + %.5f = %.5f\n", h_A[i], h_B[i], h_C[i]);
  }
  printf("Test PASSED\n");

 // Free device global memory
  err = cudaFree(d_A);
  err = cudaFree(d_B);
  err = cudaFree(d_C);

  // Free host memory
  free(h_A);
  free(h_B);
  free(h_C);

  printf("Done\n");
  return 0;
}
```

```
~/NVIDIA_CUDA-version_Samples/0_Simple/$ ./vectorAdd

[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
“     a     +     b     =     c     ”
0.84019 + 0.39438 = 1.23457
0.78310 + 0.79844 = 1.58154
0.91165 + 0.19755 = 1.10920
0.33522 + 0.76823 = 1.10345
.....
0.78900 + 0.84517 = 1.63417
0.66494 + 0.30764 = 0.97258
Test PASSED
Done
```

Thread hierarchy

Back to example P0-2_vectorAdd.cu

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements)
   {C[i] = A[i] + B[i];}
}
int main(void){
   …
   int threadsPerBlock = 256;
   int blocksPerGrid =(numElements + threadsPerBlock - 1) /
threadsPerBlock;
   vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
numElements);
    …
}
```

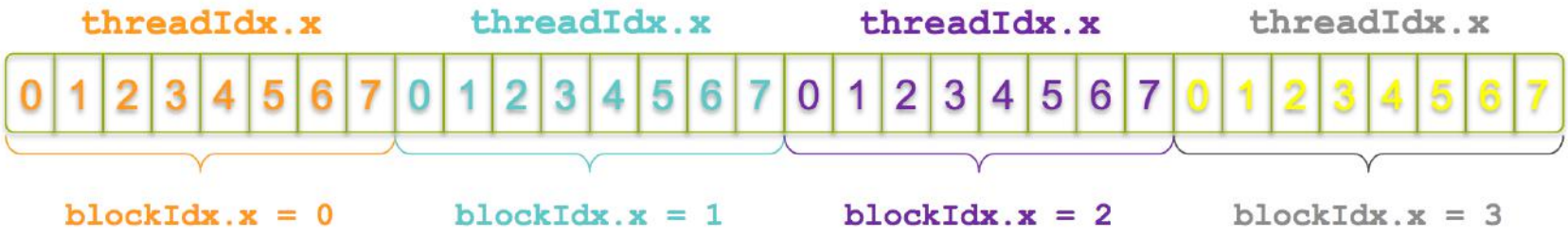


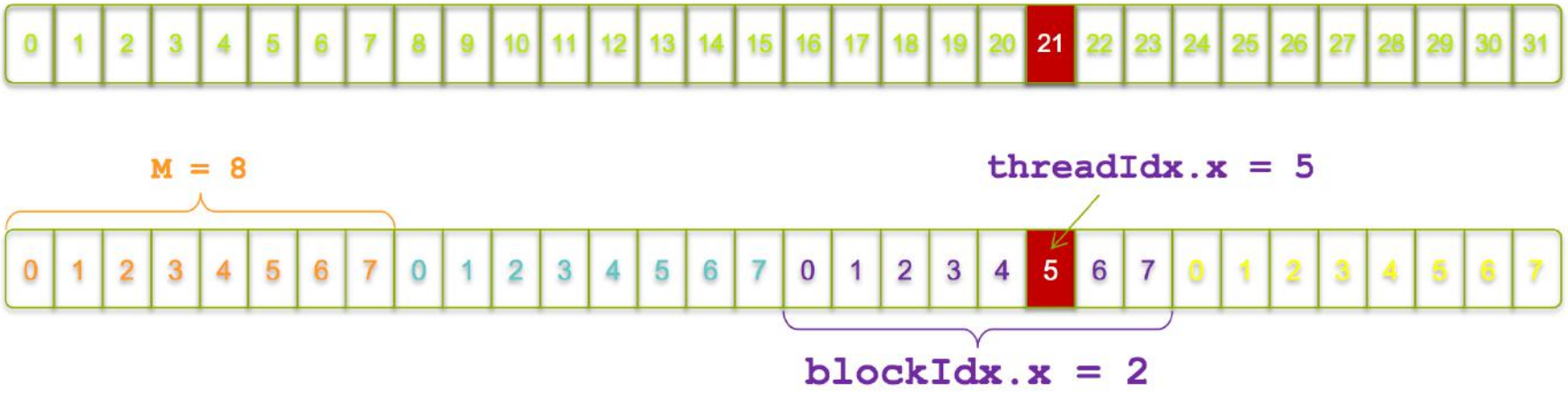
Thread Block index structure

With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x+ blockIdx.x* M;
```

Consider indexing an array with one element per thread (8 threads/block






```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2       * 8;
          = 21;
```

CUDA Thread Basics

Kernel

- ✓ In CUDA, a kernel is code (typically a function), that can be executed on the GPU.
- ✓ The kernel code operates in lock-step on the multiprocessors of the GPU
- ✓ (In so-called warps, currently consisting of 32 threads)

Thread

- ✓ A thread is an execution of a kernel with a given index.
- ✓ Each thread uses its index to access a subset of data (e.g. array) to operate on.

Block

- ✓ Threads are grouped into blocks, which are guaranteed to execute on the same multiprocessor.
- ✓ Threads within a thread block can synchronize and share data

Grid

- ✓ Thread blocks are arranged into a grid of blocks.
- ✓ (number of threads per block) x (number of blocks) = total number of running threads.

Threads, blocks, grids, warps

Grids

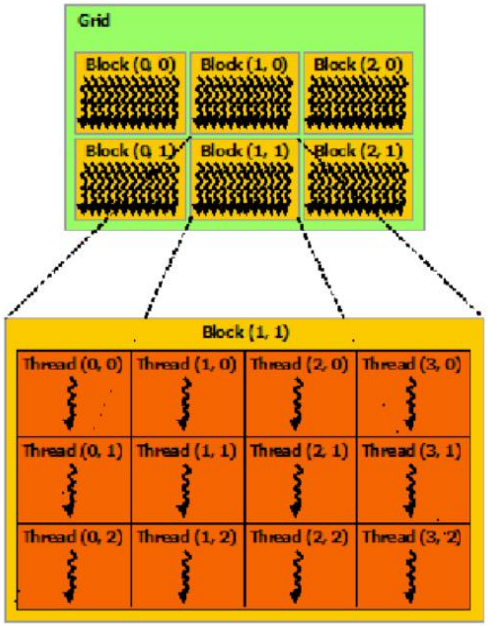
- ✓ Grids map to GPUs

Blocks

- ✓ Blocks map to the multiprocessors (MP)
- ✓ Blocks are never split across MPs
- ✓ Multiple blocks can execute simultaneously on an MP

Threads

- ✓ Threads are executed on stream processors (GPU cores)
- ✓ Warps are groups of threads that execute simultaneously, in lock-step (currently 32, not guaranteed to remain fixed).



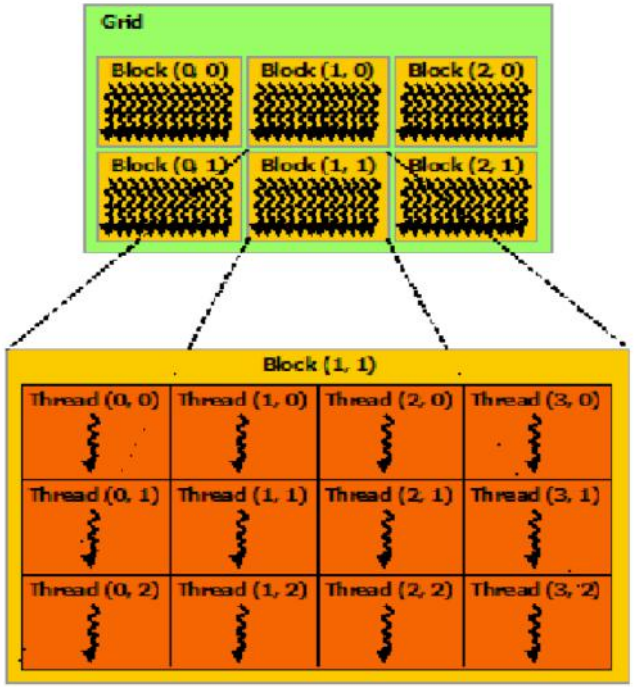
CUDA Thread <<< Block, thread>>>

- ✓ The information between the triple chevrons <<< x, y >>> is the *execution configuration*, which dictates how many device threads execute the kernel in parallel. In the CUDA programming model it launches a kernel with a *grid* of *thread blocks*.
- ✓ The first argument in the execution configuration specifies the number of thread blocks in the grid, and the second specifies the number of threads in a thread block.
- ✓ Thread blocks and grids can be made one-, two- or three-dimensional by passing dim3 (a simple struct defined by CUDA with x, y, and z members) values for these arguments,

```
kernel<<< the number of thread blocks in the grid ,  number of threads in a thread block>>>( . )
```

- ✓ CUDA defines the variables blockDim, blockIdx, and threadIdx. These predefined variables are of type dim3.
- ✓ The predefined variable blockDim contains the dimensions of each thread block as specified in the second execution configuration parameter for the kernel launch.
- ✓ The predefined variables threadIdx and blockIdx contain the index of the thread within its thread block and the thread block within the grid, respectively.
- ✓ The expression: int i = blockDim.x * blockIdx.x + threadIdx.x generates a global index that is used to access elements of the arrays.

CUDA Thread <<< Block, thread>>>



Block indexes

- ✓ blockIdx.x, blockIdx.y, blockIdx.z, and block ID in the x-, y-, and z-axis.

Thread indexes

- ✓ blockDim.x, blockDim.y, blockDim.z, threadIdx.x, threadIdx,y, threadIdx.z

Example in the figure is executing 72 threads

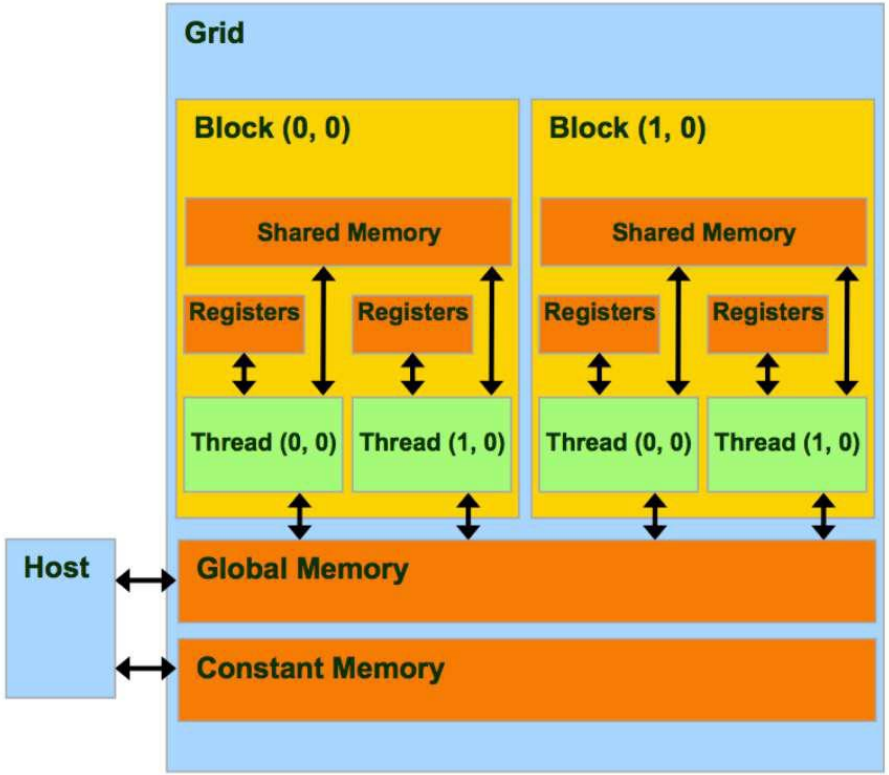
- ✓ (3 x 2) blocks = 6 blocks, (4 x 3) threads per block = 12 threads per block

CUDA memory hierarchy

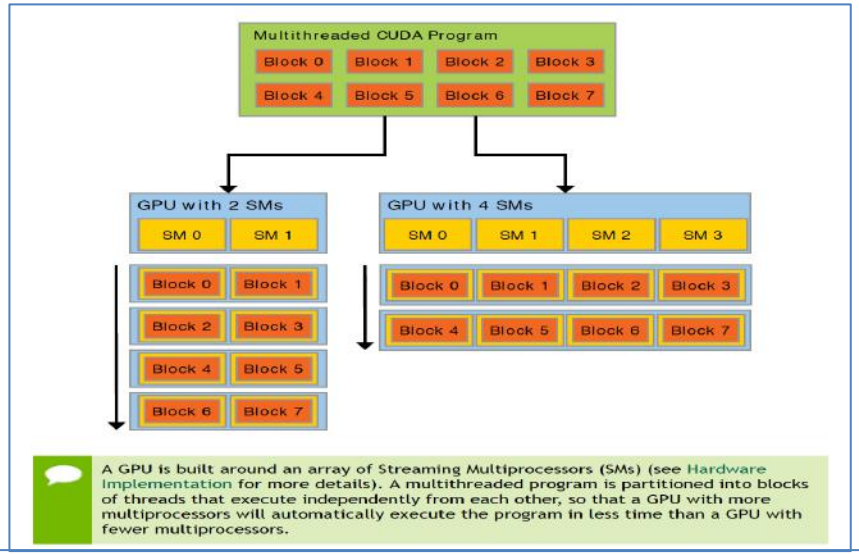
- ✓ Host memory (x86 server)
- ✓ Device memory (GPU)

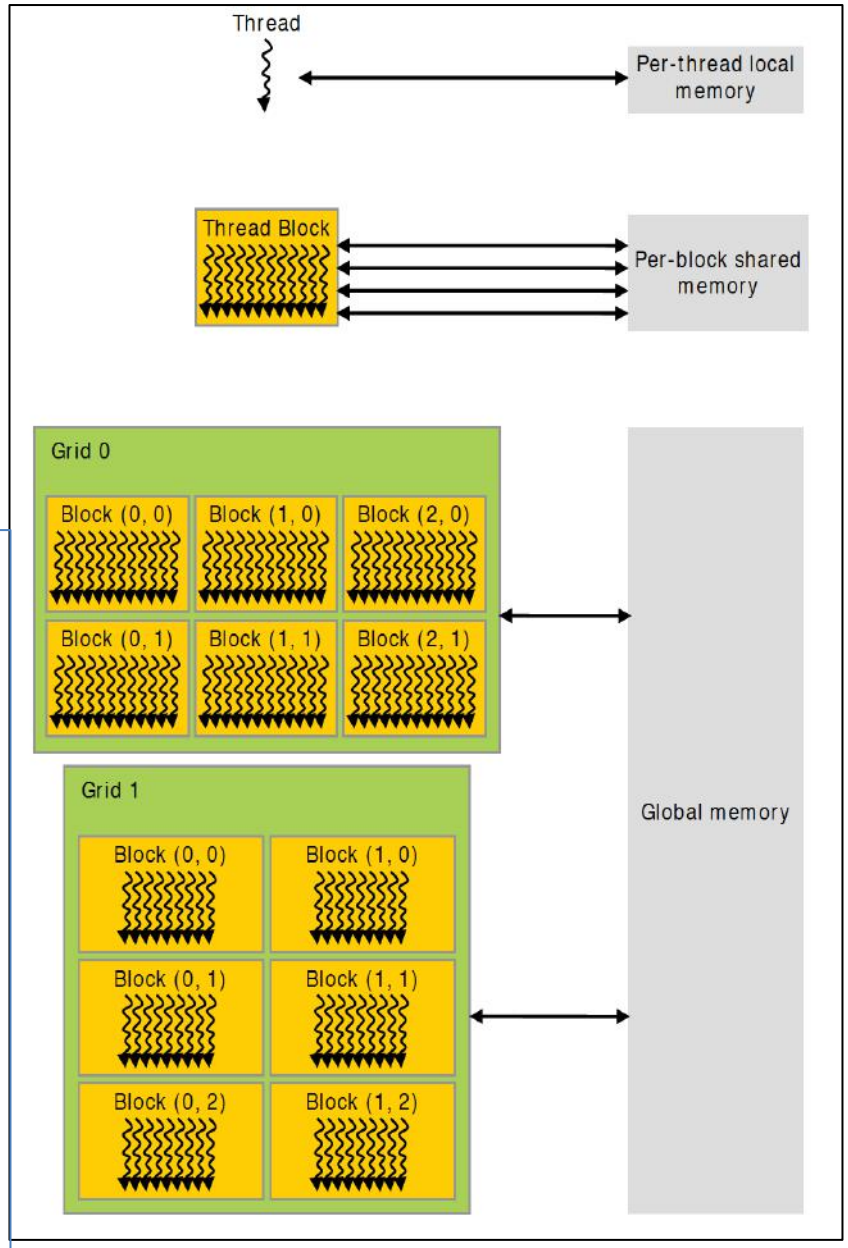
Device memory

- ✓ Global memory, visible to all threads, slow
- ✓ Shared memory, visible to all threads in a block, fast on-chip
- ✓ Registers, per-thread memory, fast on-chip
- ✓ Local memory, slow, stored in Global Memory space
- ✓ Constant memory, visible to all threads, read only, off-chip, cached broadcast to all threads



CUDA Thread Block Memory Access





Avoid data transfers between CPU and GPU

- ✓ **These are slow due to low PCI express bus bandwidth**

Minimize access to global memory

- ✓ **Hide memory access latency by launching many threads**

Take advantage of fast shared memory by tiling data

- ✓ **Partition data into subsets that fit into shared memory**
- ✓ **Handle each data subset with one thread block**
- ✓ **Load the subset from global to shared memory using multiple threads**

to exploit parallelism in memory access

- ✓ **Perform computation on data subset in shared memory**

(each thread in thread block can access data multiple times)

- ✓ **Copy results from shared memory to global memory**

Nvidia GPU Computing Applications

- ✓ A parallel computing platform that leverages NVIDIA GPUs
- ✓ Accessible through CUDA libraries, compiler directives, application programming interfaces, and extensions to several programming languages (**C/C++**, Fortran, and Python).

GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

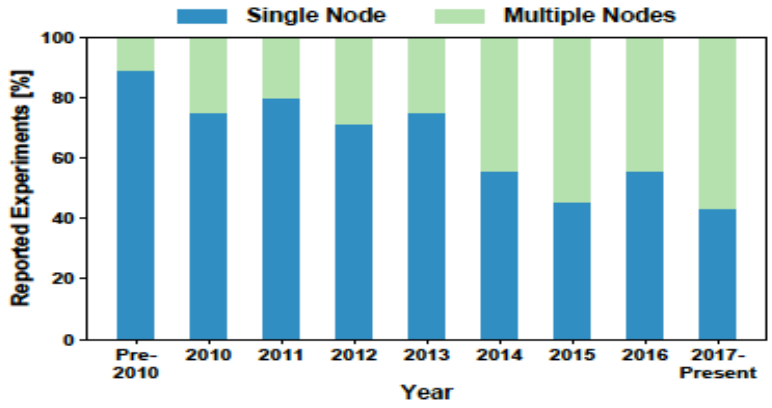
Programming Languages

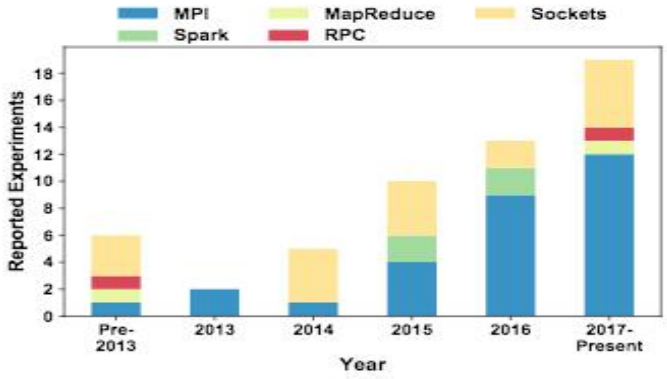
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
				

Distributed DNN





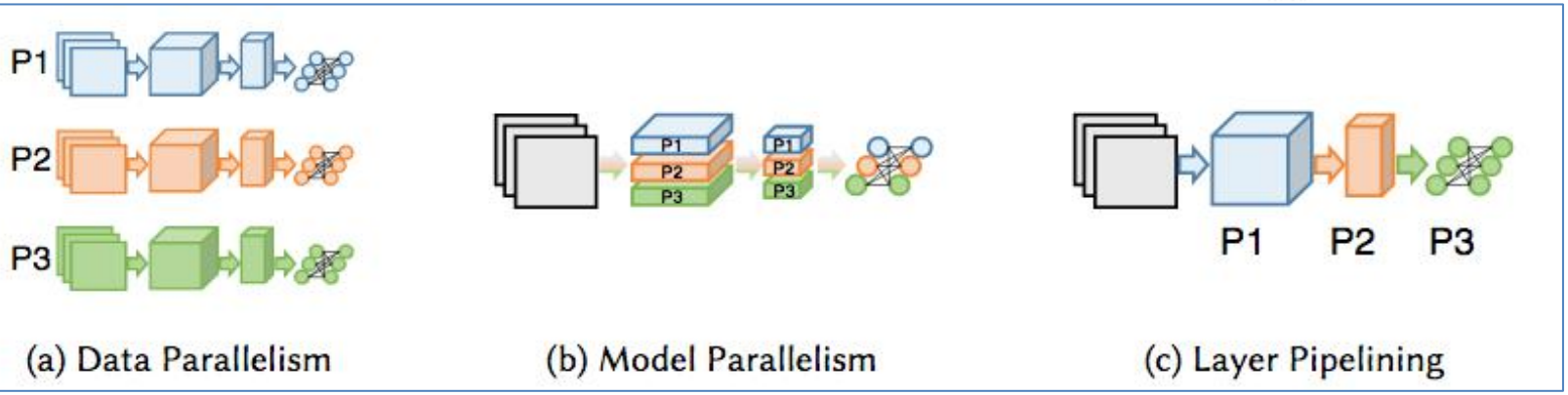
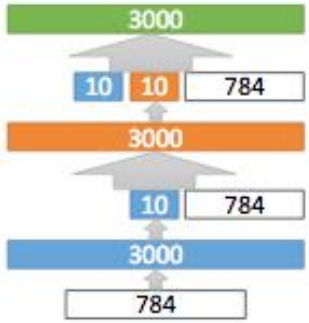


Table 1 : Training time and top-1 1-crop validation accuracy with ImageNet/ResNet-50

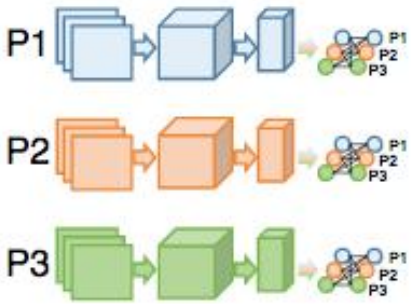
	Batch Size	Processor	DL Library	Time	Accuracy
2016	He et al.	256	Tesla P100 x8	Caffe	29 hours	75.3%
2017	Goyal et al.	8K	Tesla P100 x256	Caffe2	1 hour	76.3%
2017	Smith et al.	8K→16K	full TPU Pod	TensorFlow	30 mins	76.1%
2017	Akiba et al.	32K	Tesla P100 x1024	Chainer	15 mins	74.9%
2018	Jia et al.	64K	Tesla P40 x2048	TensorFlow	6.6 mins	75.8%
2018	Mikami et al.	**34K→68K**	**Tesla V100 x2176**	**NNL**	**224 secs**	**75.03%**

Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis - TAL BEN-NUN and TORSTEN HOEFLER,ETH Zurich, Switzerland (https://arxiv.org/pdf/1802.09941.pdf)

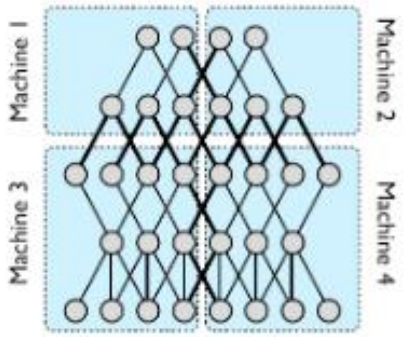
Large Scale DNN



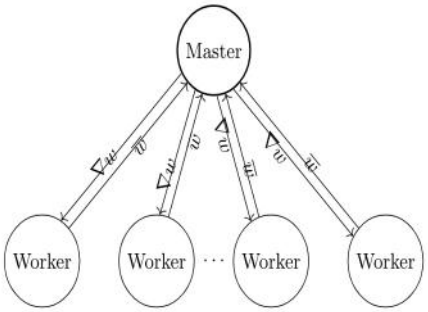
(a) Deep Stacking Network [62]

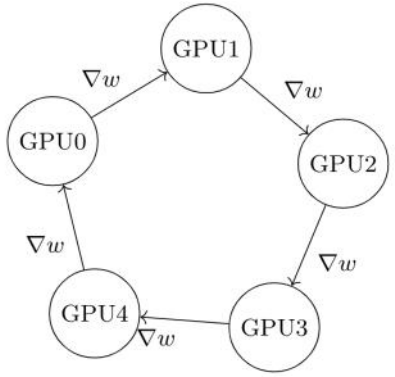


(b) Hybrid Parallelism



(c) DistBelief Replica [56]





PipeDream:https://cs.stanford.edu/~matei/papers/2019/sosp_pipedream.pdf

Mesh-TensorFlow: Deep Learning for Supercomputers

GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

Distributed Training and Inference of Deep Learning Models for Multi-Modal Land Cover Classification

- ✓ NCCL (Nvidia): collective multi-GPU communication
- ✓ Horovod (Uber): Tensorflow and Pytorch support
  - NCCLReduceScatter
  - MPIAllreduce
  - NCCLAllgather for data divisible by local_rank()
  - NCCLReduce-MPIAllreduce
  - NCCLBcastfor the remainder
  - Tensor Fusion: fuse small allreduce tensor operations into larger ones for performance gain
- ✓ GLOO (Facebook): Pytorch support
- ✓ DDL (IBM): Tensorflow, Pytorch, Caffe support. Close source.

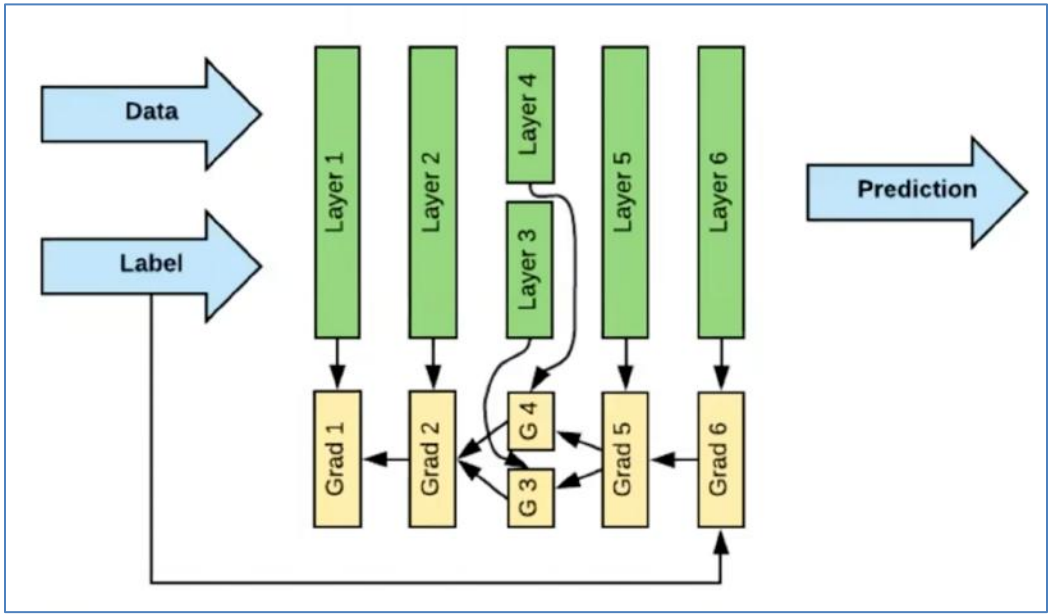
https://www.youtube.com/watch?v=fZ9P1v1jtFM (UBER)

Distributed Deep Learning with Horovod

Travis Addair, Machine Learning Platform, Uber Engineering
@tgaddair

Deep Learning @ Uber

- Self-Driving Vehicles
- Trip Forecasting
- Fraud Detection
- ... and many more!



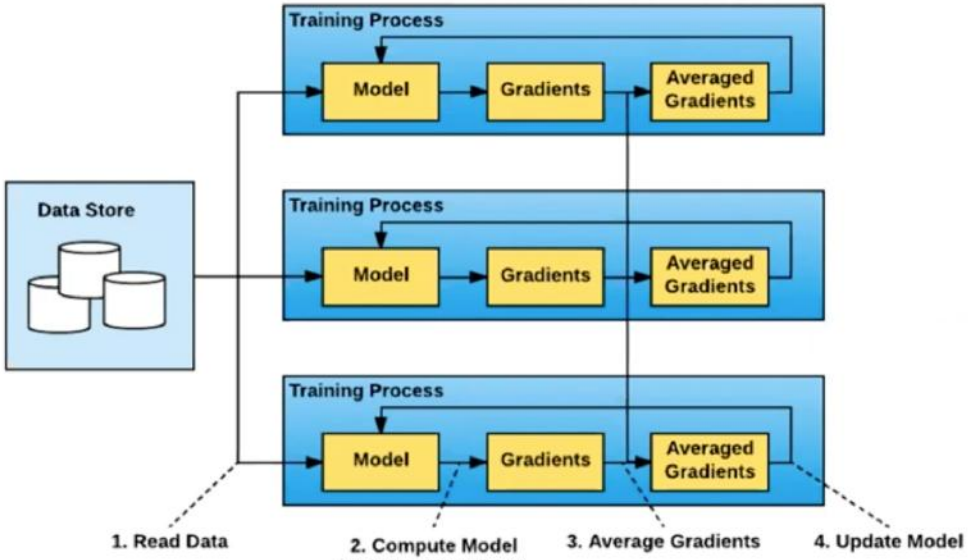
Massive amounts of data...

...make things slow (weeks!)

Solution:
distributed training.

How much GPU memory?
AWS p3x16large: 128GB
NVIDIA DGX-2: 512GB

Most models fit in a server.
➔ Use data-parallel training.



- Library for distributed deep learning.
- Works with stock TensorFlow, Keras, PyTorch, and Apache MXNet.
- Installs on top via `pip install horovod`.
- Uses advanced algorithms & can leverage features of high-performance networks (RDMA, GPUDirect).
- Separates infrastructure from ML engineers:
  - Infra team provides container & MPI environment
  - ML engineers use DL frameworks that they love
  - Both Infra team and ML engineers have consistent expectations for distributed training across frameworks

horovod.ai

Data Parallel Model

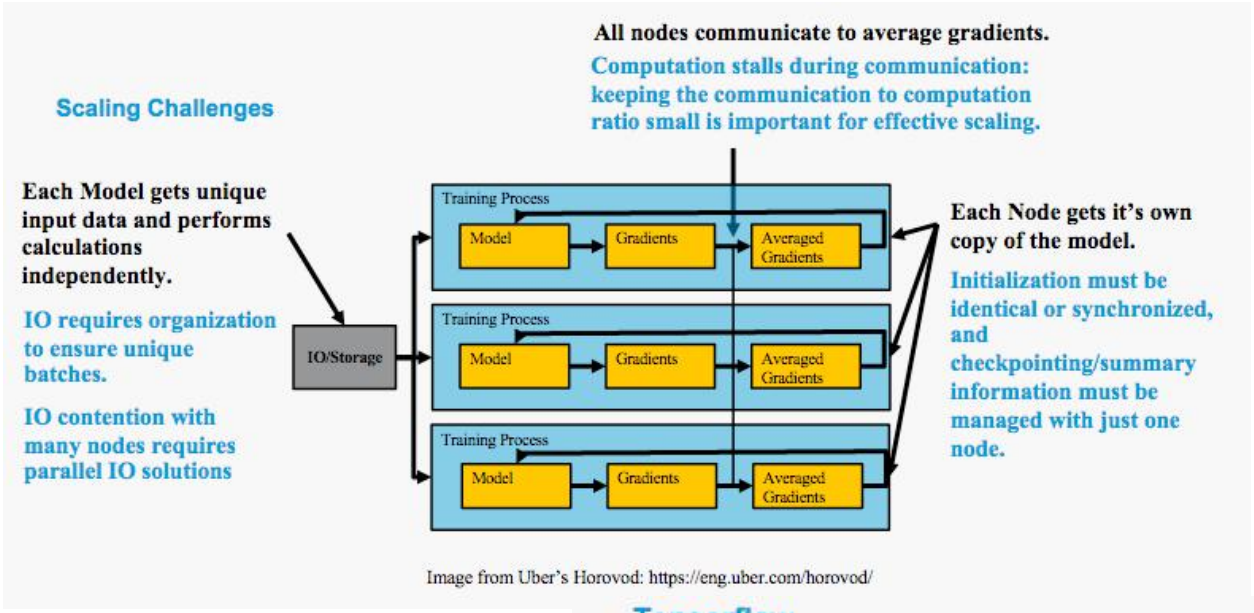
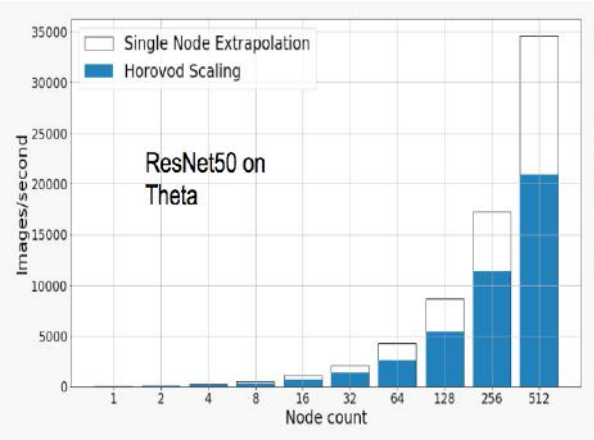


Image from Uber's Horovod: https://eng.uber.com/horovod/

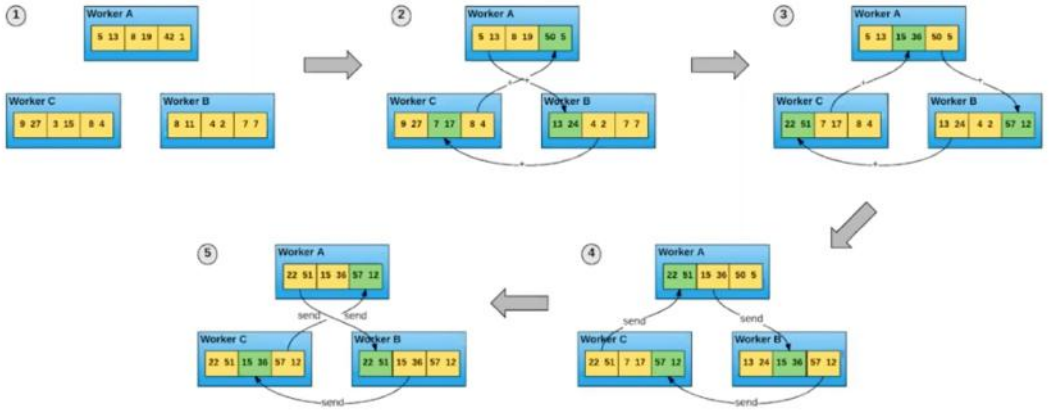
https://www.alcf.anl.gov/sites/default/files/2020-01/Tensorflow_ESP_0.pdf



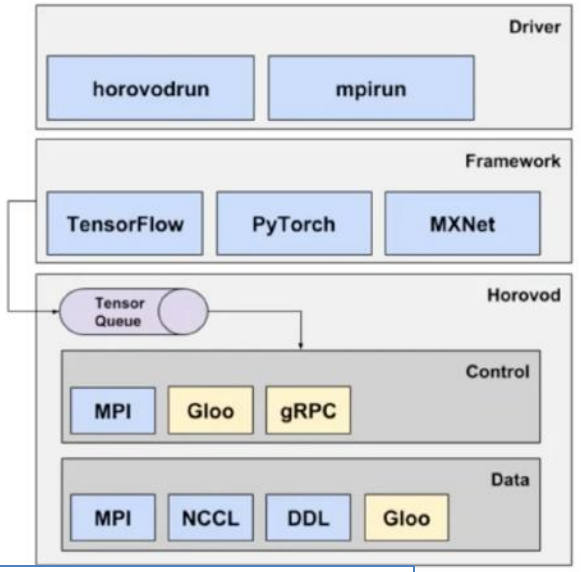
Tensorflow

```
import tensorflow as tf
import horovod.tensorflow as hvd
layers = tf.contrib.layers
learn = tf.contrib.learn
def main():
    # Horovod: initialize Horovod.
    hvd.init()
    # Download and load MNIST dataset.
    mnist = learn.datasets.mnist.read_data_sets('MNIST-data-%d' % hvd.rank())
    # Horovod: adjust learning rate based on number of GPUs.
    opt = tf.train.RMSPropOptimizer(0.001 * hvd.size())
    # Horovod: add Horovod Distributed Optimizer
    opt = hvd.DistributedOptimizer(opt)
    hooks = [
        hvd.BroadcastGlobalVariablesHook(0),
        tf.train.StopAtStepHook(last_step=20000 // hvd.size()),
        tf.train.LoggingTensorHook(tensors={'step': global_step, 'loss': loss},
                                   every_n_iter=10),
    ]
    checkpoint_dir = './checkpoints' if hvd.rank() == 0 else None
    with tf.train.MonitoredTrainingSession(checkpoint_dir=checkpoint_dir,
                                           hooks=hooks,
                                           config=config) as mon_sess
```

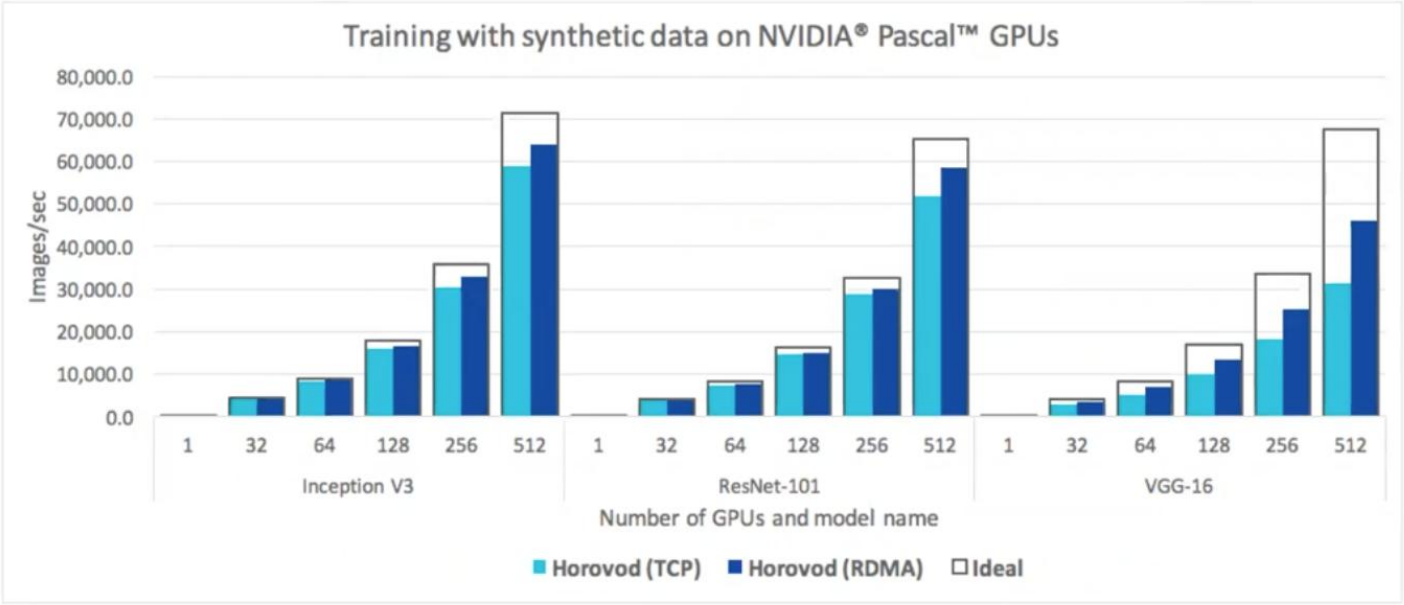
Horovod Technique: Ring-Allreduce



Patarasuk, P., & Yuan, X. (2009). Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2), 117-124. doi:10.1016/j.jpdc.2008.09.002



Horovod Performance

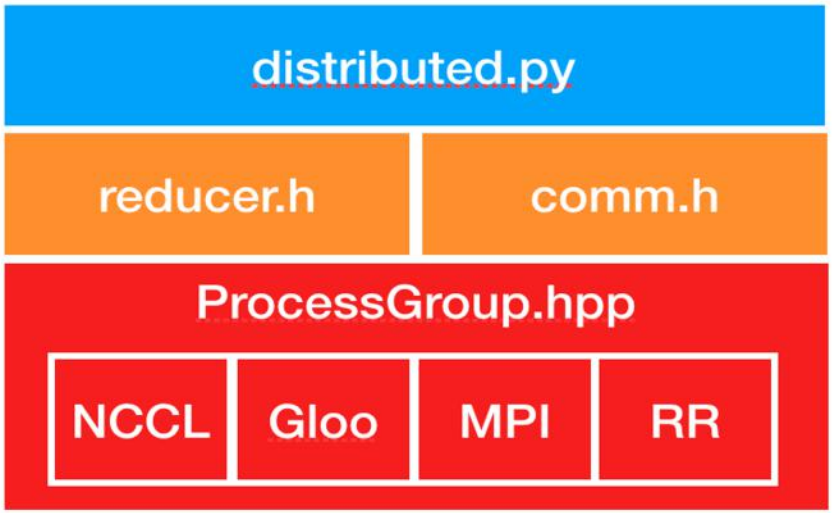


Uber

Horovod scales well beyond 128 GPUs. RDMA helps at a large scale, especially to small models with fully-connected layers like VGG-16, which are very hard to scale.

Distributed Data Parallel

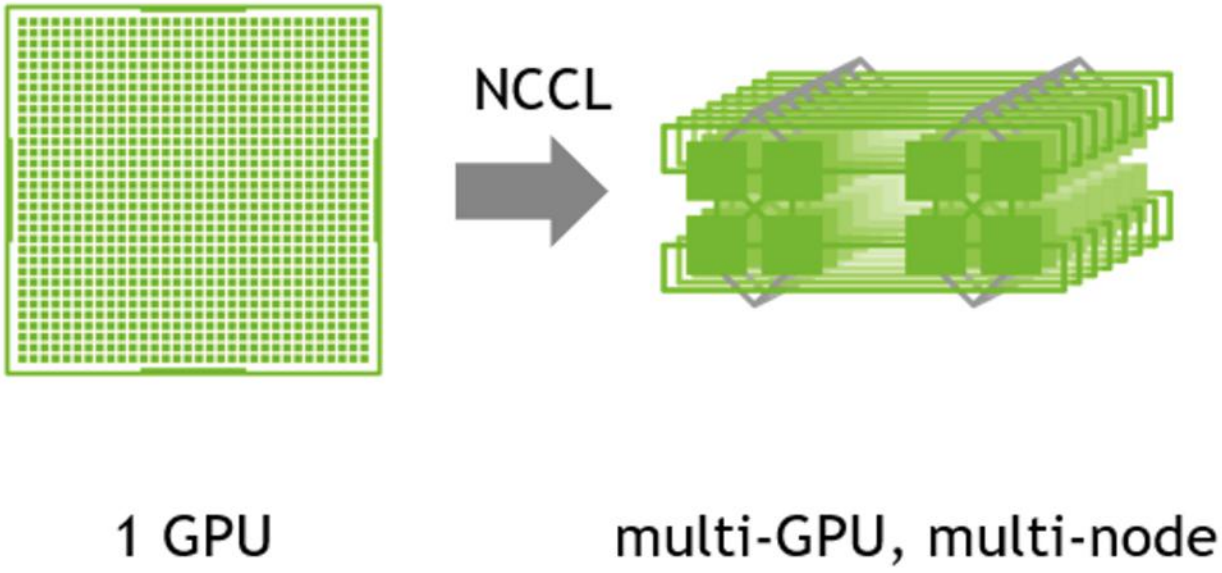
- distributed.py: is the Python entry point for DDP. It implements the initialization steps and the forward function for the nn.parallel.DistributedDataParallel module which call into C++ libraries. Its _sync_param function performs intra-process parameter synchronization when one DDP process works on multiple devices, and it also broadcasts model buffers from the process with rank 0 to all other processes. The inter-process parameter synchronization happens in Reducer.cpp.
- comm.h: implements the coalesced broadcast helper function which is invoked to broadcast model states during initialization and synchronize model buffers before the forward pass.
- reducer.h: provides the core implementation for gradient synchronization in the backward pass.



https://pytorch.org/docs/master/notes/ddp.html

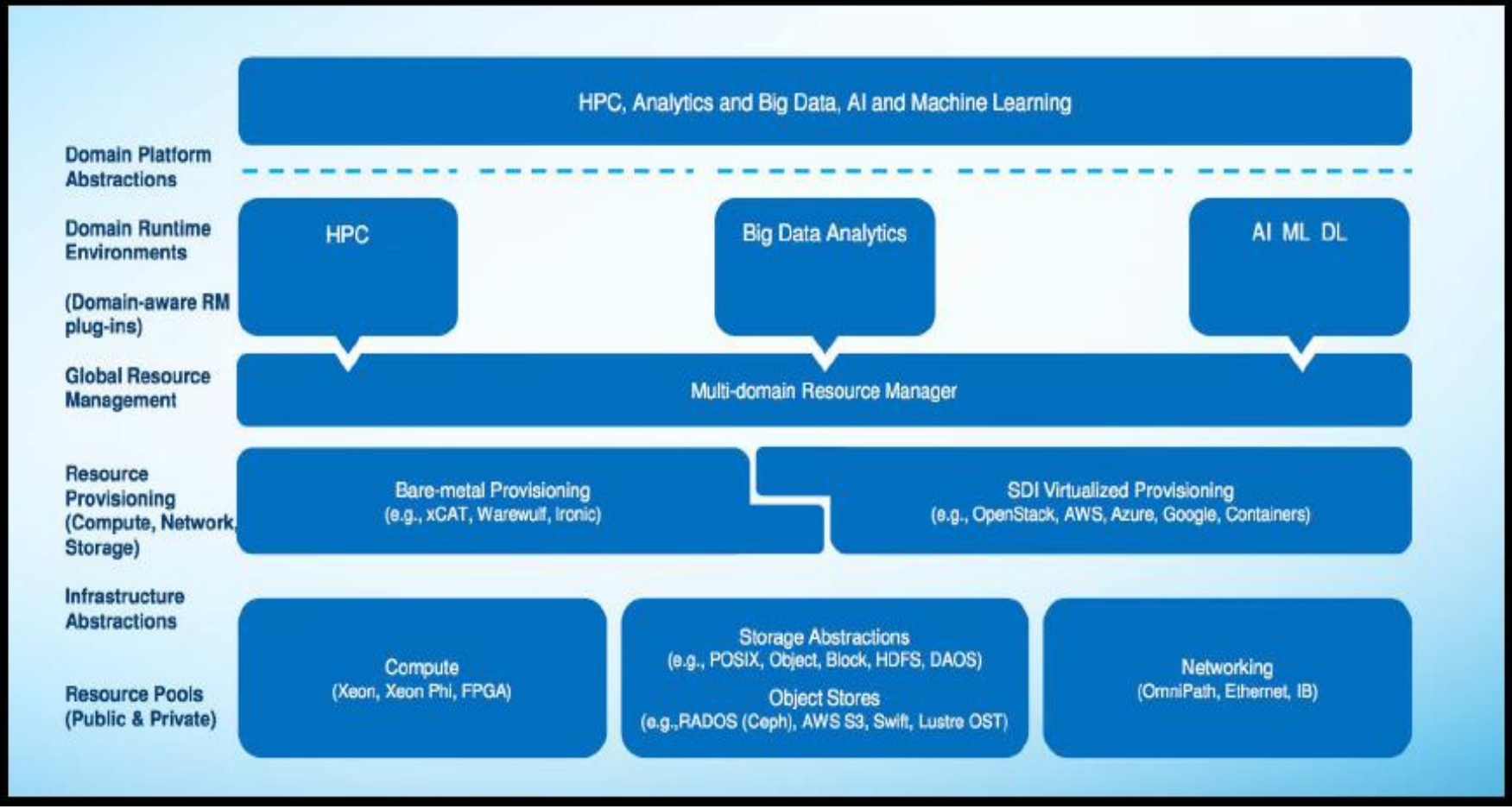
Distributed Data Parallel

- Another common backend is nccl. The NVIDIA Collective Communication Library (NCCL) implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and Networking. NCCL provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter as well as point-to-point send and receive that are optimized to achieve high bandwidth and low latency over PCIe and NVLink high-speed interconnects within a node and over NVIDIA Mellanox Network across nodes.



https://developer.nvidia.com/nccl

Towards an Integrated Sim, Data, Learn Stack



From Now to Beyond

From predict to build..
From modeling to generating
From control to autonomous
From composition to integration
From predict to build…

From Now to Beyond

https://www.nvidia.com/gtc/sessions/

NVIDIA

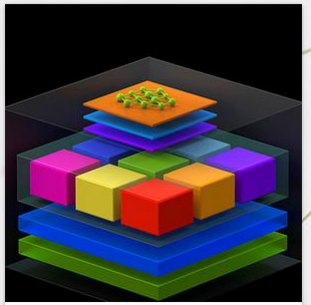
EN

GTC

CONFERENCE & TRAINING MARCH 21 - 24, 2022
KEYNOTE MARCH 22

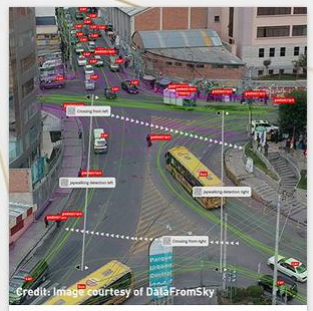
REGISTER | SIGN IN

Keynote

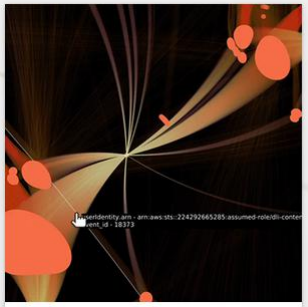


Accelerated Computing and Dev Tools

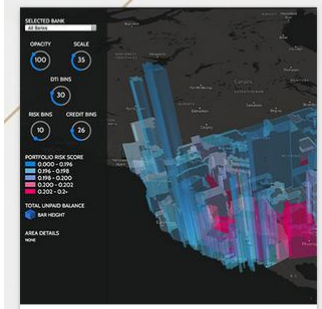
Autonomous Vehicles



Computer Vision/ Video Analytics



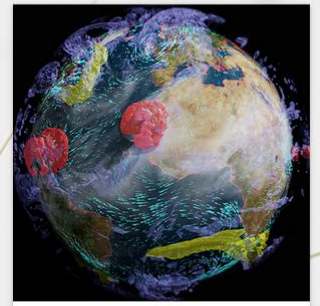
Cybersecurity



Data Science

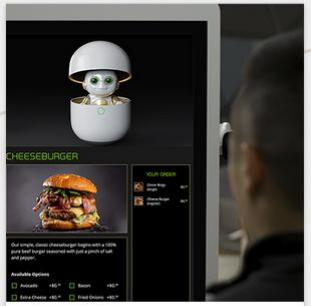
Game Development

Graphics, Design Collaboration, and Digital Twins



HPC

Inference



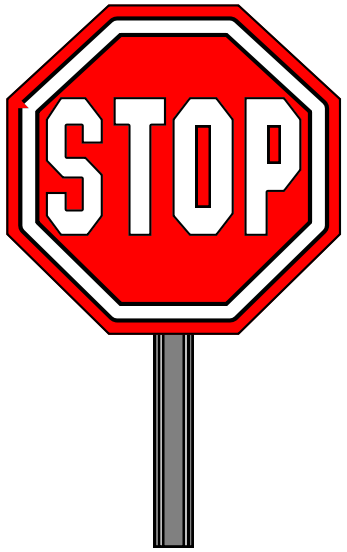
Recommender System

Robotics

Speech AI / NLP

Keynote Movie
www.nvidia.com/gtc/keynote

The End



- The End!