PEARC20 Tutorial

How to Build Your Own Deep Neural Network

Kwai Wong, Stan Tomov
Daniel Nichols, Rocco Febbo, Florent Lopez
Julian Halloy, Xianfeng Ma

University of Tennessee, Knoxville

PEARC20, July 31, 2020

Acknowledgements:

- Support from NSF, UTK, JICS, NICS
- MagmaDNN is a project grown out from the RECSEM REU Summer program supported under NSF award #1659502
- Efforts and reference materials from many colleagues, collaborators, researchers, and students
- www.olcf.ornl.gov, www.icl.utk.edu, cfdlab.utk.edu, www.xsede.org, www.jics.utk.edu/recsem-reu,
- Source code: www.bitbucket.org/icl/magmadnn
- LAPENNA, www.jics.utk.edu/lapenna

NSF

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

JICS Joint Institute for Computational Sciences

OAK RIDGE National Laboratory

How to Build Your Own Neural Network Framework

The tutorial aims to show researchers how to write a deep neural network (DNN) software program. In this tutorial, we will show how typical DNN algorithms are implemented, illustrated with programming structure of MagmaDNN and example exercises.

- ✓ **This tutorial is intended for researchers that have previously used an open source DNN framework but would like to learn more about its programming structure**
- ✓ **The primary computing platform is a Linux computer with a Nvidia GPU card. We will not use the Jetson Nano cards for demonstrations. Illustration of computing on multiple GPUs will be done on a DGX box.**
- ✓ **This tutorial has 4 major sections, each one will last for about 45 minutes and a 10 minutes Q&A session at the end of each section.**
- ✓ **Please fill out the evaluation form**

- ✓ **Overview of Deep Neural Network : MLP, CNN, CUDNN, MKL DNN, TensorFlow, MagmaDNN, MNIST example**
- ✓ **GPU computing : Linear Algebra, MKL, CUDA, GPU programming, Magma, mixed precision, exercises**
- ✓ **Programming structure of a DNN framework : I/O, Memory Management, Tensor, layers, activation and fit functions, CFIR 100, MagmaDNN**
- ✓ **Multiple GPUs and in-depth example exercises : data and model parallelism, distributed Computing, VGG, ResNet ImageNet.**

Acknowledgements and References

This portion of the tutorial contains many extracted materials from many online websites and courses. Listed below are the major sites. This portion of the materials is not intended for public distribution. Please visit the sites websites for detail contents. If you are beginner users of DNN, I suggest to read the following list of websites in its order.

1) http://neuralnetworksanddeeplearning.com
2) https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist
3) https://www.deeplearning.ai/deep-learning-specialization/
4) http://cs231n.stanford.edu/
5) More sites are listed in www.jics.utk.edu/actia
6) Others are listed on the slides

NSF

INNOVATIVE
COMPUTING LABORATORY

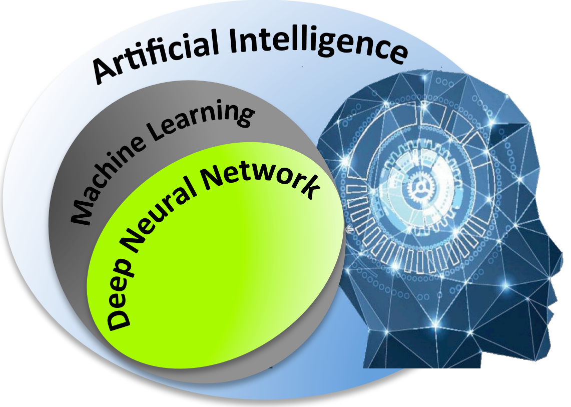
THE UNIVERSITY OF
TENNESSEE
KNOXVILLE

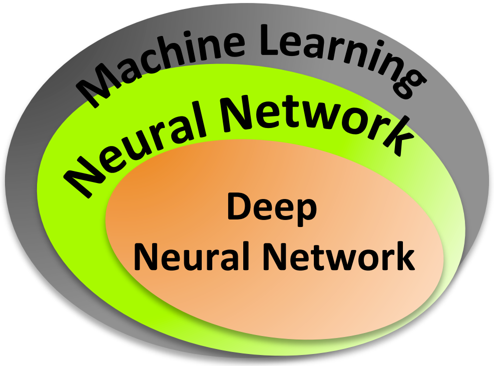
JICS
Joint Institute for
Computational Sciences
UT ORNL
Computational Sciences

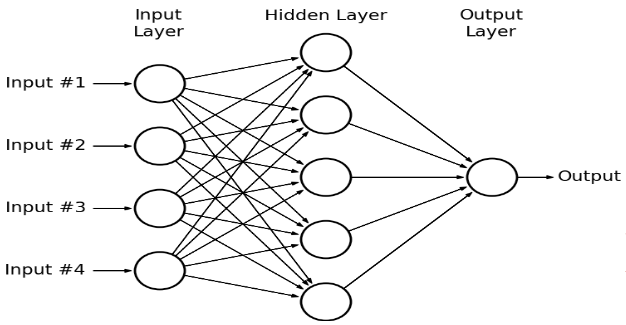
Overview of Deep Neural Network

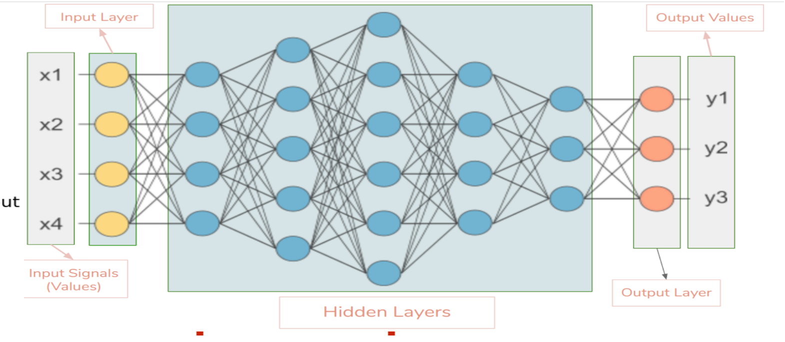
- ✓ **Introduction to machine learning and Deep Neural Network**
- ✓ **Activation function, optimization, cost function, SDG**
- ✓ **Regression and Classification**
- ✓ **Regression, MLP, forward and backward process**
- ✓ **Computational graph, example of backpropagation**
- ✓ **Classification, CNN,**
- ✓ **Arithmetic of CNN**
- ✓ **Number of Parameters**
- ✓ **AlexNet, VGG16, ResNet**
- ✓ **More about optimizers**
- ✓ **TensorFlow, MagmaDNN, MNIST example**

AI, ML, NN, DNN



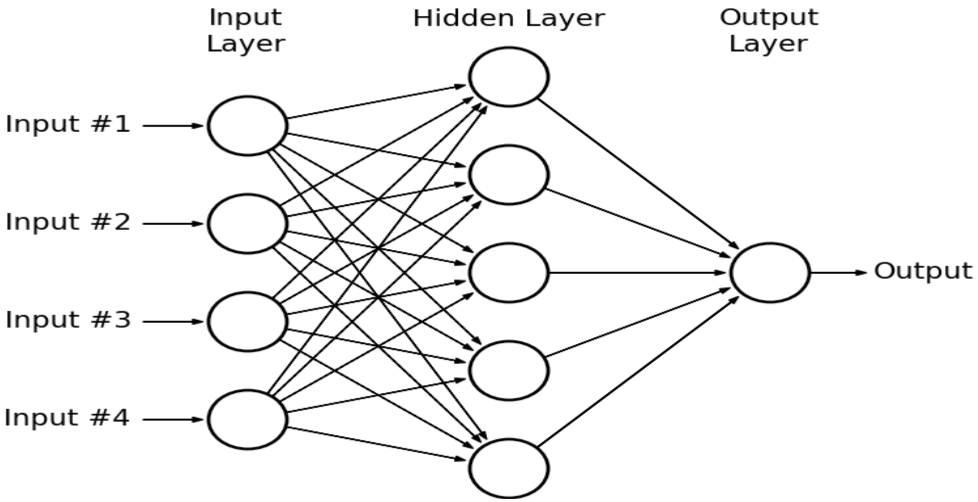


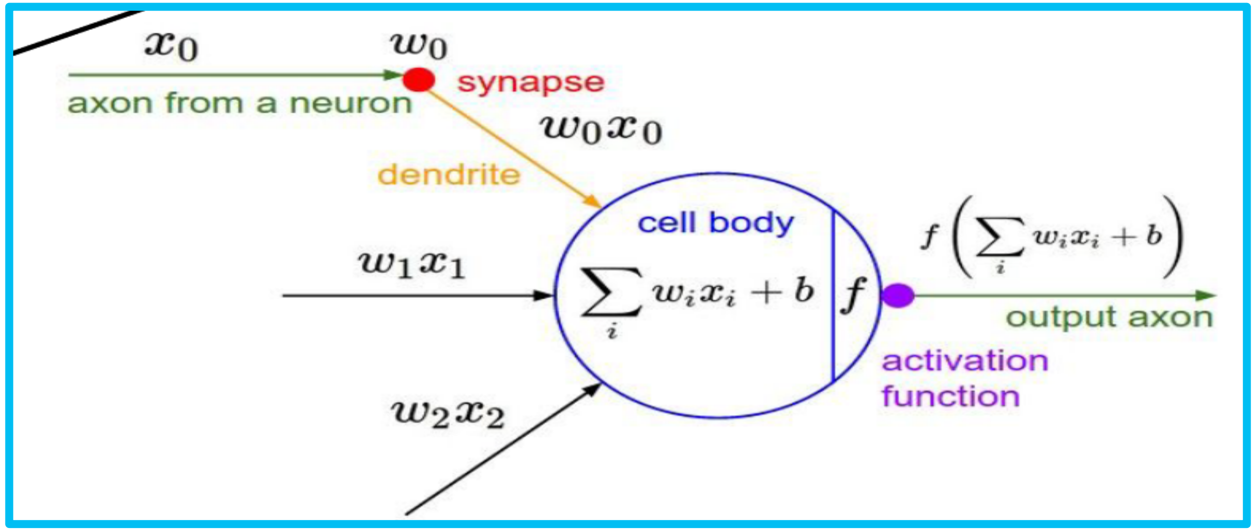




- ✓ **Artificial Intelligence (AI)**: science and engineering of making intelligent machines to perform the human tasks (John McCarthy,1956). AI applications is ubiquitous.
- ✓ **Machine learning (ML) :** A field of study that gives computers the ability to learn without being explicitly programmed (Arthur Samuel, 1959). A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E (Tom Mitchell,1998).
- ✓ **Neural Network (NN) :** Neural Network modeling, a subfield of **ML** is algorithm inspired by structure and functions of biological neural nets
- ✓ **Deep Neural Network (DNN) :** (aka deep learning): an extension of **NN** composed of many layers of functional neurons, is dominating the science of modern **AI** applications
- ✓ **Supervised Learning (SL)** : A class in ML, dataset has labeled values, use to predict output values associated with new input values.

NN Modeling

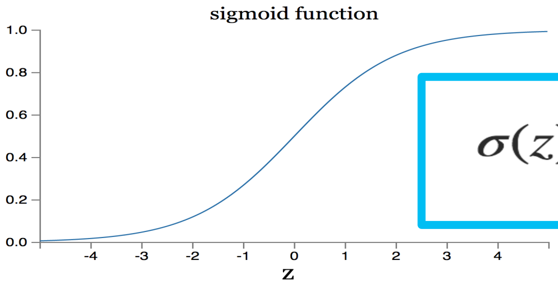




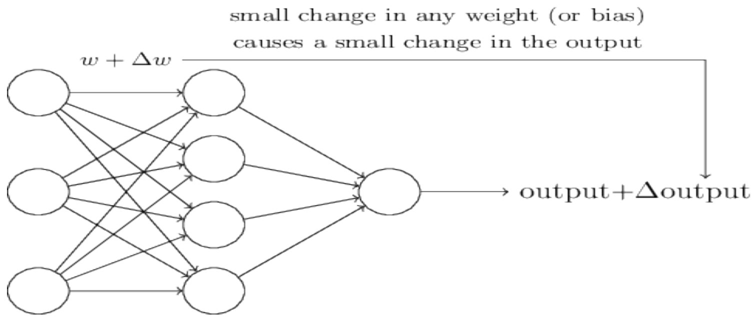
- ✓ A node in the neural network is a mathematical function or activation function which maps input to output values.
- ✓ Inputs represent a set of vector containing weights (w) and bias (b). They are the sets of parameters to be determined for prediction, regression.
- ✓ Many nodes form a **neural layer**, **links** connect layers together, defining a NN model.
- ✓ Activation function (f or σ), is generally a nonlinear data operator which faciltates identification of complex features.

Activation Function

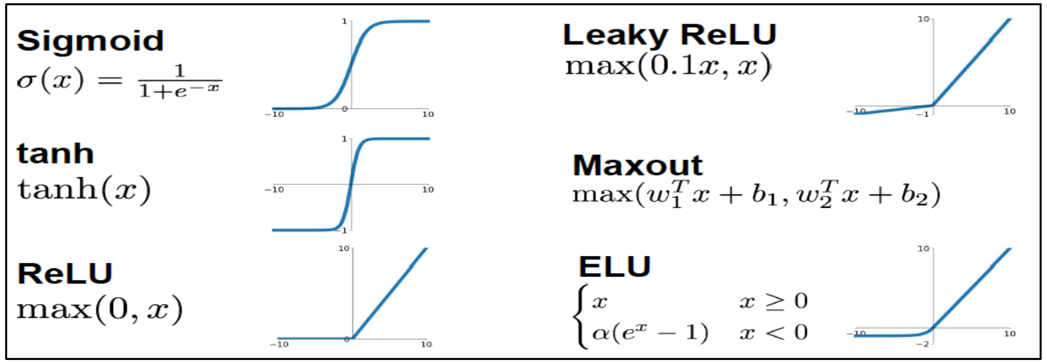
neuralnetworksanddeeplearning.com



$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$


small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

$$\Delta \text{output} \approx \sum_j \frac{\partial\, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial\, \text{output}}{\partial b} \Delta b,$$


Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh
$\tanh(x)$

ReLU
$\max(0, x)$

Leaky ReLU
$\max(0.1x, x)$

Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

- ✓ Sigmond (logistic) function, *σ*, is a smooth out preceptron operating on a set of data, *σ*(*w*·*x*+*b*). The smoothness of *σ* means that small changes Δ*w* in the weights and Δ*b* in the bias will produce a small change Δoutput in the output from the neuron.
- ✓ If *z*≡*w*·*x*+*b* is a large positive number, then *exp(−z)*≈0 and so *σ*(*z*)≈1.
- ✓ If *z*=*w*·*x*+*b* is very negative, then *exp(−z)*→∞, and *σ*(*z*)≈0. So when *z*=*w*·*x*+*b* is very negative.
- ✓ It's only when *w*·*x*+*b* is of modest size that there's much deviation from the perceptron model.
- ✓ Δoutput is a *linear function* of the changes Δ*wj* and Δ*b* in the weights and bias. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output.

Minimization, Training

neuralnetworksanddeeplearning.com

$$\boxed{C(w,b) \equiv \frac{1}{2n} \sum_x \| y(x) - a \|^2.}$$

Define a ***cost function***, ***C* is the *quadratic* cost function** also referred as the *mean squared error(MSE) .*

Define a **cost function C(w,b), the mean squared error (MSE)** , so the neural network training algorithm finds the weights and biases giving *C*(*w*,*b*)≈0, or **minimizes** the cost *C*(*w*,*b*) as a function of the weights and biases, casting it as an optimization problem using the ***gradient descent algorithm***.

- ✓ Need to find a way of choosing Δ*v*1 and Δ*v*2 so as to make Δ*C (Δoutput)* negative (move to minimum).
- ✓ To make gradient descent work correctly, choose the learning rate *η* to be small but not too small such that the gradient descent algorithm stops works properly.

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T.$$

Be negative

$$\Delta \text{output} \approx \sum_j \frac{\partial\, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial\, \text{output}}{\partial b} \Delta b,$$

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2.$$

$$\boxed{\Delta C \approx \nabla C \cdot \Delta v.}$$

Choose

$$\boxed{\Delta v = -\eta \nabla C,}$$

$$\boxed{\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \| \nabla C \|^2.}$$

$$\boxed{v \rightarrow v' = v - \eta \nabla C.}$$

Stochastic Gradient Descent

✓ Neural network is an optimization process to find a set of parameters that ensure the prediction function f(x, W) to be as close as possible to the true solution y for any input x

✓ Use gradient descent to find the weights *w* and biases *b* which minimize the cost function.

✓ To compute the gradient ∇C we need to compute the gradients ∇Cx separately for each training input, *x*, and then average them, $\nabla C=1/n\sum\nabla Cx$. Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly.

✓ A way is to use ***stochastic gradient descent*** to speed up learning. The idea is to estimate the gradient ∇C by computing a small sample of randomly chosen training inputs, refer to as a ***mini-batch*** *of input*.

✓ By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning, provided the sample size *m* is large enough that the average value of the ∇CXj roughly equals to the average over all ∇Cx.

✓ Another randomly chosen mini-batch are selected and trained, until all the exhausted the training inputs are used. It is said to complete an *epoch* (iteration) of training, then more iteration.

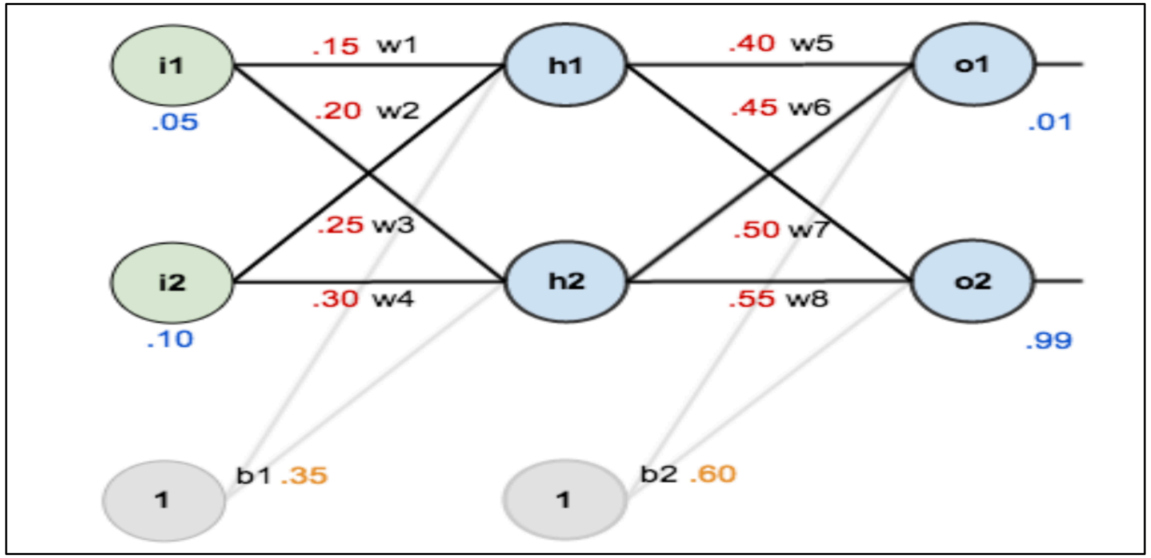
$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

neuralnetworksanddeeplearning.com

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Forward Path Exercise



$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

$$out_{h2} = 0.596884378$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

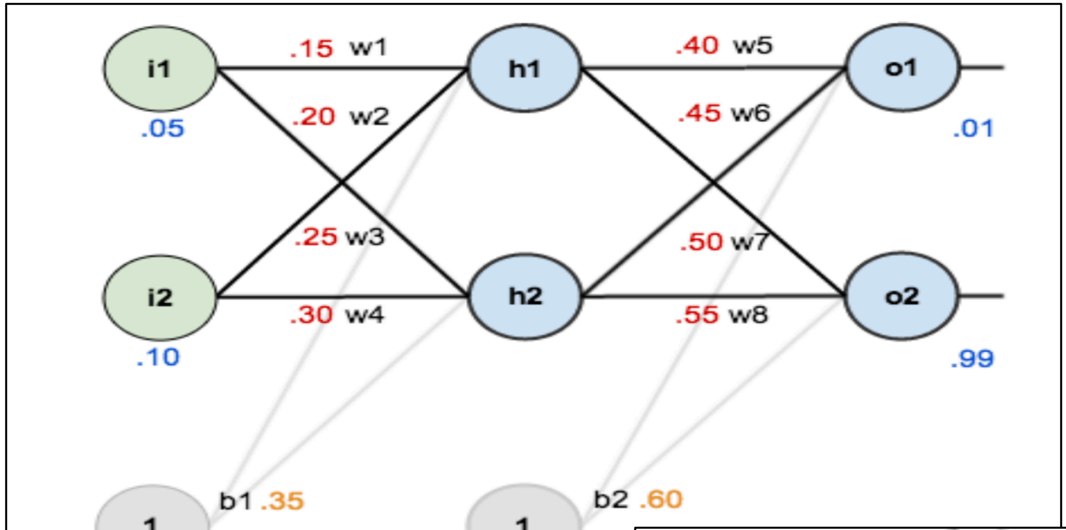
$$out_{o2} = 0.772928465$$

$$E_{o2} = 0.023560026$$

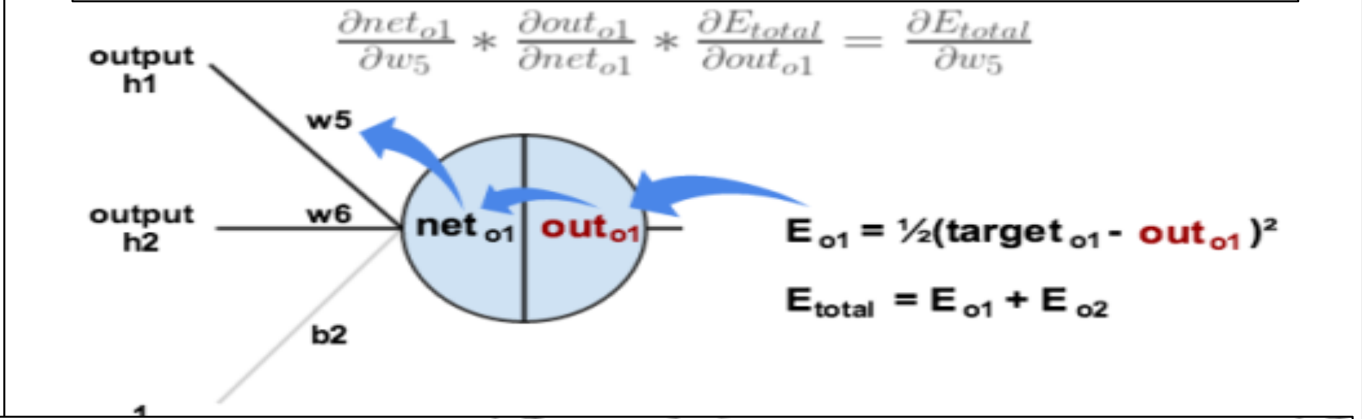
$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Backpropagation Example



$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$



$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$

$E_{o1} = ½(\text{target}_{o1} - \text{out}_{o1})^2$

$E_{total} = E_{o1} + E_{o2}$

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

$$\frac{\partial E_{total}}{\partial w_5} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

$$\delta_{o1} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = \frac{\partial E_{total}}{\partial net_{o1}}$$

$$\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})$$

$$\frac{\partial E_{total}}{\partial w_5} = \delta_{o1} out_{h1}$$

$$\frac{\partial E_{total}}{\partial w_5} = -\delta_{o1} out_{h1}$$

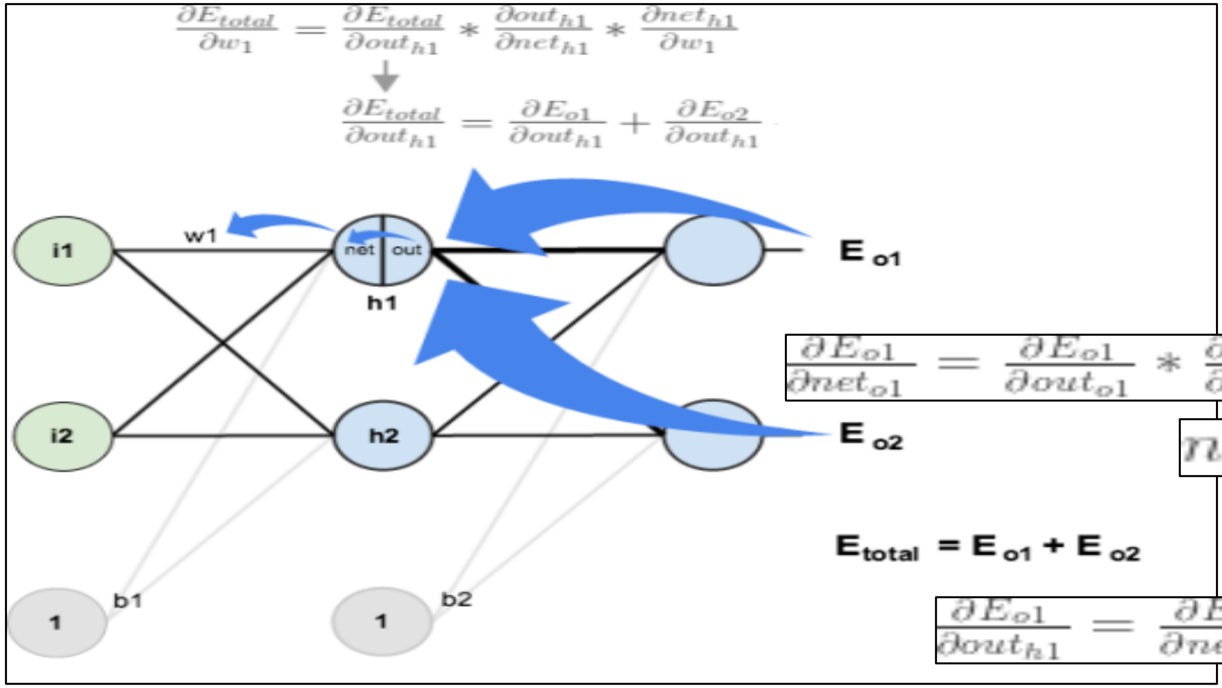
$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

Backpropagation Example



$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$E_{total} = E_{o1} + E_{o2}$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

$$\frac{\partial E_{total}}{\partial w_1} = \left(\sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}}\right) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = \left(\sum_o \delta_o * w_{ho}\right) * out_{h1}(1 - out_{h1}) * i_1$$

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

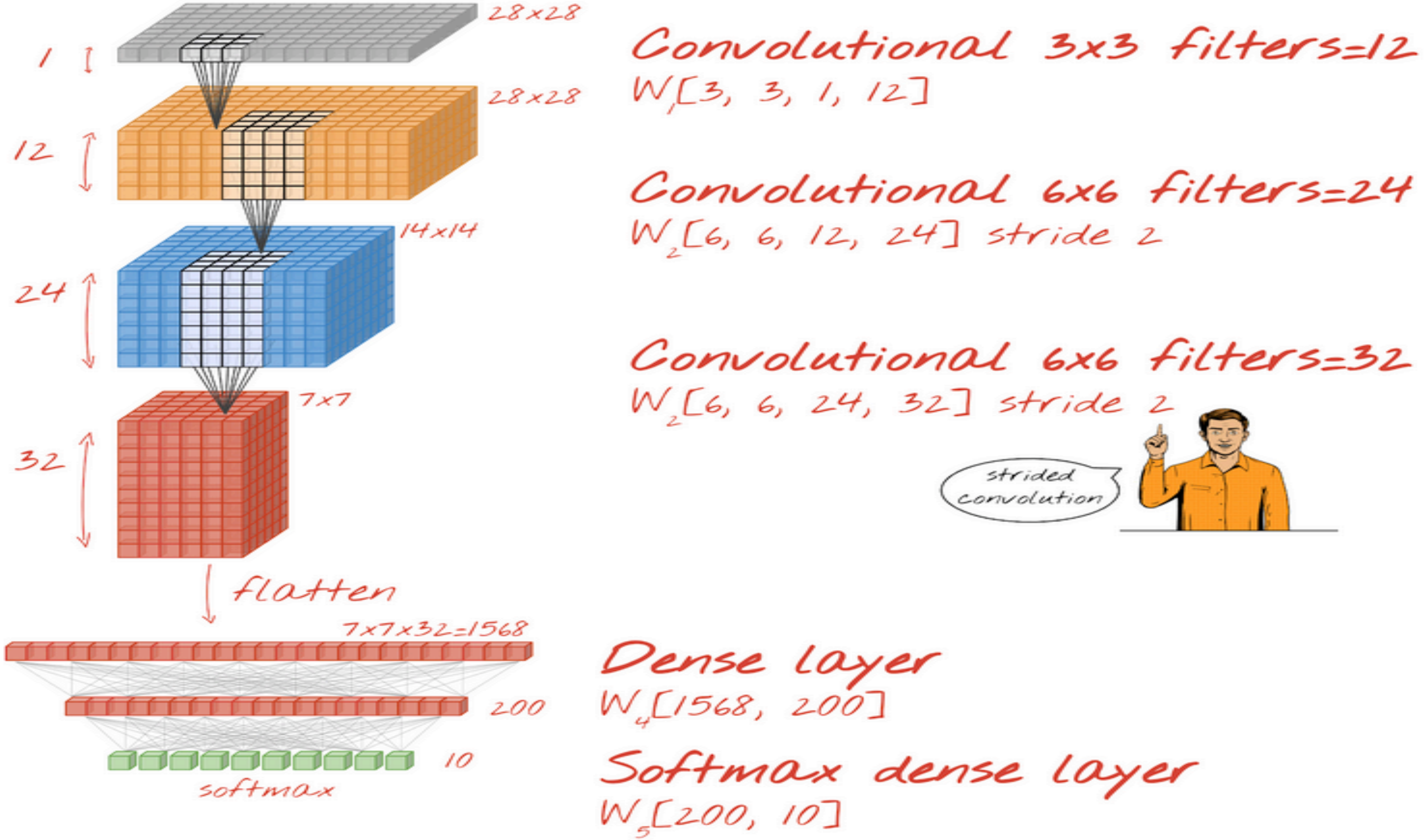
$$\frac{\partial E_{total}}{\partial w_1} = \delta_{h1} i_1$$

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

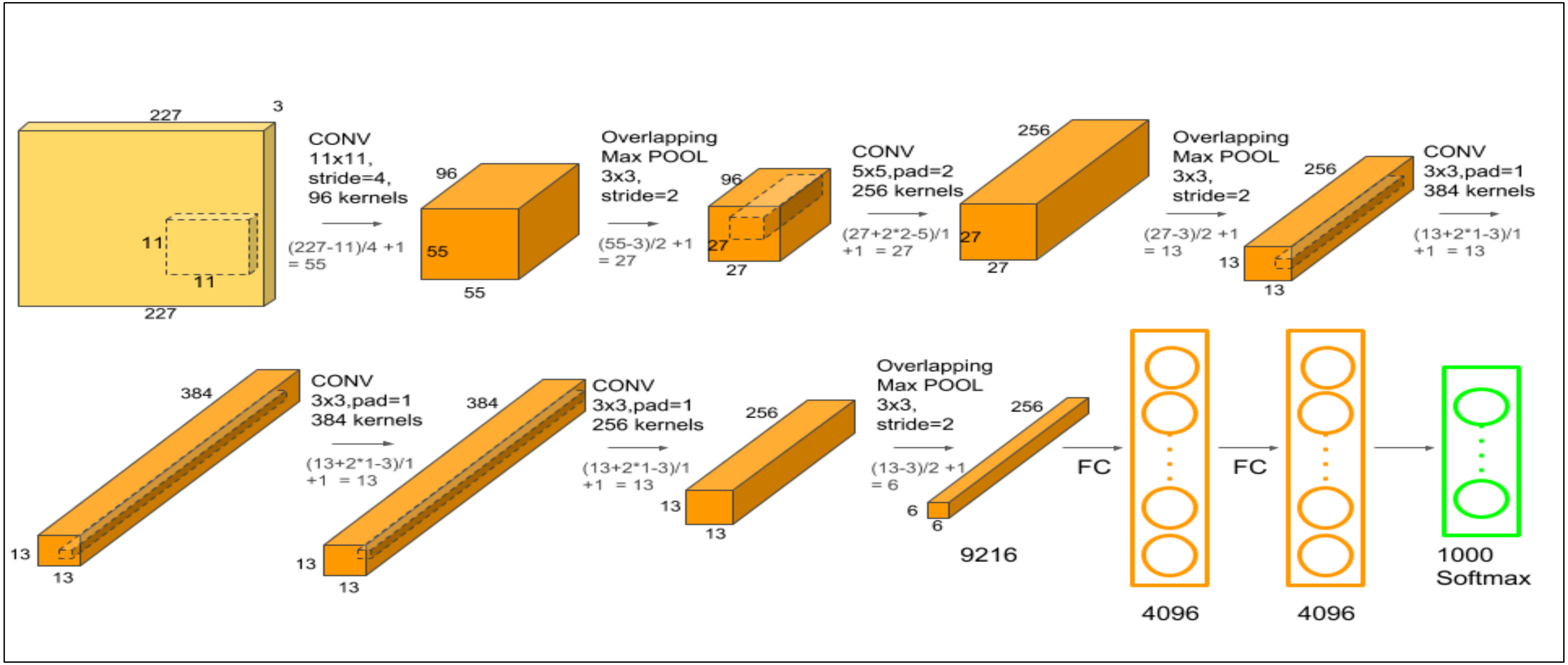
$$w_4^+ = 0.29950229$$

Convolution NN



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

Number of Parameters
AlexNet



https://www.learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network/

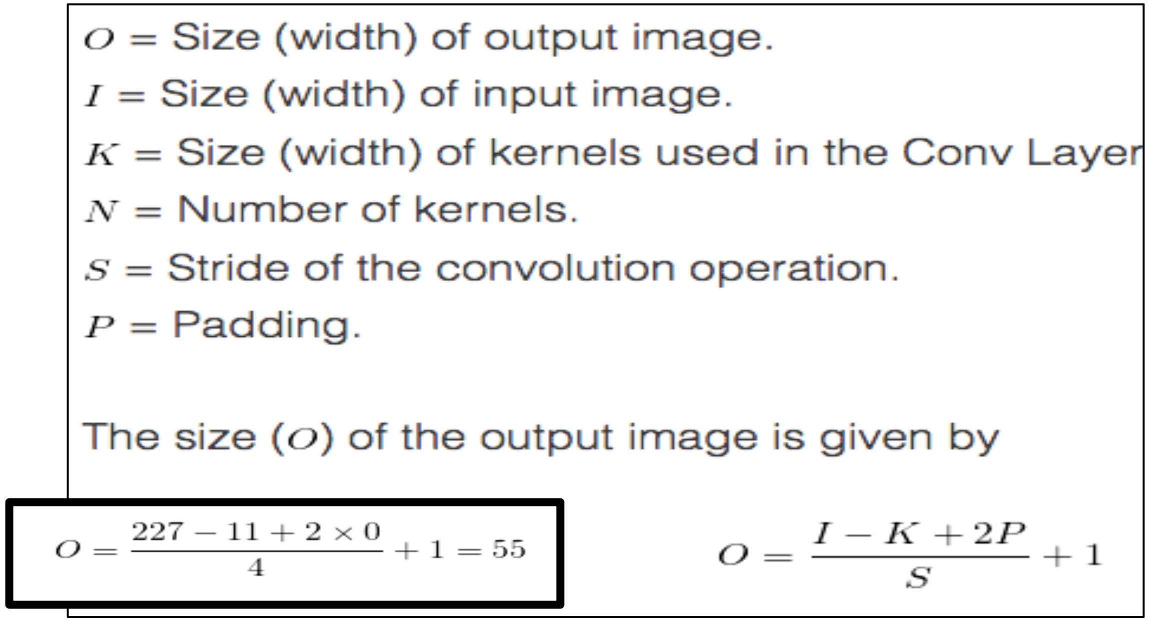
Number of Parameters

- ✓ **Input**: Color images of size 227x227x3. The AlexNet paper mentions the input size of 224×224 but that is a typo in the paper.
- ✓ **Conv-1**: The first convolutional layer consists of 96 kernels of size 11×11 applied with a stride of 4 and padding of 0.
- ✓ **MaxPool-1**: The maxpool layer following Conv-1 consists of pooling size of 3×3 and stride 2.
- ✓ **Conv-2**: The second conv layer consists of 256 kernels of size 5×5 applied with a stride of 1 and padding of 2.
- ✓ **MaxPool-2**: The maxpool layer following Conv-2 consists of pooling size of 3×3 and a stride of 2.
- ✓ **Conv-3**: The third conv layer consists of 384 kernels of size 3×3 applied with a stride of 1 and padding of 1.
- ✓ **Conv-4**: The fourth conv layer has the same structure as the third conv layer. It consists of 384 kernels of size 3×3 applied with a stride of 1 and padding of 1.
- ✓ **Conv-5**: The fifth conv layer consists of 256 kernels of size 3×3 applied with a stride of 1 and padding of 1.
- ✓ **MaxPool-3**: The maxpool layer following Conv-5 consists of pooling size of 3×3 and a stride of 2.
- ✓ **FC-1**: The first fully connected layer has 4096 neurons.
- ✓ **FC-2**: The second fully connected layer has 4096 neurons.
- ✓ **FC-3**: The third fully connected layer has 1000 neurons

https://www.learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network/

Number of Parameters

Size of the Output Tensor (Image) of a Conv Layer



O = Size (width) of output image.

I = Size (width) of input image.

K = Size (width) of kernels used in the Conv Layer

N = Number of kernels.

S = Stride of the convolution operation.

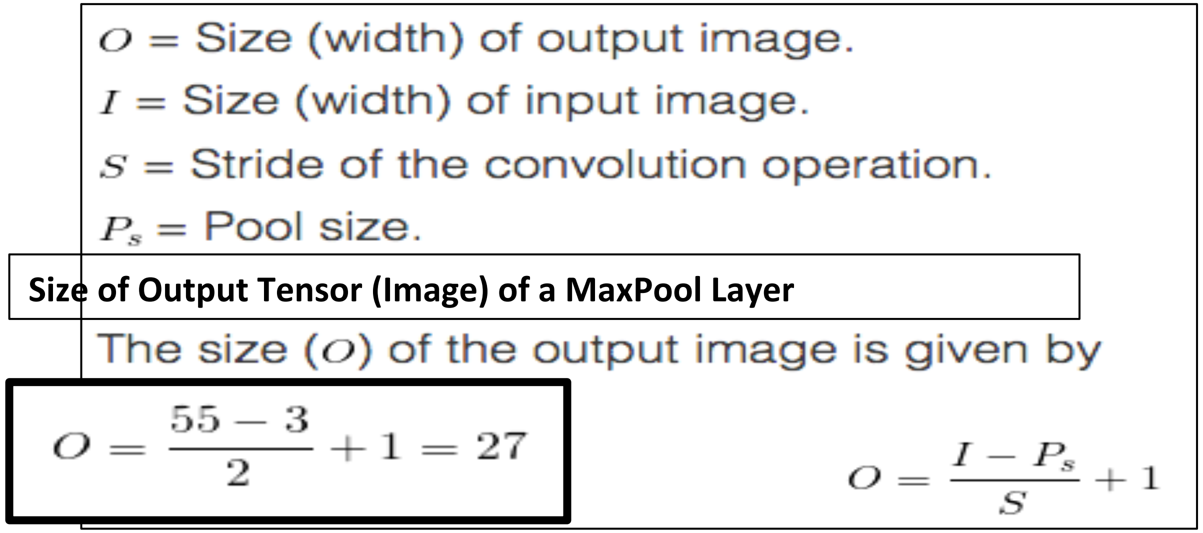
P = Padding.

The size (O) of the output image is given by

$$O = \frac{227 - 11 + 2 \times 0}{4} + 1 = 55$$

$$O = \frac{I - K + 2P}{S} + 1$$

Size of Output Tensor (Image) of a MaxPool Layer



O = Size (width) of output image.

I = Size (width) of input image.

S = Stride of the convolution operation.

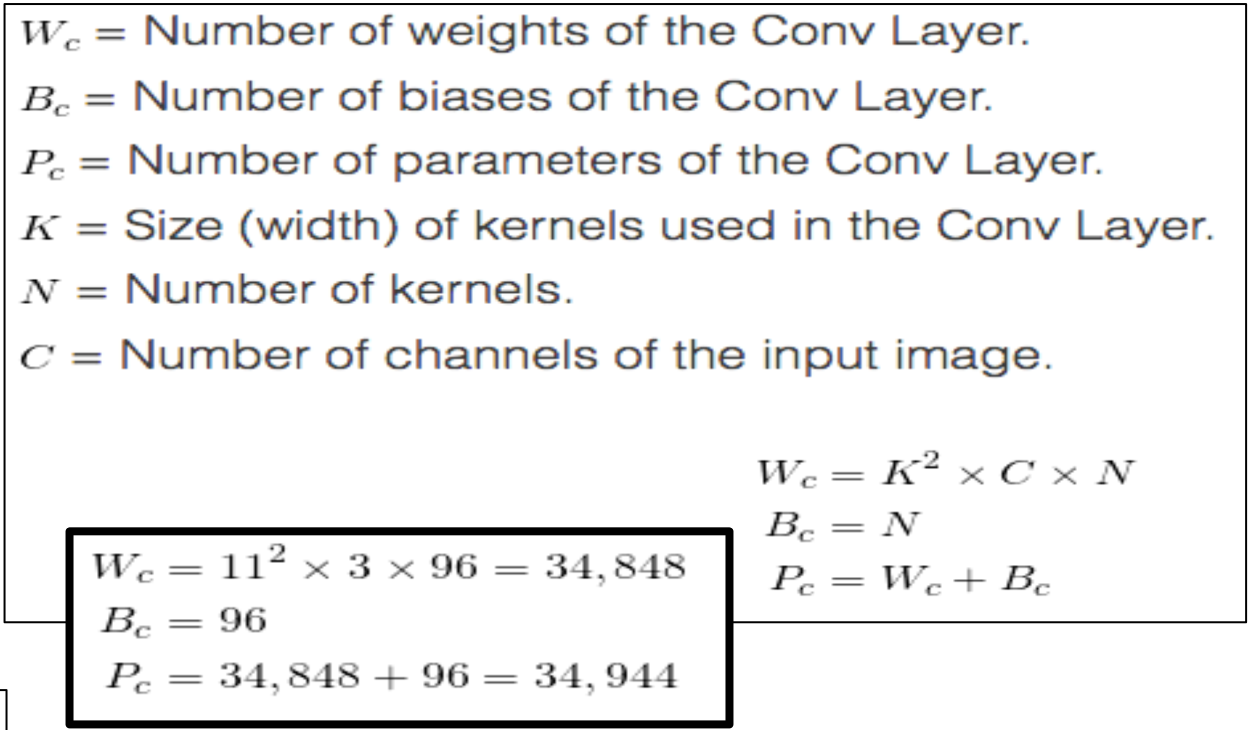
P_s = Pool size.

The size (O) of the output image is given by

$$O = \frac{55 - 3}{2} + 1 = 27$$

$$O = \frac{I - P_s}{S} + 1$$

Number of Parameters of a Conv Layer



W_c = Number of weights of the Conv Layer.

B_c = Number of biases of the Conv Layer.

P_c = Number of parameters of the Conv Layer.

K = Size (width) of kernels used in the Conv Layer.

N = Number of kernels.

C = Number of channels of the input image.

$$W_c = K^2 \times C \times N$$
$$B_c = N$$
$$P_c = W_c + B_c$$

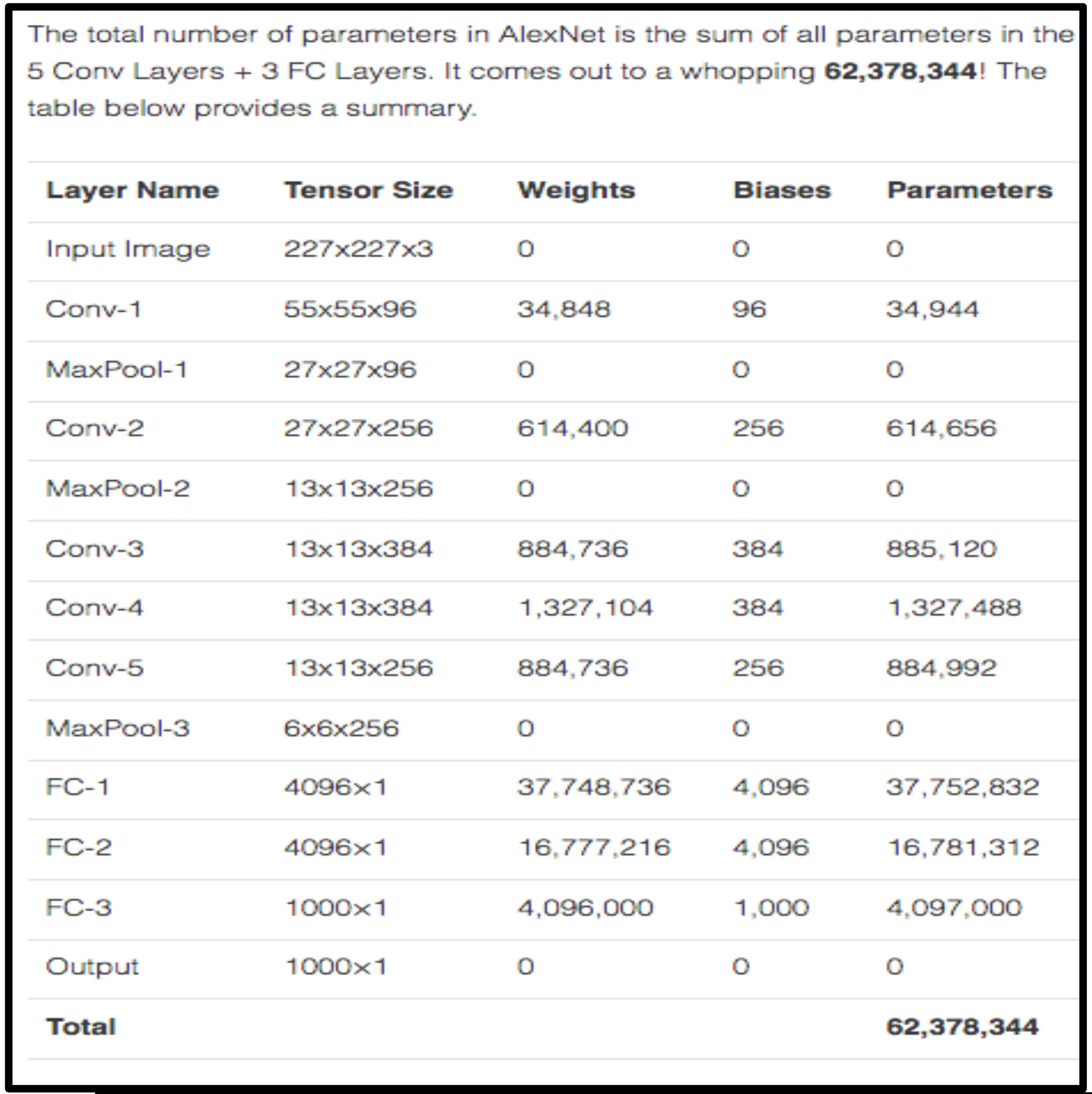
$$W_c = 11^2 \times 3 \times 96 = 34,848$$
$$B_c = 96$$
$$P_c = 34,848 + 96 = 34,944$$

Number of Parameters of a MaxPool Layer = 0

https://www.learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network/

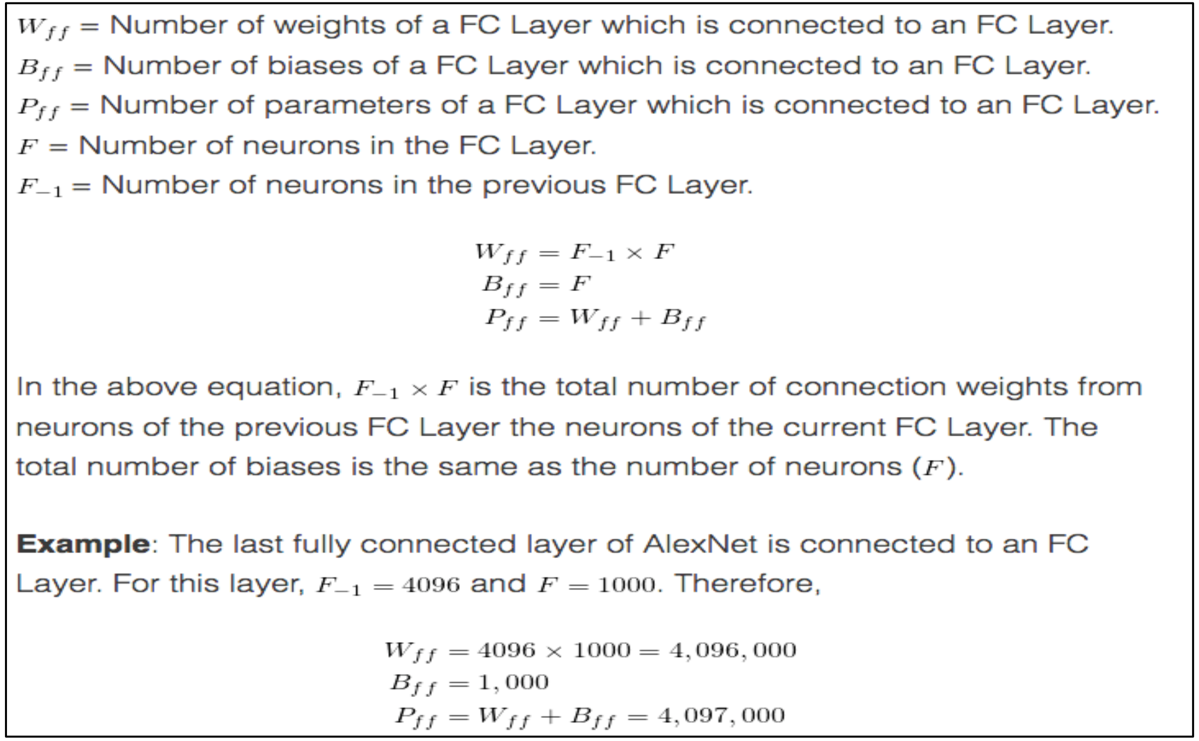
https://www.learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network/

Number of Parameters



The total number of parameters in AlexNet is the sum of all parameters in the 5 Conv Layers + 3 FC Layers. It comes out to a whopping **62,378,344**! The table below provides a summary.

Layer Name	Tensor Size	Weights	Biases	Parameters
Input Image	227x227x3	0	0	0
Conv-1	55x55x96	34,848	96	34,944
MaxPool-1	27x27x96	0	0	0
Conv-2	27x27x256	614,400	256	614,656
MaxPool-2	13x13x256	0	0	0
Conv-3	13x13x384	884,736	384	885,120
Conv-4	13x13x384	1,327,104	384	1,327,488
Conv-5	13x13x256	884,736	256	884,992
MaxPool-3	6x6x256	0	0	0
FC-1	4096×1	37,748,736	4,096	37,752,832
FC-2	4096×1	16,777,216	4,096	16,781,312
FC-3	1000×1	4,096,000	1,000	4,097,000
Output	1000×1	0	0	0
Total				**62,378,344**



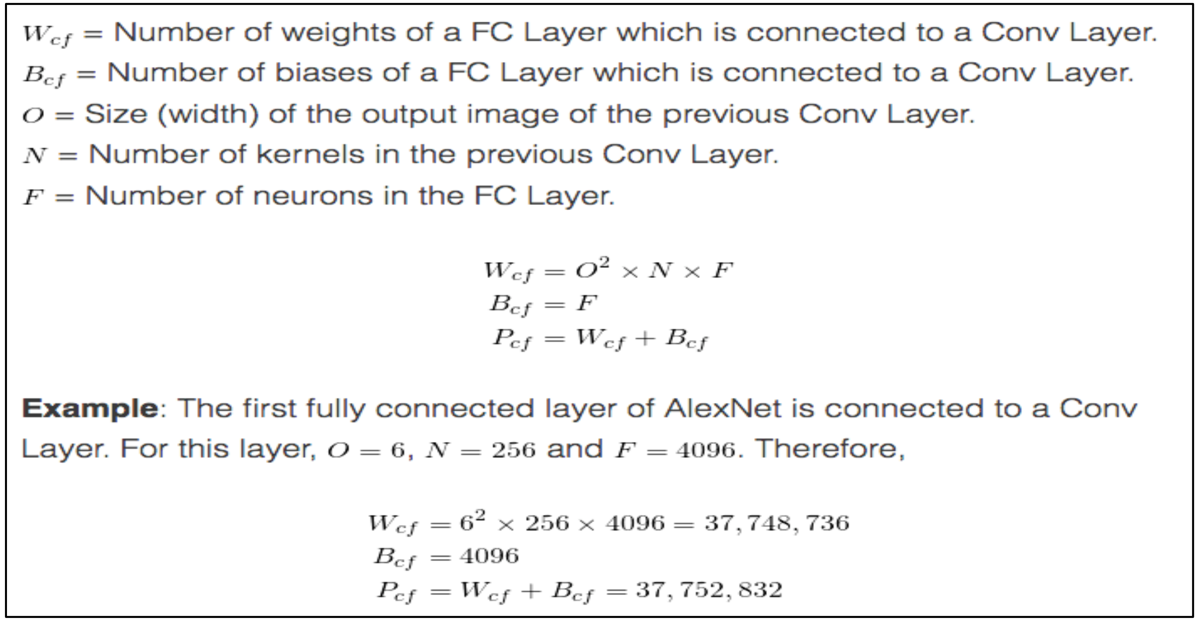
W_{ff} = Number of weights of a FC Layer which is connected to an FC Layer.
B_{ff} = Number of biases of a FC Layer which is connected to an FC Layer.
P_{ff} = Number of parameters of a FC Layer which is connected to an FC Layer.
F = Number of neurons in the FC Layer.
F_{-1} = Number of neurons in the previous FC Layer.

$$\begin{aligned} W_{ff} &= F_{-1} \times F \\ B_{ff} &= F \\ P_{ff} &= W_{ff} + B_{ff} \end{aligned}$$

In the above equation, $F_{-1} \times F$ is the total number of connection weights from neurons of the previous FC Layer the neurons of the current FC Layer. The total number of biases is the same as the number of neurons (F).

Example: The last fully connected layer of AlexNet is connected to an FC Layer. For this layer, $F_{-1} = 4096$ and $F = 1000$. Therefore,

$$\begin{aligned} W_{ff} &= 4096 \times 1000 = 4,096,000 \\ B_{ff} &= 1,000 \\ P_{ff} &= W_{ff} + B_{ff} = 4,097,000 \end{aligned}$$



W_{cf} = Number of weights of a FC Layer which is connected to a Conv Layer.
B_{cf} = Number of biases of a FC Layer which is connected to a Conv Layer.
O = Size (width) of the output image of the previous Conv Layer.
N = Number of kernels in the previous Conv Layer.
F = Number of neurons in the FC Layer.

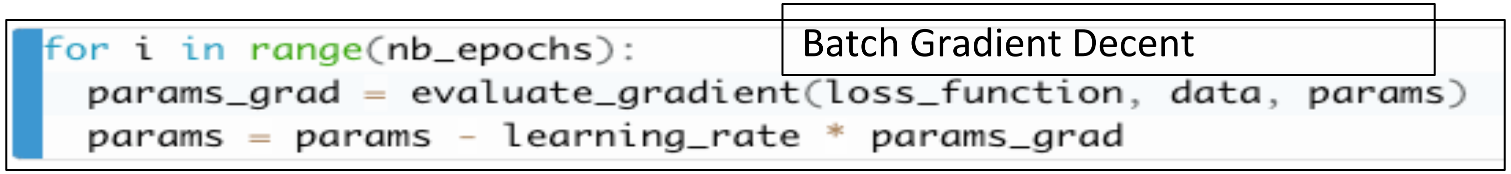
$$\begin{aligned} W_{cf} &= O^2 \times N \times F \\ B_{cf} &= F \\ P_{cf} &= W_{cf} + B_{cf} \end{aligned}$$

Example: The first fully connected layer of AlexNet is connected to a Conv Layer. For this layer, $O = 6$, $N = 256$ and $F = 4096$. Therefore,

$$\begin{aligned} W_{cf} &= 6^2 \times 256 \times 4096 = 37,748,736 \\ B_{cf} &= 4096 \\ P_{cf} &= W_{cf} + B_{cf} = 37,752,832 \end{aligned}$$

https://www.learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network/

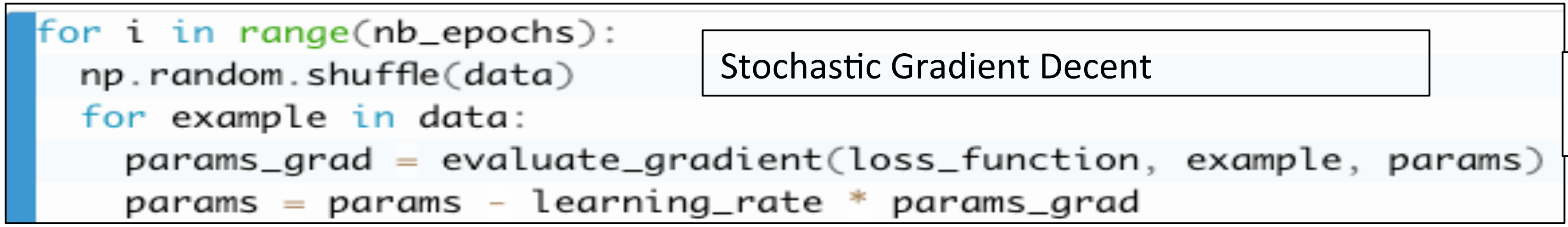
GD Optimization Algorithms



```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Batch Gradient Decent

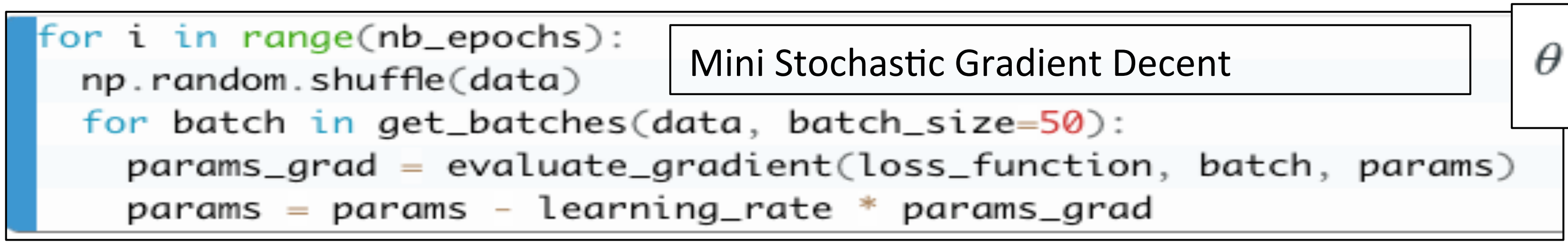
$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$



```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Stochastic Gradient Decent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$



```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Mini Stochastic Gradient Decent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

- ✓ Choosing a proper learning rate can be difficult, slow convergence or diverge!!
- ✓ Learning rate have to be defined in advance and are thus unable to adapt.
- ✓ If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent.
- ✓ Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima.

https://ruder.io/optimizing-gradient-descent/index.html

https://sites.google.com/view/mlss-2019/lectures-and-tutorials

Gradient-based Optimization with Momentum

https://ruder.io/optimizing-gradient-descent/index.html

Momentum [5] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Image 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Momentum

Nesterov
accelerated gradien

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

Now that we are able to adapt our updates to the slope of our error function and speed up SGD in turn, we would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.

Adagrad uses a different learning rate for every parameter θi at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we use gt to denote the gradient at time step t. gt,i is then the partial derivative of the objective function w.r.t. to the parameter θi at time step t:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}).$$

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

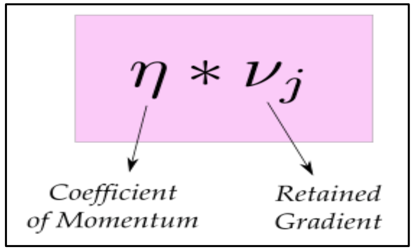
RMSProp Optimizer

$$\text{Repeat Until Convergence } \{$$

$$\nu_j \leftarrow \eta * \nu_j - \alpha * \nabla_w \sum_{1}^{m} L_m(w)$$

$$\omega_j \leftarrow \nu_j + \omega_j$$

$$\}$$



$$\eta * \nu_j$$

Coefficient of Momentum

Retained Gradient

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}).$$

$$\nu_1 = -G_1$$

$$\nu_2 = -0.9 * G_1 - G_2$$

$$\nu_3 = -0.9 * (0.9 * G_1 - G_2) - G3 = -0.81 * (G_1) - (0.9) * G_2 - G_3$$

RMSProp also tries to dampen the oscillations, but in a different way than momentum. RMS prop also takes away the need to adjust learning rate, and does it automatically. More so, RMSProp choses a different learning rate for each parameter.

$For\ each\ Parameter\ w^j$

$(j\ subscript\ dropped\ for\ clarity)$

$$\nu_t = \rho\nu_{t-1} + (1 - \rho) * g_t^2$$

$$\Delta\omega_t = -\frac{\eta}{\sqrt{\nu_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

η : *Initial Learning rate*

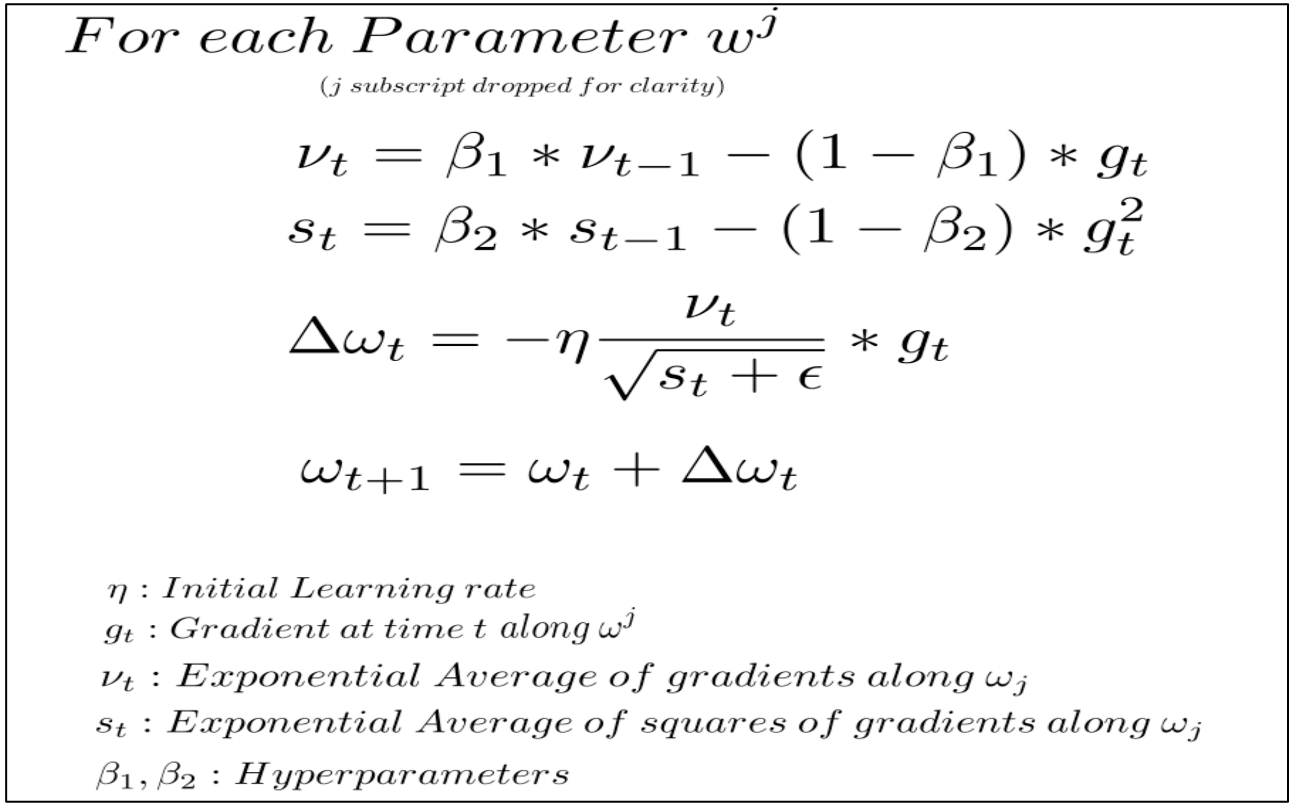
ν_t : *Exponential Average of squares of gradients*

g_t : *Gradient at time t along* ω^j

https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

Adam Optimizer

So far, we've seen RMSProp and Momentum take contrasting approaches. While momentum accelerates our search in direction of minima, RMSProp impedes our search in direction of oscillations.
Adam or **Adaptive Moment Optimization** algorithms combines the heuristics of both Momentum and RMSProp. Here are the update equations.



For each Parameter w^j

(j subscript dropped for clarity)

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

η : *Initial Learning rate*

g_t : *Gradient at time t along* ω^j

ν_t : *Exponential Average of gradients along* ω_j

s_t : *Exponential Average of squares of gradients along* ω_j

β_1, β_2 : *Hyperparameters*

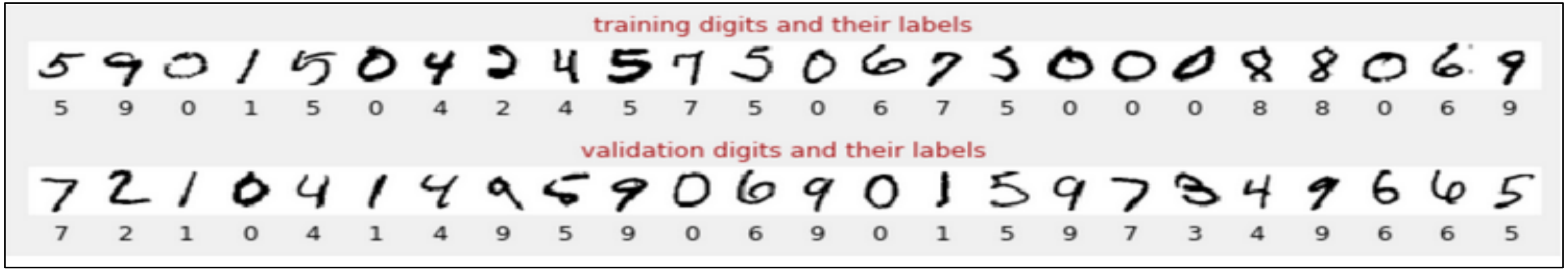
- ✓ Adaptive Moment Estimation
- ✓ Compute adaptive learning rates of parameters
- ✓ Estimates first and second moments of gradients

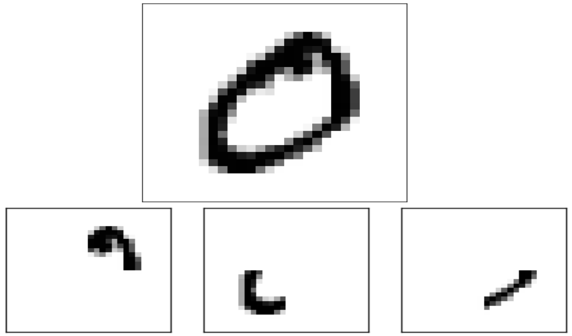
The hyperparameter *beta1* is generally kept around 0.9 while *beta_2* is kept at 0.99. Epsilon is chosen to be 1e-10 generally.

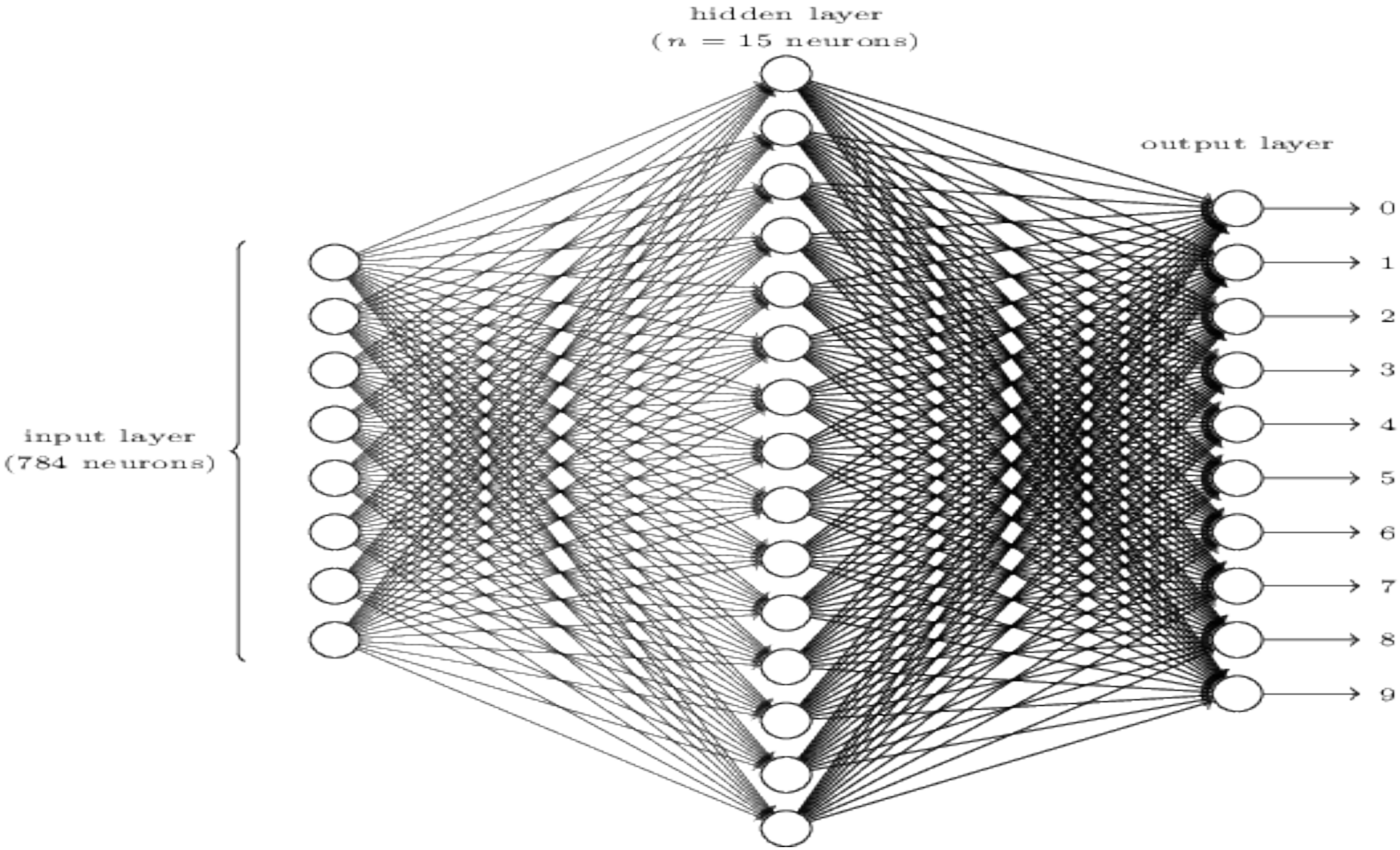
https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

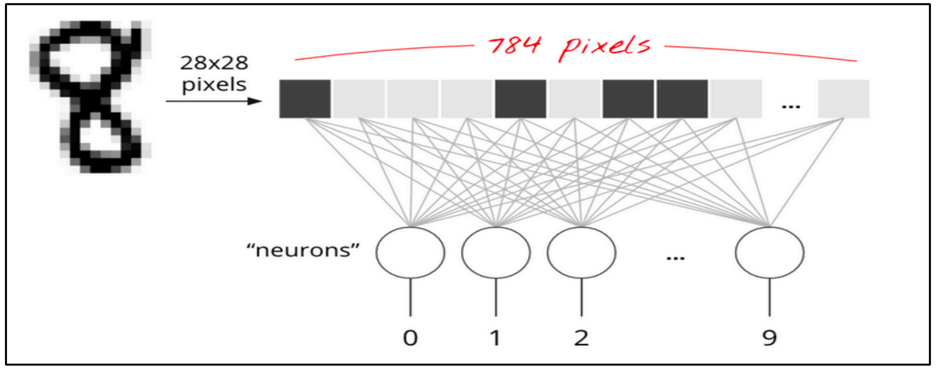
MNIST Example (28x28 pixels)

neuralnetworksanddeeplearning.com









Tensorflow Example

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist

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

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
```

Tensorflow is imported

Sets the mnist dataset to variable “mnist”

Loads the mnist dataset

Builds the layers of the model
4 layers in this model

```
predictions = model(x_train[:1]).numpy()
predictions

tf.nn.softmax(predictions).numpy()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

loss_fn(y_train[:1], predictions).numpy()

model.compile(optimizer='adam', loss=loss_fn, metrics=['accuracy'])
```

returns a vector of “logits” scores, one for each class

Converts the logits to probabilities for each class

Returns scalar loss for each example

Loss Function

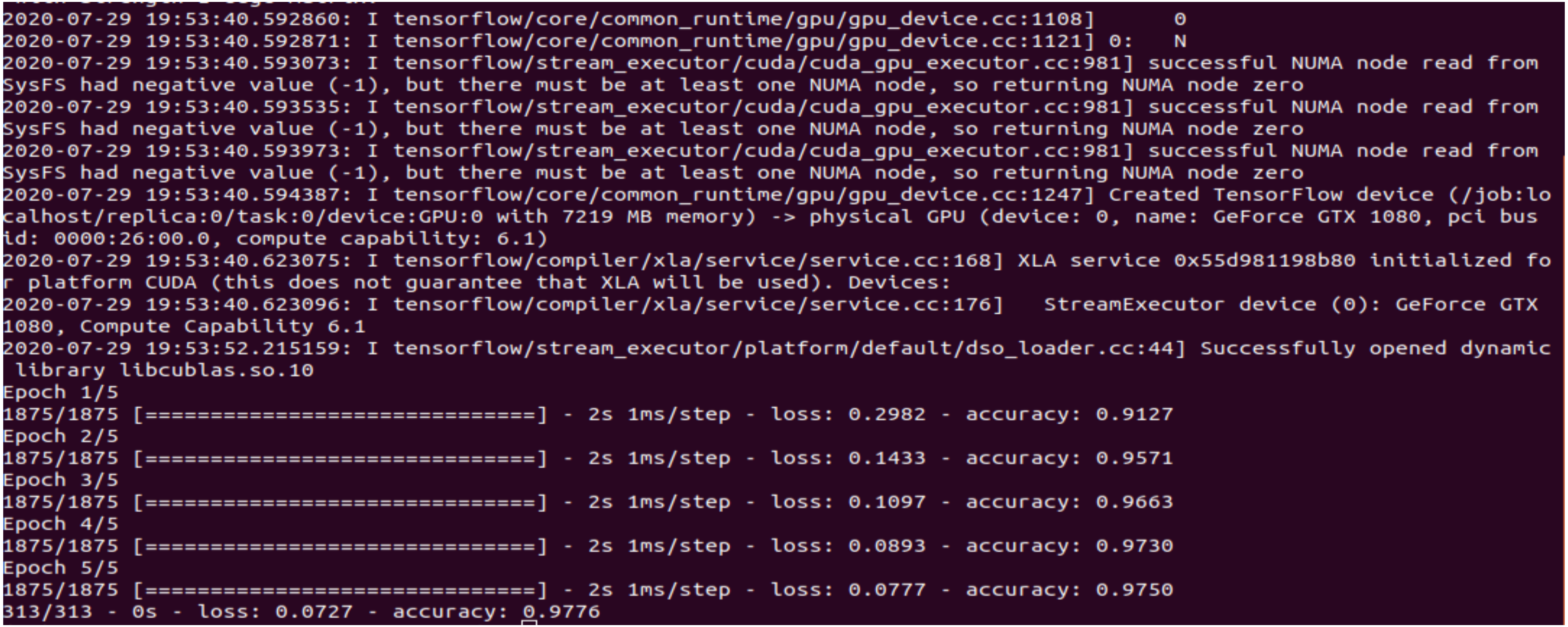
Compiles the model with the adam optimizer

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

model.evaluate(x_test,  y_test, verbose=2)
```

Adjusts model parameters to minimize the loss
Tests the model performance on a test set

Tensorflow Example

The model is trained in about 10 seconds completing 5 epochs with a Nvidia GTX 1080 GPU and has an accuracy of around 97-98%



```
2020-07-29 19:53:40.592860: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1108]      0
2020-07-29 19:53:40.592871: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1121] 0:   N
2020-07-29 19:53:40.593073: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:981] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2020-07-29 19:53:40.593535: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:981] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2020-07-29 19:53:40.593973: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:981] successful NUMA node read from
SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2020-07-29 19:53:40.594387: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1247] Created TensorFlow device (/job:lo
calhost/replica:0/task:0/device:GPU:0 with 7219 MB memory) -> physical GPU (device: 0, name: GeForce GTX 1080, pci bus
id: 0000:26:00.0, compute capability: 6.1)
2020-07-29 19:53:40.623075: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x55d981198b80 initialized fo
r platform CUDA (this does not guarantee that XLA will be used). Devices:
2020-07-29 19:53:40.623096: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): GeForce GTX
1080, Compute Capability 6.1
2020-07-29 19:53:52.215159: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic
 library libcublas.so.10
Epoch 1/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.2982 - accuracy: 0.9127
Epoch 2/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.1433 - accuracy: 0.9571
Epoch 3/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.1097 - accuracy: 0.9663
Epoch 4/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0893 - accuracy: 0.9730
Epoch 5/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.0777 - accuracy: 0.9750
313/313 - 0s - loss: 0.0727 - accuracy: 0.9776
```

```
#include <cstdint>
#include <cstdio>
#include <vector>
#include <iostream>

/* we must include magmadnn as always */
#include "magmadnn.h"

/* tell the compiler we're using functions from the magmadnn namespace */
using namespace magmadnn;

// void print_image(uint32_t image_idx, Tensor<float> *images, Tensor<float> *labels,
uint32_t n_rows, uint32_t n_cols);

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

  using T = float;

  // Every magmadnn program must begin with magmadnn_init. This
  // allows magmadnn to test the environment and initialize some GPU
  // data.
   magmadnn_init();

   // Location of the MNIST dataset
   std::string const mnist_dir = ".";
   // Load MNIST trainnig dataset
   magmadnn::data::MNIST<T> train_set(mnist_dir, magmadnn::data::Train);
   magmadnn::data::MNIST<T> test_set(mnist_dir, magmadnn::data::Test);
```

MagmaDNN MNIST

```
 // Number of features
   uint32_t n_features;
   // Memory used for training (CPU or GPU)
   memory_t training_memory_type;

   n_features = train_set.nrows() * train_set.ncols();

   /* if you run this program with a number argument, it will print out that number sample
on the command line */
   if (argc == 2) {
     train_set.print_image(
         std::atoi(argv[1]));
   }
    // Initialize our model parameters
   model::nn_params_t params;
   params.batch_size = 128; /* batch size: the number of samples to process in each mini-
batch */
   params.n_epochs = 10;    /* # of epochs: the number of passes over the entire training set
*/
   params.learning_rate = 0.05;

   // This is only necessary for a general example which can handle
   // all install types. Typically, When you write your own MagmaDNN
   // code you will not need macros as you can simply pass DEVICE to
   // the x_batch constructor.
#if defined(MAGMADNN_HAVE_CUDA)
   // training_memory_type = DEVICE;
   // std::cout << "Training on GPUs" << std::endl;
   training_memory_type = HOST;
   std::cout << "Training on CPUs" << std::endl;
#else
   training_memory_type = HOST;
   std::cout << "Training on CPUs" << std::endl;
#endif
```

```
// Creating the network
   // Create a variable (of type T=float) with size (batch_size x
   // n_features) This will serve as the input to our network.
   auto x_batch = op::var<T>(
        "x_batch",
        {params.batch_size, train_set.nchanels(),  train_set.nrows(), train_set.ncols()},
        {NONE, {}}, training_memory_type);

   // Initialize the layers in our network
   auto input = layer::input(x_batch);
   auto pool = layer::pooling(input->out(), {2, 2}, {0, 0}, {2, 2}, AVERAGE_POOL);
   // auto pool = layer::pooling(input->out(), {2, 2}, {0, 0}, {2, 2}, MAX_POOL);
   auto flatten = layer::flatten(pool->out());
   // auto flatten = layer::flatten(input->out());

   auto fc1 = layer::fullyconnected(flatten->out(), 784, false);
   // auto fc1 = layer::fullyconnected(input->out(), 784, false);
   auto act1 = layer::activation(fc1->out(), layer::RELU);
   auto fc2 = layer::fullyconnected(act1->out(), 500, false);
   auto act2 = layer::activation(fc2->out(), layer::RELU);

   auto fc3 = layer::fullyconnected(act2->out(), train_set.nclasses(), false);
   auto act3 = layer::activation(fc3->out(), layer::SOFTMAX);
   auto output = layer::output(act3->out());

// Wrap each layer in a vector of layers to pass to the model
   std::vector<layer::Layer<float> *> layers =
      {input,
       // Pooling layer
       pool,
       // flatten layer
       flatten,
```

```
       // First fully-connected layer
       fc1, act1,
       // Second fully-connected layer
       fc2, act2,
       //Third fully-connected layer
       fc3, act3,
       output};

   // This creates a Model for us. The model can train on our data
   // and perform other typical operations that a ML model     can
   // - layers: the previously created vector of layers containing our
   // network
    model::NeuralNetwork<float> model(layers, optimizer::CROSS_ENTROPY, optimizer::SGD,
params);

   // metric_t records the model metrics such as accuracy, loss, and
   // training time
   model::metric_t metrics;

   /* fit will train our network using the given settings.
      X: independent data
      y: ground truth
      metrics: metric struct to store run time metrics
      verbose: whether to print training info during training or not */
   model.fit(&train_set.images(), &train_set.labels(), metrics, true);

  // Clean up memory after training
   delete output;

   // Every magmadnn program should call magmadnn_finalize before
   // exiting
   magmadnn_finalize();
   return 0;
}
```

How to Build Your Own Neural Network Framework

- ✓ **Lecture 1 : Overview of Deep Neural Network : MLP, CNN, CUDNN, MKL DNN, TensorFlow, MagmaDNN, MNIST example**
- ✓ **Lecture 2 : GPU computing : Linear Algebra, MKL, CUDA, GPU programming, Magma, mixed precision, exercises**
- ✓ **Lecture 3 : Programming structure of a DNN framework : I/O, Memory Management, Tensor, layers, activation and fit functions, CFIR 100, MagmaDNN**
- ✓ **Lecture 4 : Multiple GPUs and in-depth example exercises : data and model parallelism, distributed Computing, VGG16, ResNet 18, 50, ImageNet.**

GPU Computing

Stan Tomov
Research Asst. Professor

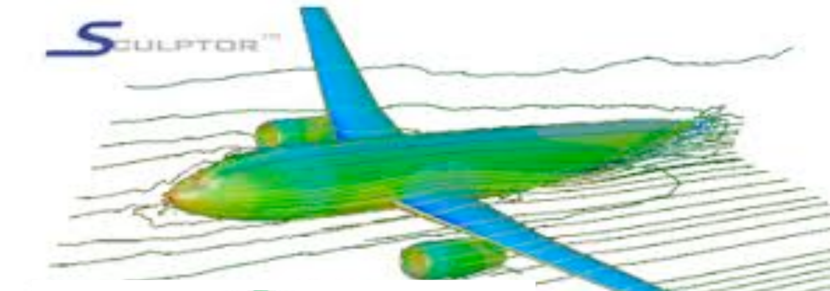
The Innovative Computing Laboratory
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville

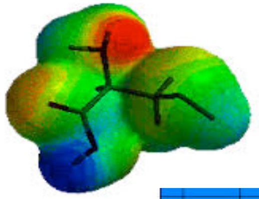
- ✓ **Linear Algebra Overview**
- ✓ **Linear Algebra Libraries**
- ✓ **GPU architectures and programming**
- ✓ **MAGMA libraries**
- ✓ **Performance and benchmarks**
- ✓ **Mixed precision**
- ✓ **Exercises**

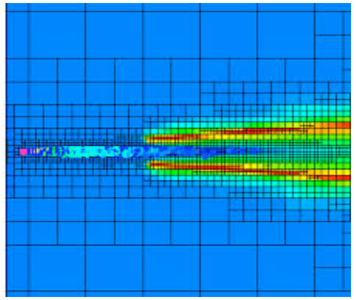
Linear Algebra in Applications

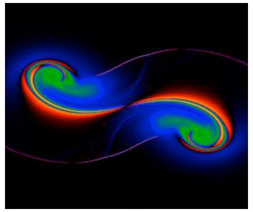
- Dense Linear Algebra (DLA) is needed in a wide variety of science and engineering applications, including ML and data analytics problems:

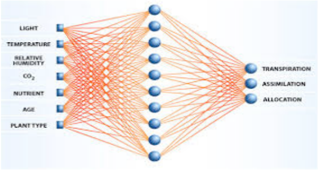
- Linear systems: Solve $Ax = b$
  - Computational electromagnetics, material science, applications using boundary integral equations, airflow past wings, fluid flow around ship and other offshore constructions, and many more
- Least squares: Find x to minimize $\| Ax – b \|$
  - Computational statistics (e.g., linear least squares or ordinary least squares), econometrics, control theory, signal processing, curve fitting, and many more
- Eigenproblems: Solve $Ax = \lambda x$
  - Computational chemistry, quantum mechanics, material science, face recognition, PCA, data-mining, marketing, Google Page Rank, spectral clustering, vibrational analysis, compression, and many more
- SVD: $A = U \Sigma V^*$ ($Au = \sigma v$ and $A^*v = \sigma u$)
  - Information retrieval, web search, signal processing, big data analytics, low rank matrix approximation, total least squares minimization, pseudo-inverse, and many more
- Many variations depending on structure of A
  - A can be symmetric, positive definite, tridiagonal, Hessenberg, banded, sparse with dense blocks, etc.
- DLA is crucial to the development of sparse solvers











LA for modern architectures

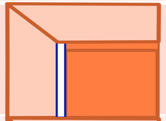
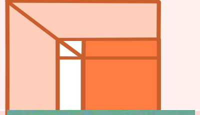
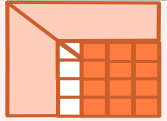
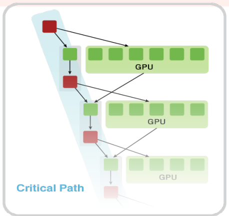
- **Leverage latest numerical algorithms and building blocks**
  MAGMA, PLASMA, SLATE (DOE funded) , MAGMA Sparse, POMPEI project*

- **Polymorphic approach**
  Use MAGMA sub-packages for various architectures;
  Provide portability through single templated sources using C++

- **Programming model**
  BLAS tasking + scheduling

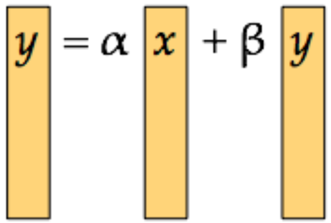
- **Open standards**
  OpenMP4 tasking + MPI

Use of BLAS for portability

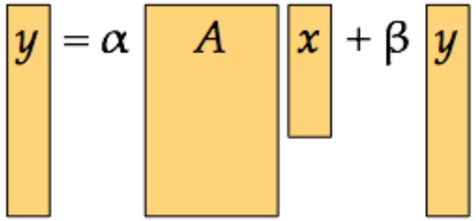
Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Level 1 **BLAS**
LAPACK (80's) (Blocking, cache friendly)		Level 3 **BLAS**
ScaLAPACK (90's) (Distributed Memory)		**PBLAS**
PLASMA (00's) New Algorithms (many-core friendly)		**BLAS** on tiles + DAG scheduling
MAGMA Hybrid Algorithms (heterogeneity friendly)		**BLAS** tasking + (CPU / GPU / Xeon Phi) hybrid scheduling

BLAS: Basic Linear Algebra Subroutines

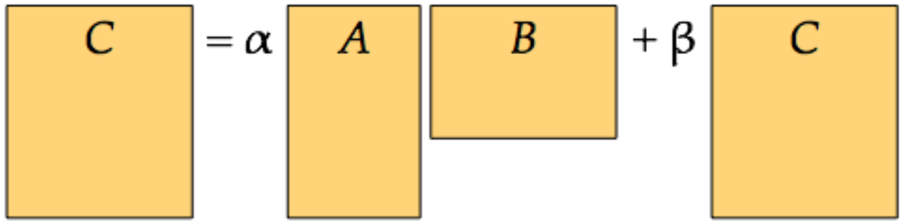
- Level 1 BLAS — vector operations
  - $O(n)$ data and flops (floating point operations)
  - Memory bound:
    $O(1)$ flops per memory access



- Level 2 BLAS — matrix-vector operations
  - $O(n^2)$ data and flops
  - Memory bound:
    $O(1)$ flops per memory access



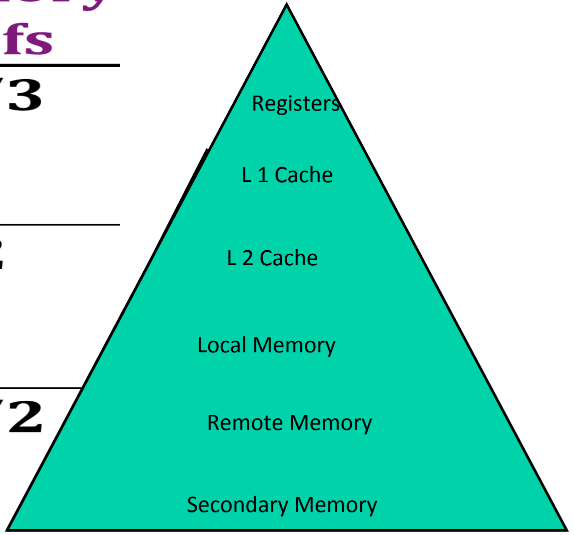
- Level 3 BLAS — matrix-matrix operations
  - $O(n^2)$ data, $O(n^3)$ flops
  - Surface-to-volume effect
  - Compute bound:
    $O(n)$ flops per memory access



Why Higher Level BLAS?

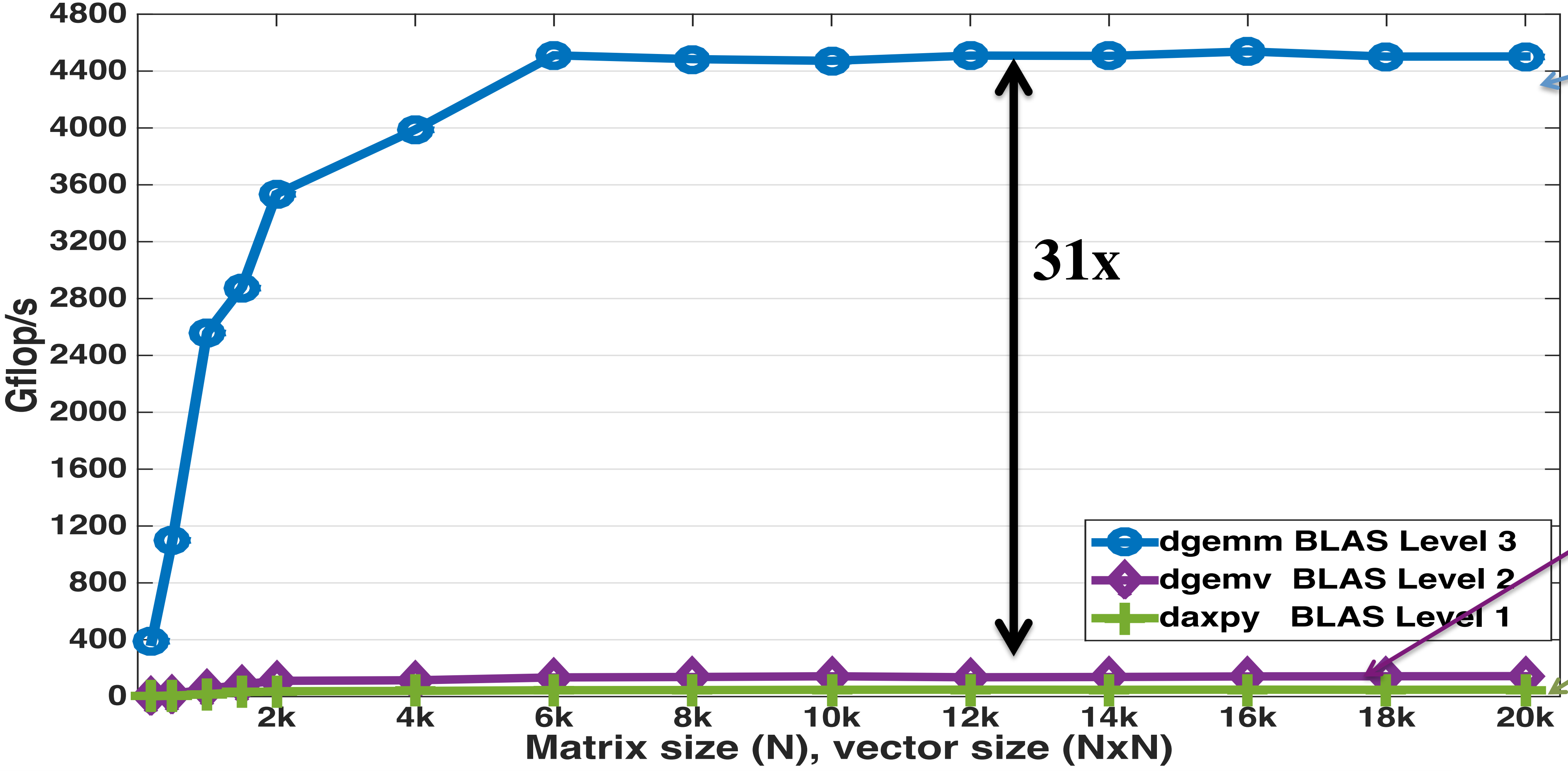
- By taking advantage of the principle of locality:
- Present the user with as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.
- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

BLAS	Memory Refs	Flops	Flops/ Memory Refs
Level 1 $y=y+\alpha x$	3n	2n	2/3
Level 2 $y=y+Ax$	n^2	$2n^2$	2
Level 3 $C=C+AB$	$4n^2$	$2n^3$	n/2



Level 1, 2 and 3 BLAS

Nvidia **P100**, 1.19 GHz, Peak DP = 4700 Gflop/s



C = C + A*B

4503 Gflop/s

y = y + A*x

145 Gflop/s

y = α *x + y

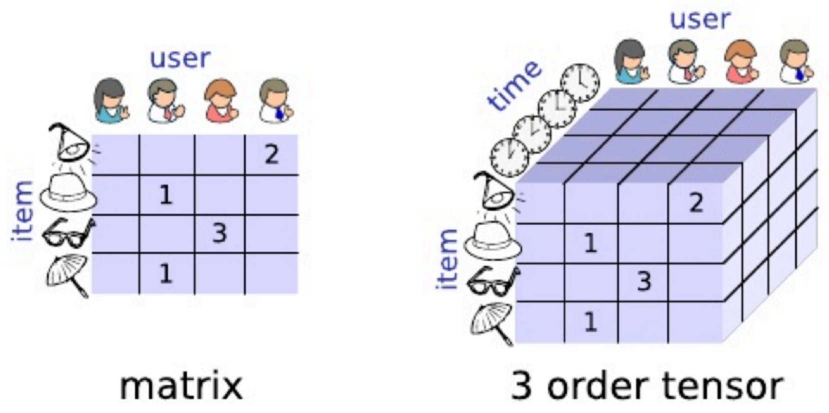
52 Gflop/s

Nvidia P100
The theoretical peak double precision is 4700 Gflop/s
CUDA version 8.0

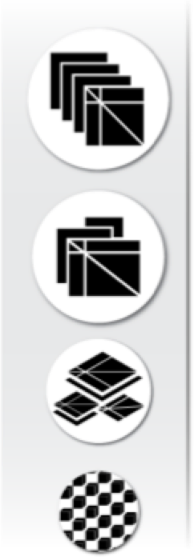
What about accelerated LA for Data Analytics?

- Traditional libraries like MAGMA can be used as backend to accelerate the LA computations in data analytics applications
- Need support for
  1) New data layouts, 2) Acceleration for small matrix computations, 3) Data analytics tools

Need data processing and analysis support for
Data that is multidimensional / relational



Small matrices, tensors, and batched computations



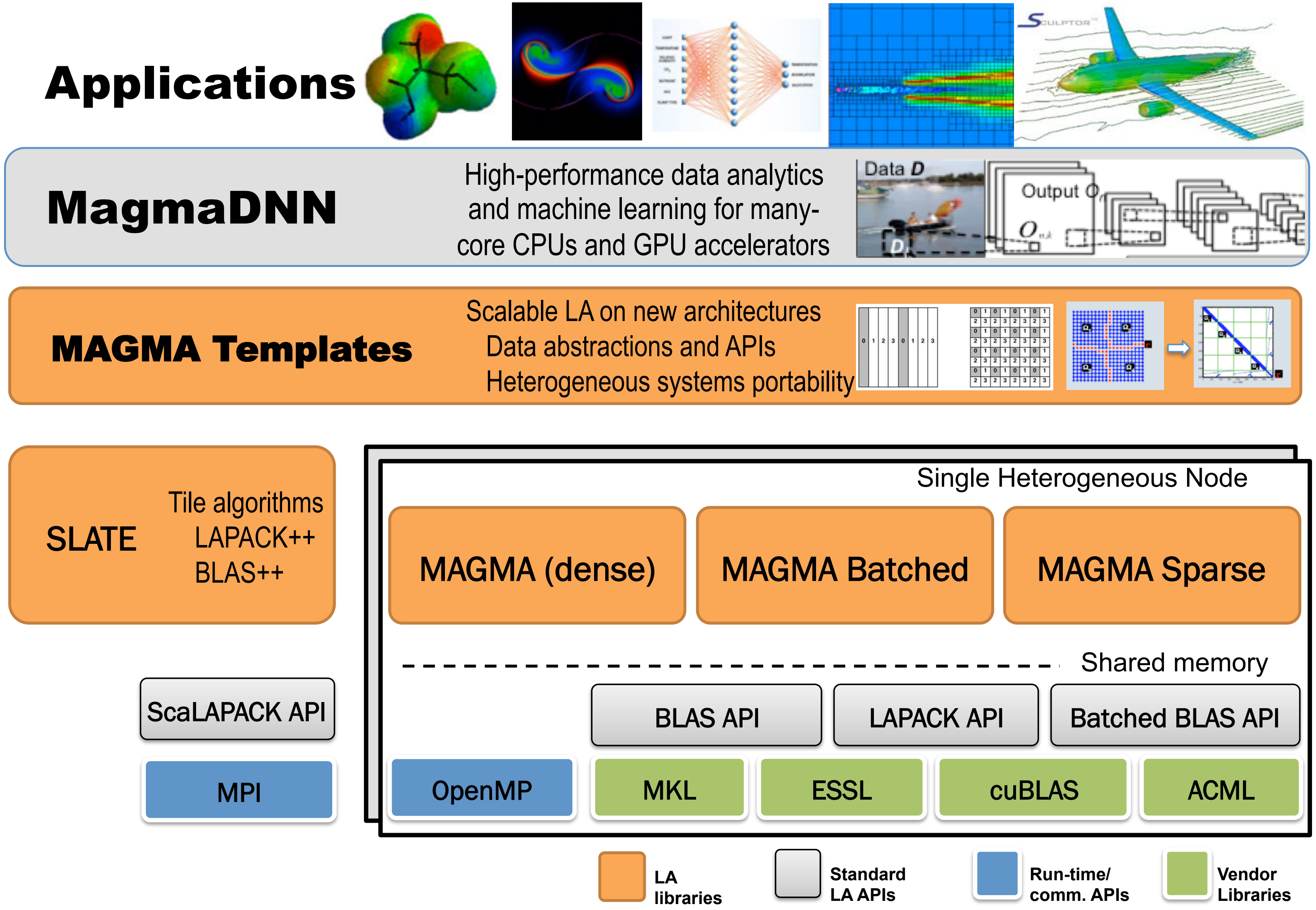
Fixed-size batches

Variable-size batches

Dynamic batches

Tensors

MagmaDNN software stack



Applications

MagmaDNN

High-performance data analytics and machine learning for many-core CPUs and GPU accelerators

MAGMA Templates

Scalable LA on new architectures
Data abstractions and APIs
Heterogeneous systems portability

SLATE

Tile algorithms
LAPACK++
BLAS++

ScaLAPACK API

MPI

Single Heterogeneous Node

MAGMA (dense)

MAGMA Batched

MAGMA Sparse

Shared memory

BLAS API

LAPACK API

Batched BLAS API

OpenMP

MKL

ESSL

cuBLAS

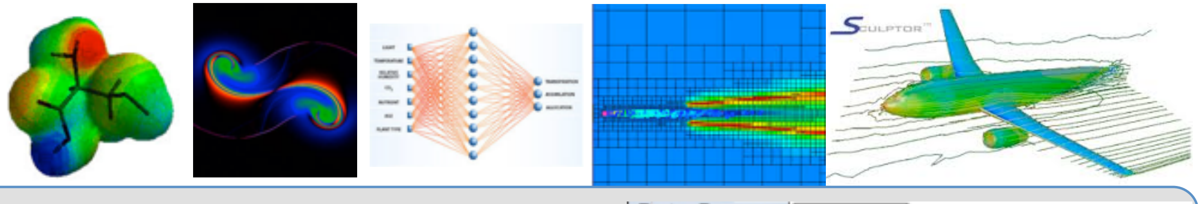
ACML

🟧 LA libraries
⬜ Standard LA APIs
🟦 Run-time/ comm. APIs
🟩 Vendor Libraries

- **MagmaDNN is HP Data Analytics and ML framework built around the MAGMA libraries aimed at providing a modularized and efficient tool for training DNNs.**
- **MagmaDNN makes use of the highly optimized MAGMA libraries giving significant speed boosts over other modern frameworks.**

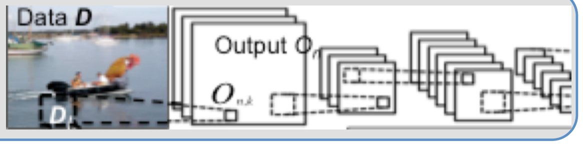
MagmaDNN software stack

Applications



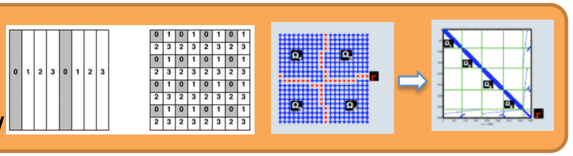
MagmaDNN

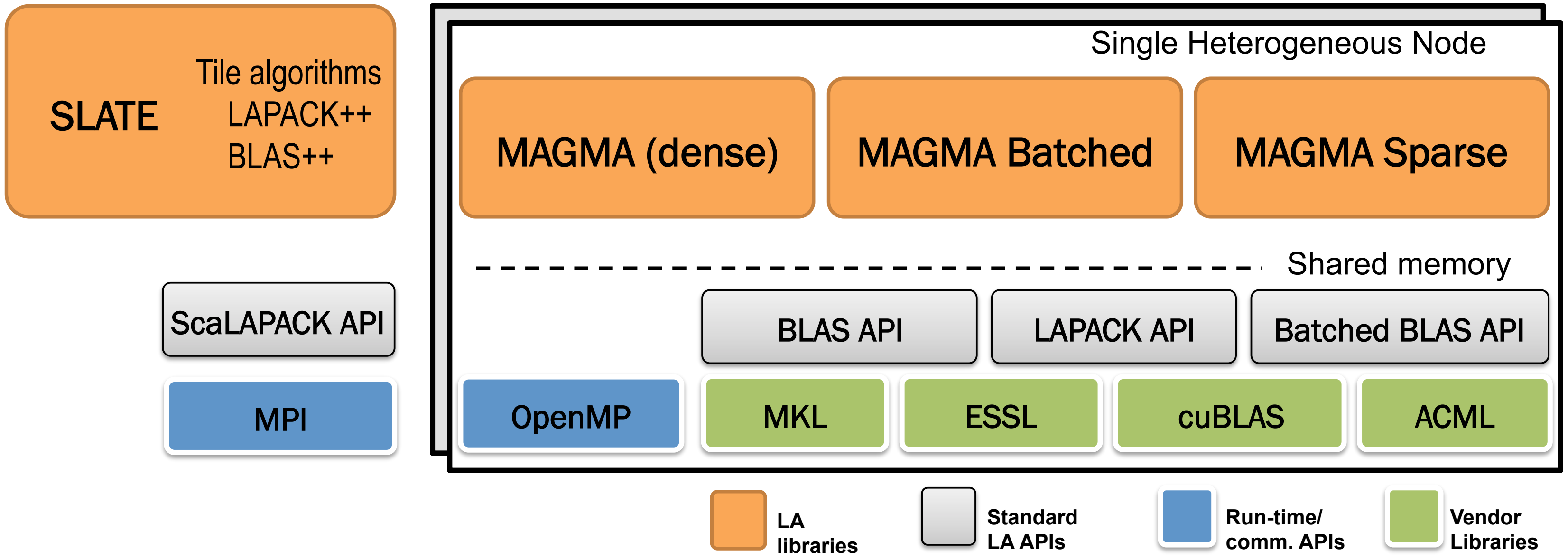
High-performance data analytics and machine learning for many-core CPUs and GPU accelerators



MAGMA Templates

Scalable LA on new architectures
Data abstractions and APIs
Heterogeneous systems portability





SLATE

Tile algorithms
LAPACK++
BLAS++

ScaLAPACK API

MPI

Single Heterogeneous Node

MAGMA (dense) | MAGMA Batched | MAGMA Sparse

Shared memory

BLAS API | LAPACK API | Batched BLAS API

OpenMP | MKL | ESSL | cuBLAS | ACML

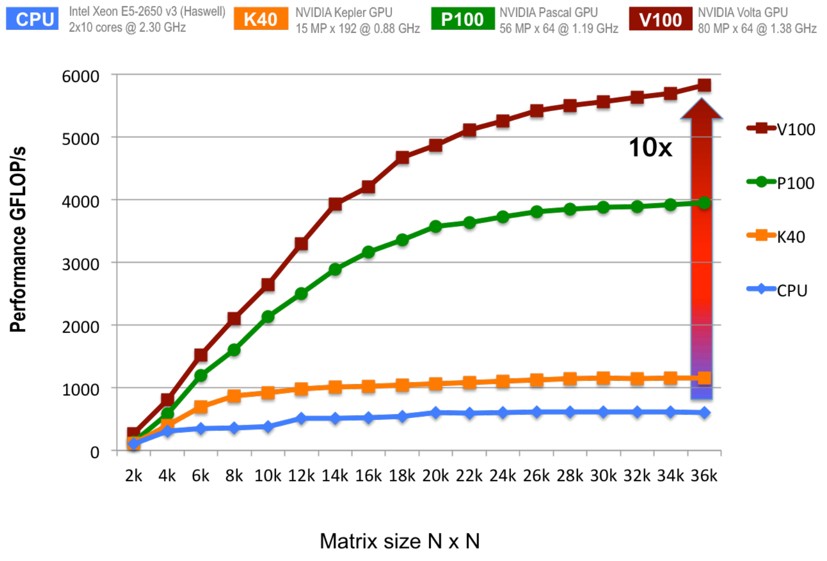
LA libraries

Standard LA APIs

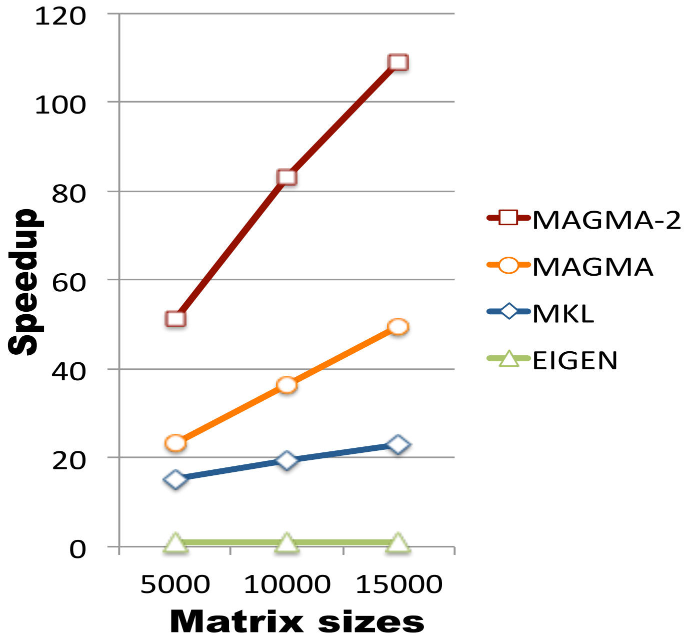
Run-time/ comm. APIs

Vendor Libraries

MAGMA 2.3 LU factorization in double precision arithmetic



SVD performance speedup



Outline

Availability

Routines

Code

Testers

Methodology

MAGMA

- Availability
  - http://icl.utk.edu/magma/ download (latest MAGMA 2.5.3) , documentation, forum
  - https://bitbucket.org/icl/magma Git repo
- Support
  - Linux, macOS, Windows
  - CUDA >= 7.0; recommend latest CUDA
  - CUDA architecture >= 2.0 (Fermi, Kepler, Maxwell, Pascal, Volta, Turing)
  - BLAS & LAPACK: Intel MKL, OpenBLAS, macOS Accelerate, ...
- May be pre-installed on supercomputers

```
titan-ext1> module avail magma
---------- /sw/xk6/modulefiles ----------
magma/1.3                 magma/1.6.2(default)
```

Installation options

1. **Makefile**
   - Edit make.inc for compilers and flags (see make.inc examples)
   - `magma> make && make install`
2. **CMake**
   - `magma> mkdir build && cd build`
   - `magma/build> cmake ..` or `ccmake ..`
   - Adjust settings, esp. LAPACK_LIBRARIES and GPU_TARGET
   - `magma/build> make && make install`
3. **Spack**
   - Part of xSDK
   - `spack install magma`
   - Caveat: nvcc is picky about its host compiler (gcc, ...)

Outline

Availability

Routines

Code

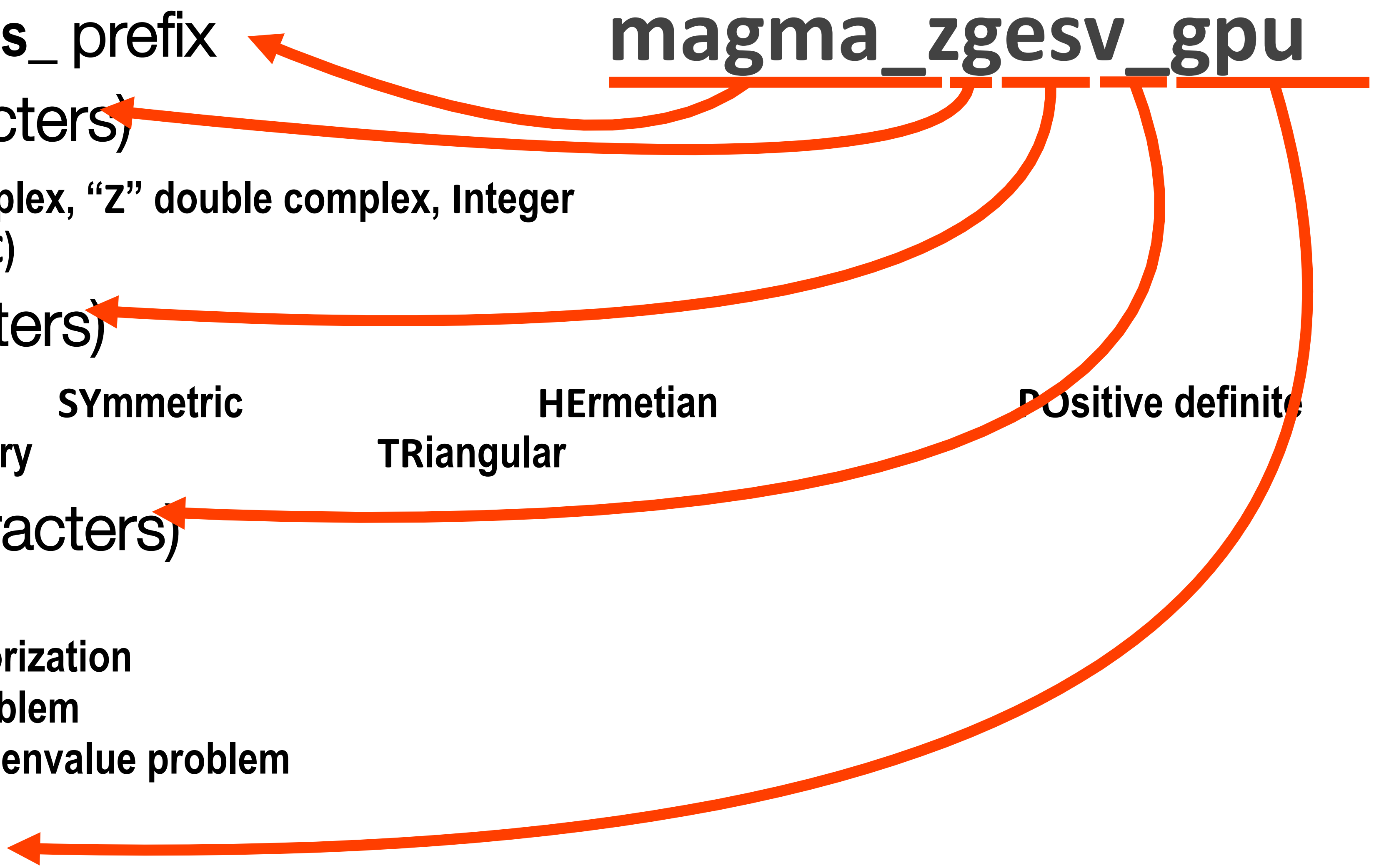
Testers

Methodology

MAGMA Overview

- Hybrid LAPACK-style functions
  - Matrix factorizations: LU, Cholesky, QR, eigenvalue, SVD, ...
  - Solve linear systems and linear least squares, ...
  - Nearly all are synchronous:
    return on CPU when computation is finished
- GPU BLAS and auxiliary functions
  - Matrix-vector multiply, matrix norms, transpose (in-place and out-of-place), ...
  - Most are asynchronous:
    return immediately on CPU; computation proceeds on GPU
- Wrappers around CUDA and cuBLAS
  - BLAS routines (gemm, symm, symv, ...)
  - Copy host ⇔ device, queue (stream) support, GPU malloc & free, ...
- Batched BLAS and LAPACK
- Sparse LA component

Naming example



- **magma_** or **magmablas_** prefix
- **Precision** (1–2 characters)
  - Single, Double, single Complex, “Z” double complex, Integer
    Mixed precision (DS and ZC)
- **Matrix type** (2 characters)
  - GEneral SYmmetric HErmetian POsitive definite
    ORthogonal UNitary TRiangular
- **Operation** (2–3+ characters)

SV	solve
TRF	triangular factorization
EV	eigenvalue problem
GV	generalized eigenvalue problem
etc.	

- **_gpu** suffix for interface

Linear solvers

- Solve linear system: $AX = B$
- Solve linear least squares: minimize $\|AX - B\|_2$

Type	Routine	Mixed precision routine	Interface CPU	 GPU
General	dgesv	dsgesv	✓	✓
Positive definite	dposv	dsposv	✓	✓
Symmetric	dsysv		✓	
Hermitian	zhesv		✓	
Least squares	dgels		✓	✓

Selected routines; complete documentation at http://icl.utk.edu/magma/

Eigenvalue / singular value problems

- **Eigenvalue problem:** $Ax = \lambda x$
- **Generalized eigenvalue problem:** $Ax = \lambda Bx$ **(and variants)**
- **Singular value decomposition:** $A = U\Sigma V^H$

				Interface	
Matrix type	**Operation**	**Routine**		**CPU**	**GPU**
General	SVD	dgesvd,	dgesdd	✓	
General non-symmetric	Eigenvalue	dgeev		✓	
Symmetric	Eigenvalue	dsyevd /	zheevd	✓	✓
Symmetric	Generalized	dsygvd /	zhegvd	✓	

Additional variants; complete documentation at http://icl.utk.edu/magma/
Fastest are divide-and-conquer (gesdd, syevd) and 2-stage versions.

Computational routines

- **Computational routines solve one part of problem**

				Interface	
Matrix type	**Operation**	**Routine**		**CPU**	**GPU**
General	LU	dgetrf		✓	✓
	Solve (given LU)	dgetrs			✓
	Inverse	dgetri			✓
SPD	Cholesky	dpotrf		✓	✓
	Solve (given LL^T)	dpotrs			✓
	Inverse	dpotri		✓	✓
General	QR	dgeqrf		✓	✓
	Generate Q	dorgqr /	zungqr	✓	✓
	Multiply by Q	dormqr /	zunmqr	✓	✓

Selected routines; complete documentation at http://icl.utk.edu/magma/

BLAS and auxiliary routines

Category	**Operation**	**Routine** (all GPU interface)
Level 1 BLAS	$y = \alpha x + y$	daxpy
	$r = x^{\mathrm{T}} y$	ddot
Level 2 BLAS	$y = \alpha A x + \beta y$, general A	dgemv
	$y = \alpha A x + \beta y$, symmetric A	dsymv
Level 3 BLAS	$C = \alpha AB + \beta C$	dgemm
	$C = \alpha AB + \beta C$, symmetric A	dsymm
	$C = \alpha AA^T + \beta C$, symmetric C	dsyrk
Auxiliary	$\lVert A \rVert_1$, $\lVert A \rVert_{\mathrm{inf}}$, $\lVert A \rVert_{\mathrm{fro}}$, $\lVert A \rVert_{\mathrm{max}}$	dlange (norm, general A)
		dlansy (norm, symmetric A)
	$B = A^{\mathrm{T}}$ (out-of-place)	dtranspose
	$A = A^{\mathrm{T}}$ (in-place, square)	dtranspose_inplace

Selected routines; complete documentation at http://icl.utk.edu/magma/

Outline

Availability

Routines

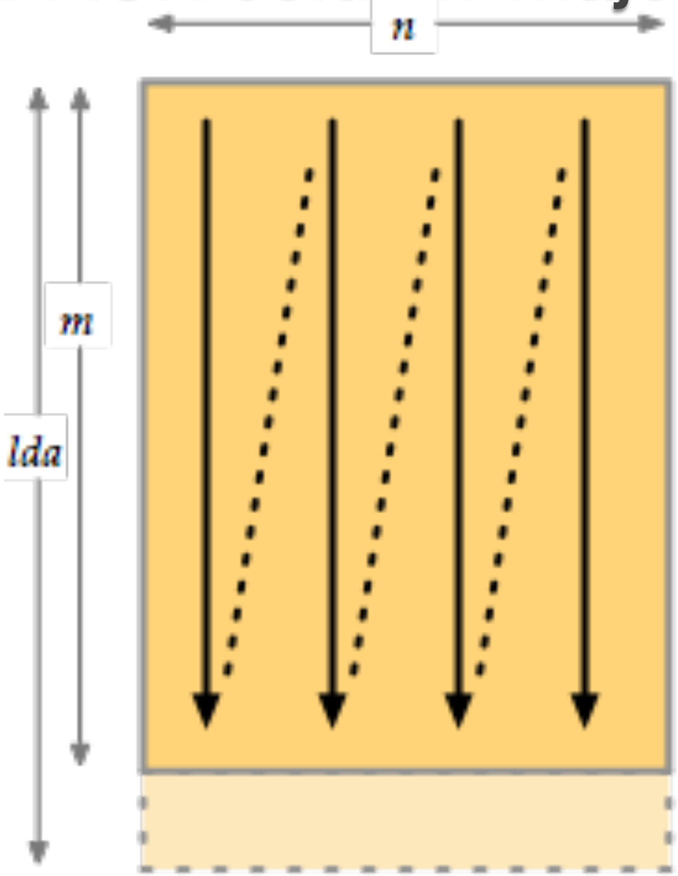
Code

Testers

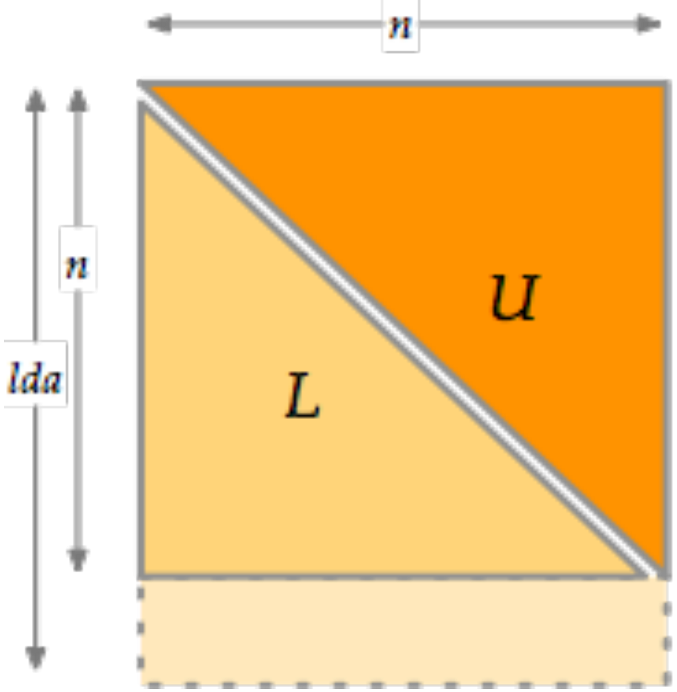
Methodology

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 $AX = B$
  - Double precision, GEneral matrix, SolVe (DGESV)
- Traditional LAPACK call
- Complete codes available at bit.ly/magma-tutorial

```
#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

```
#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();
}
```

Simple example

- MAGMA GPU interface
  - Add _gpu suffix
  - Input & output matrices in GPU device memory (“d” prefix on variables)
  - ipiv still in CPU memory
  - Set GPU stride (ldda) to multiple of 32 for better performance
  - roundup returns ceil(n / 32) * 32
- MAGMA malloc & free
  - Type-safe wrappers around cudaMalloc & cudaFree

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

    int n = 100, nrhs = 10;
    int ldda = magma_roundup( n, 32 );
    int lddx = magma_roundup( n, 32 );
    int* ipiv = new int[ n ];

    double *dA, *dX;
    magma_dmalloc( &dA, ldda*n );
    magma_dmalloc( &dX, lddx*nrhs );
    assert( dA != nullptr );
    assert( dX != nullptr );

    // ... fill in dA and dX (on GPU)

    // solve AX = B where B is in X
    int info;
    magma_dgesv_gpu( n, nrhs,
                     dA, ldda, ipiv,
                     dX, lddx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in dX
    magma_free( dA );
    magma_free( dX );
    delete[] ipiv;

    magma_finalize();
}
```

BLAS example

- Matrix multiply $C = -AB + C$
  - Double-precision, GEneral Matrix Multiply (DGEMM)
- Asynchronous
  - BLAS take queue and are async
  - Return to CPU immediately
  - Queue wraps CUDA stream and cuBLAS handle
  - Can create queue from existing CUDA stream and cuBLAS handle, if required

```
int main( int argc, char** argv )
{
    // ... setup matrices on GPU:
    // m-by-k matrix dA,
    // k-by-n matrix dB,
    // m-by-n matrix dC.

    int device;
    magma_queue_t queue;
    magma_getdevice( &device );
    magma_queue_create( device, &queue );

    // C = -A B + C
    magma_dgemm( MagmaNoTrans,
                 MagmaNoTrans, m, n, k,
                 -1.0, dA, ldda,
                       dB, lddb,
                  1.0, dC, lddc, queue );

    // ... do concurrent work on CPU

    // wait for gemm to finish
    magma_queue_sync( queue );

    // ... use result in dC

    magma_queue_destroy( queue );

    // ... cleanup
}
```

Copy example

- Copy data host ⇔ device
  - setmatrix (host to device)
  - getmatrix (device to host)
  - copymatrix (device to device)
  - setvector (host to device)
  - getvector (device to host)
  - copyvector (device to device)
- Default is synchronous
  - Return when transfer is done
- Strides (lda, ldda) can differ on CPU and GPU
  - Set GPU stride (ldda) to multiple of 32 for better performance

```
int main( int argc, char** argv )
{
    // ... setup A, X in CPU memory;
    // dA, dX in GPU device memory

    int device;
    magma_queue_t queue;
    magma_getdevice( &device );
    magma_queue_create( device, &queue );

    // copy A, X to dA, dX
    magma_dsetmatrix( n, n,
                       A, lda,
                      dA, ldda, queue );
    magma_dsetmatrix( n, nrhs,
                       X, ldx,
                      dX, lddx, queue );

    // ... solve AX = B

    // copy result dX to X
    magma_dgetmatrix( n, nrhs,
                      dX, lddx,
                       X, ldx, queue );

    // ... use result in X

    magma_queue_destroy( queue );

    // ... cleanup
}
```

Async copy

- Add _async suffix
- Use pinned CPU memory
  - Page locked, so DMA can access it
  - Better performance
  - Required by CUDA for async behavior
  - But pinned memory is limited resource, and expensive to allocate
- Overlap:
  - Sending data (host to device)
  - Getting data (device to host)
  - Host computation
  - Device computation

```
int main( int argc, char** argv )
{
    // ... setup dA, dX, queue

    // allocate A, X in pinned CPU memory
    double *A, *X;
    magma_dmalloc_pinned( &A, lda*n );
    magma_dmalloc_pinned( &X, ldx*nrhs );

    // ... fill in A and X

    // copy A, X to dA, dX, then wait
    magma_dsetmatrix_async( n, n,
            A, lda, dA, ldda, queue );
    magma_dsetmatrix_async( n, nrhs,
            X, ldx, dX, lddx, queue );
    magma_queue_sync( queue );

    // ... solve AX = B

    // copy result dX to X, then wait
    magma_dgetmatrix_async( n, nrhs,
            dX, ldx, X, lddx, queue );
    magma_queue_sync( queue );

    // ... use result in X

    magma_free_pinned( A );
    magma_free_pinned( X );

    // ... cleanup
}
```

Outline

Availability

Routines

Code

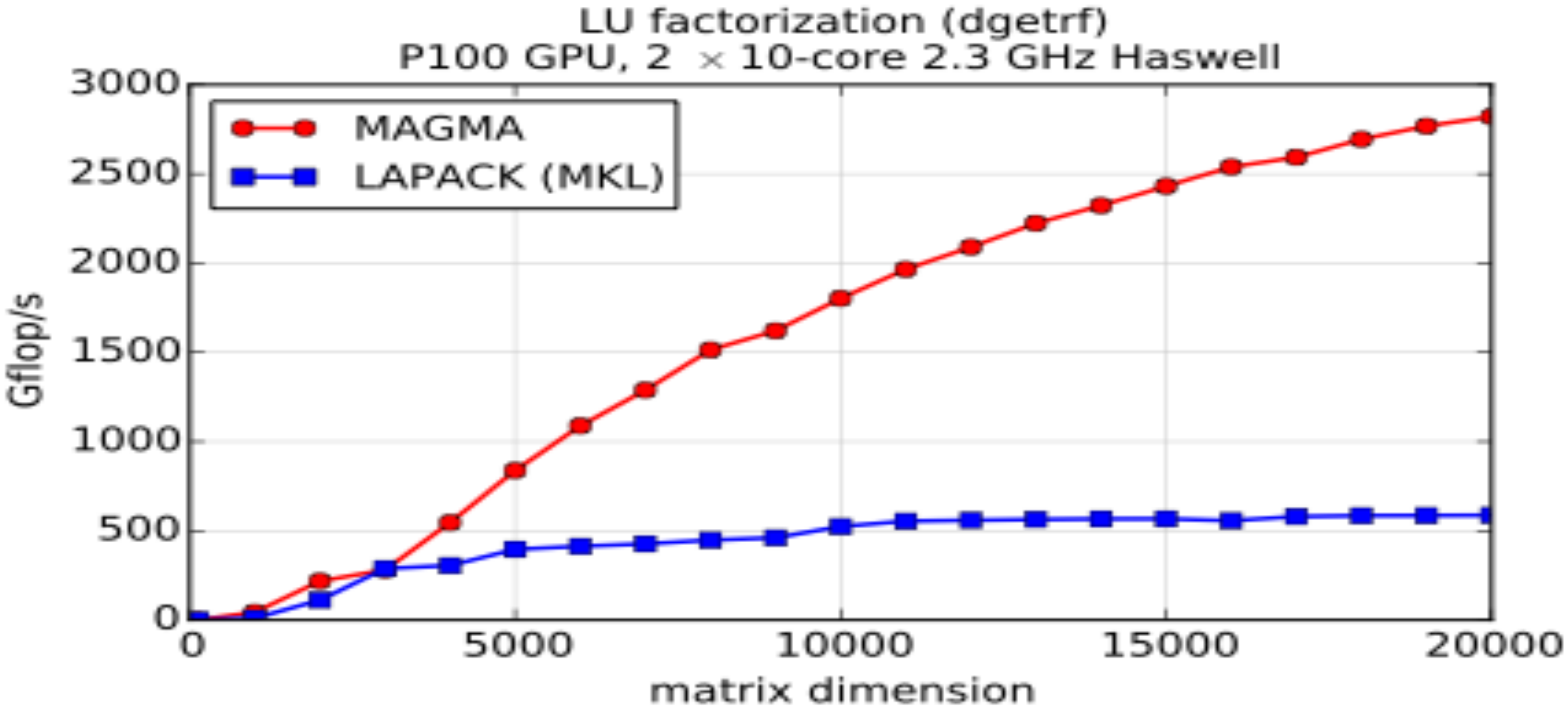
Testers

Methodology

Testers: LU factorization (dgetrf)

```
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
```

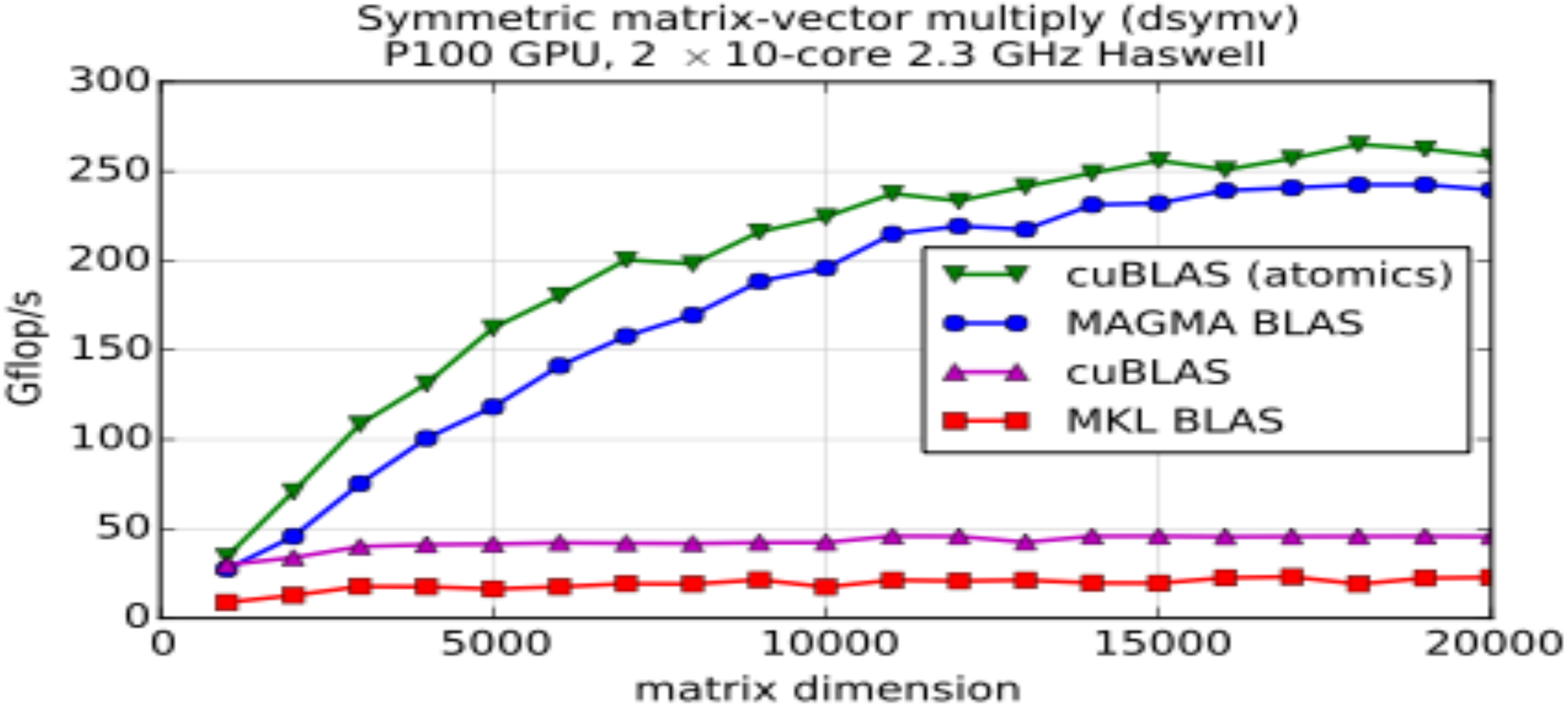
Testers: LU factorization (dgetrf)



Testers: symmetric matrix-vector multiply (dsymv)

```
# (abbreviated output)
magma> cd testing
magma/testing> ./testing_dsymv -n 123 -n 1000:20000:1000 --lapack --check
% MAGMA 2.5.0  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

% uplo = Lower
%   N     MAGMA    Atomics    cuBLAS       CPU   error
%       Gflop/s    Gflop/s   Gflop/s   Gflop/s
%=======================================================
  123      0.76       0.76      0.51      0.58   ok     # warmup run
 1000     27.44      34.41     29.88      8.86   ok
 2000     45.74      70.83     33.91     12.78   ok
 3000     75.30     108.51     40.09     17.76   ok
 4000    100.64     131.23     41.13     17.57   ok
 5000    118.17     162.35     41.46     16.33   ok
 6000    141.21     180.43     42.16     17.55   ok
 7000    157.81     200.44     41.94     19.32   ok
 8000    169.54     198.21     41.78     19.12   ok
 9000    188.40     216.07     42.28     21.50   ok
10000    195.92     224.50     42.36     17.44   ok
11000    214.93     237.51     45.91     21.30   ok
12000    219.33     233.44     45.76     20.81   ok
13000    217.52     241.45     42.49     21.29   ok
14000    231.26     249.06     45.84     19.71   ok
15000    232.12     255.98     45.87     19.60   ok
16000    239.26     250.89     45.58     22.61   ok
17000    240.74     257.13     45.69     23.15   ok
18000    242.45     265.05     45.75     19.09   ok
19000    242.53     262.48     45.81     22.42   ok
20000    239.53     258.24     45.63     22.83   ok
```

Testers: symmetric matrix-vector multiply (dsymv)



Test everything: run_tests.py

- Python script to run:
  - All testers
  - All possible options (left/right, lower/upper, ...)
  - Various size ranges (small, medium, large; square, tall, wide)
- Occasionally, tests fail innocuously
  - E.g., error = 1.1e-15 > tol = 1e-15
- Some experimental routines are known to fail
  - E.g., gegqr_gpu, geqr2x_gpu
  - See magma/BUGS.txt
- Running ALL tests can take > 24 hours

Test everything: run_tests.py

```
magma/testing> python ./run_tests.py *trsm --xsmall --small > trsm.txt
testing_strsm -SL -L -DN -c       ok  # left, lower, non-unit, [no-trans]
testing_dtrsm -SL -L -DN -c       ok
testing_ctrsm -SL -L -DN -c       ok
testing_ztrsm -SL -L -DN -c       ok
testing_strsm -SL -L -DU -c       ok  # left, lower, unit, [no-trans]
testing_dtrsm -SL -L -DU -c       ok
testing_ctrsm -SL -L -DU -c       ok
testing_ztrsm -SL -L -DU -c       ok
testing_strsm -SL -L -C -DN -c    ok  # left, lower, non-unit, conj-trans
testing_dtrsm -SL -L -C -DN -c    ok
testing_ctrsm -SL -L -C -DN -c    ok
testing_ztrsm -SL -L -C -DN -c    ok
...
testing_strsm -SR -U -T -DU -c    ok  # right, upper, unit, trans
testing_dtrsm -SR -U -T -DU -c    ok
testing_ctrsm -SR -U -T -DU -c    ok
testing_ztrsm -SR -U -T -DU -c    ok

**************************************************************************************
summary
**************************************************************************************
 6240 tests in 192 commands passed
   96 tests failed accuracy test
    0 errors detected (crashes, CUDA errors, etc.)
routines with failures:
    testing_ctrsm --ngpu 2 -SL -L -C -DN -c
    testing_ctrsm --ngpu 2 -SL -L -C -DU -c
...
```

Outline

Availability

Routines

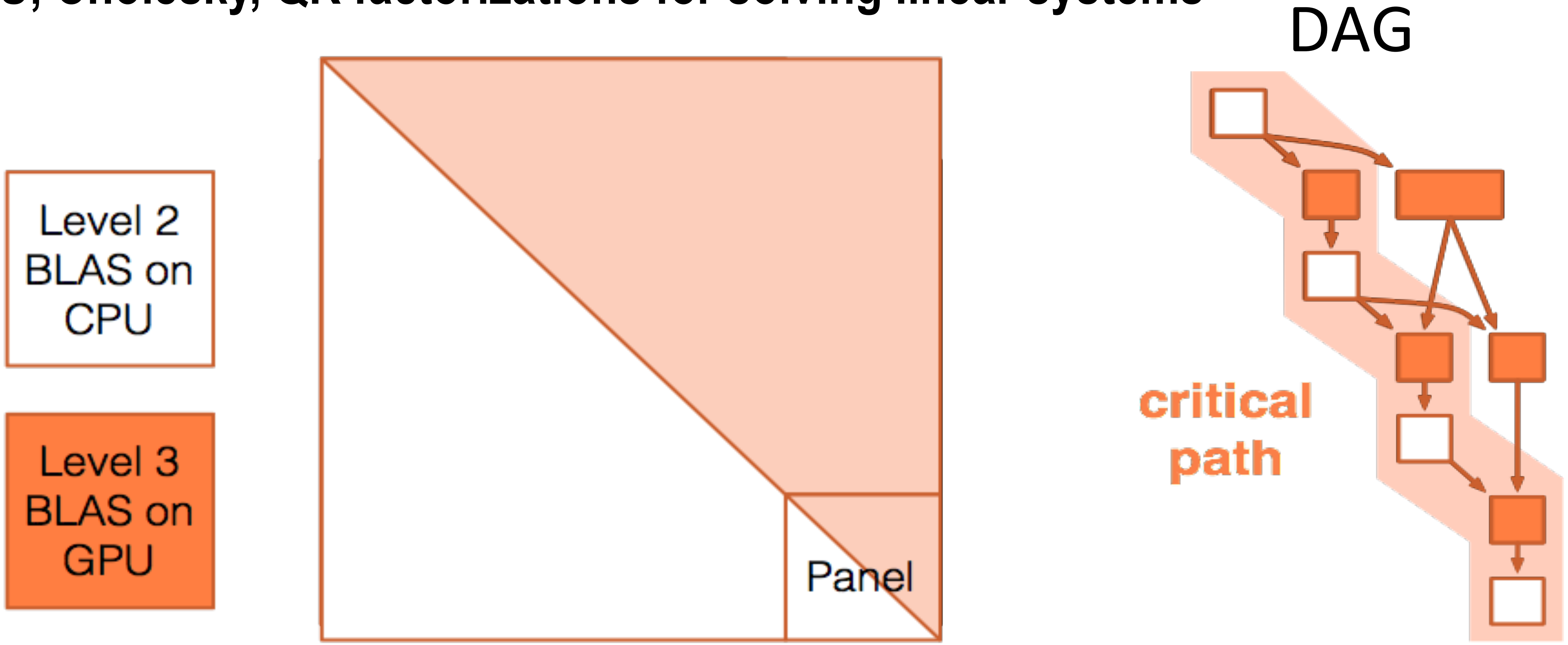
Code

Testers

Methodology

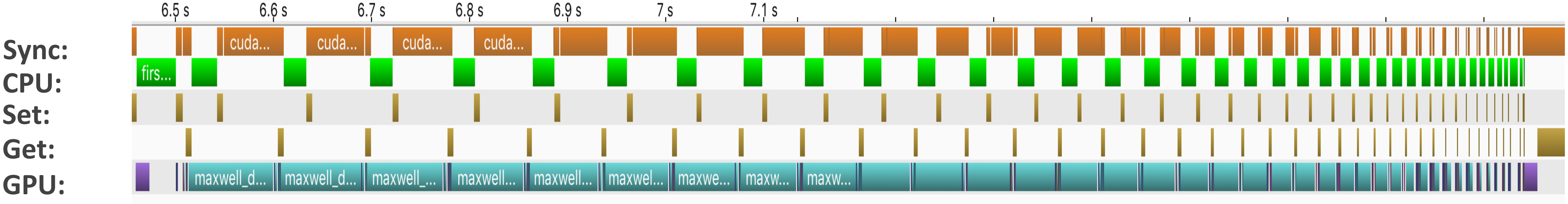
One-sided factorizations

- **LU, Cholesky, QR factorizations for solving linear systems**



Execution trace

- **Panels** on CPU **(green)** and **set/get communication** **(brown)** overlapped with **trailing matrix updates** **(teal)** on GPU
- Goal to keep GPU busy all the time; CPU may idle

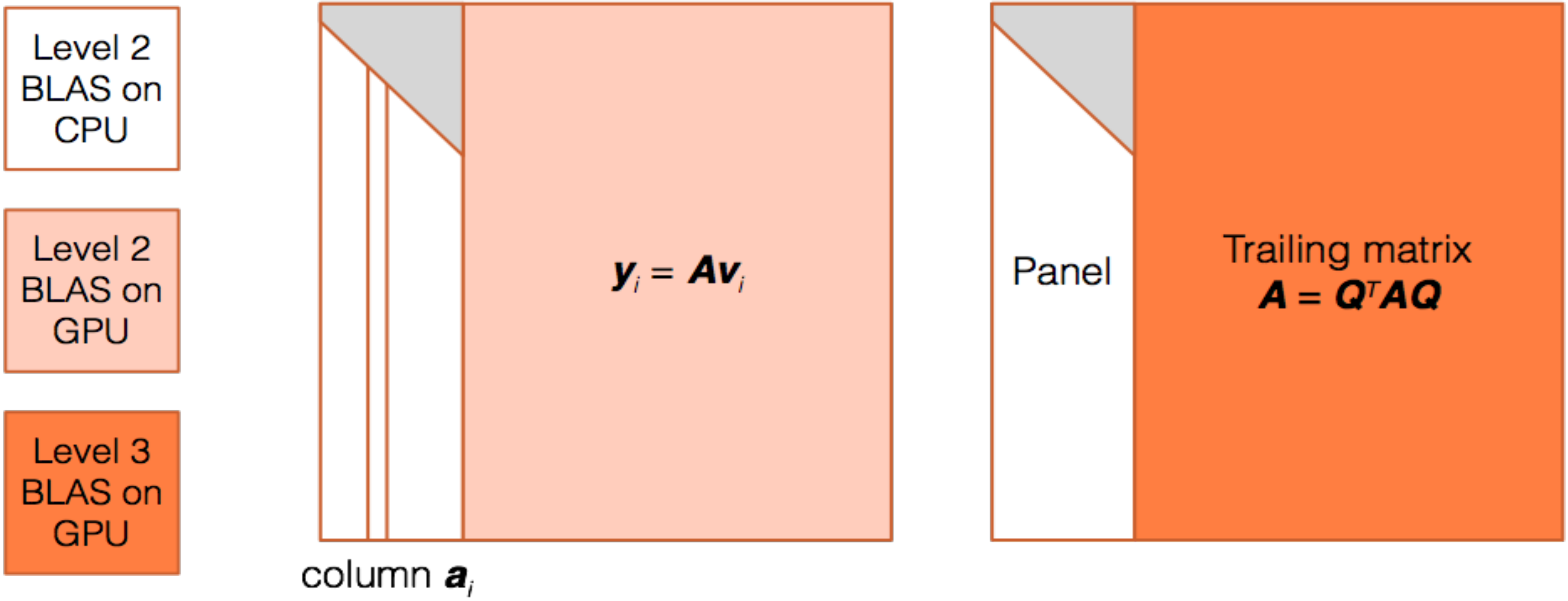


LU factorization (dgetrf), n = 20000
P100 GPU, 2 × 10-core 2.3 GHz Haswell

- Optimization: for LU, we transpose matrix on GPU so row-swaps are fast

Two-sided factorizations

- Hessenberg, tridiagonal, bidiagonal factorizations for eigenvalue and singular value problems



Numerical Linear Algebra (NLA) in Applications

- **For big NLA problems**
  (BLAS, convolutions, SVD, linear system solvers, etc.)

- **Adding in MAGMA application backends for small problems**

  - Machine learning / DNNs
  - Data mining / analytics
  - High-order FEM,
  - Graph analysis,
  - Neuroscience,
  - Astrophysics,
  - Quantum chemistry,
  - Signal processing, and more

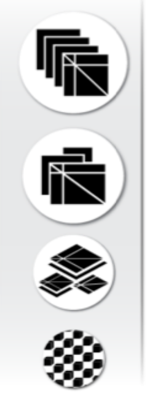
Large matrices

In contemporary libraries:

BLAS
LAPACK
ScaLAPACK

MAGMA (for GPUs)

Small matrices / tensors



Fixed-size batches

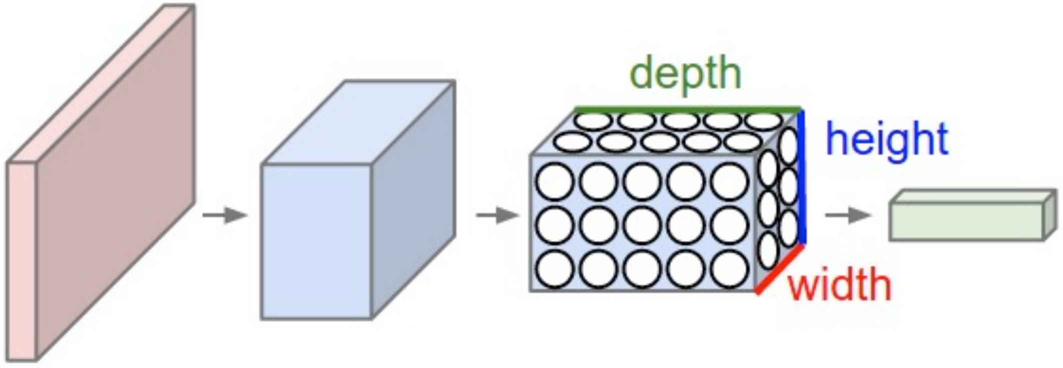
Variable-size batches

Dynamic batches

Tensors

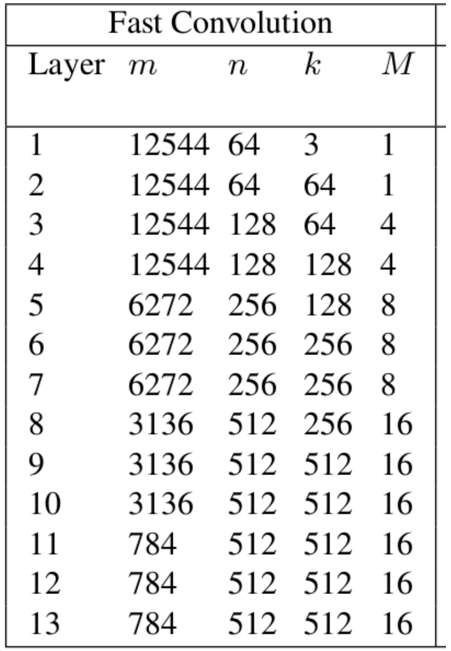
Accelerating NN – express with GEMMs of various sizes

Convolution Network (ConvNet) example



Require matrix-matrix products of various sizes, including batched GEMMs

- **Convolutions can be accelerated in various ways:**
  - **Unfold and GEMM**
  - **FFT**
  - **Winograd minimal filtering – reduction to batched GEMMs**



Fast Convolution

Layer	m	n	k	M
1	12544	64	3	1
2	12544	64	64	1
3	12544	128	64	4
4	12544	128	128	4
5	6272	256	128	8
6	6272	256	256	8
7	6272	256	256	8
8	3136	512	256	16
9	3136	512	512	16
10	3136	512	512	16
11	784	512	512	16
12	784	512	512	16
13	784	512	512	16

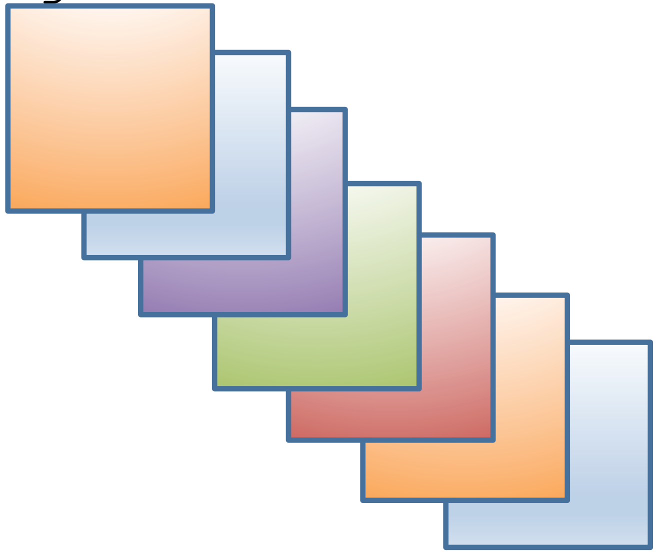
- **Use autotuning to handle complexity of tuning**

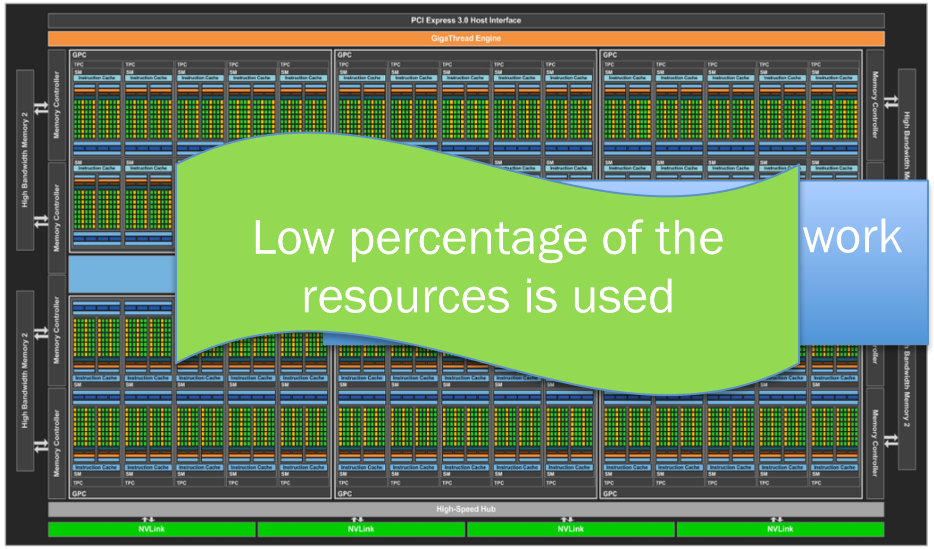
MAGMA Batched Computations

1. Non-batched computation

- **loop over the matrices one by one** and compute using multithread (note that, since matrices are of small sizes there is not enough work for all the cores). So we expect low performance as well as threads contention might also affect the performance

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





Low percentage of the resources is used

work

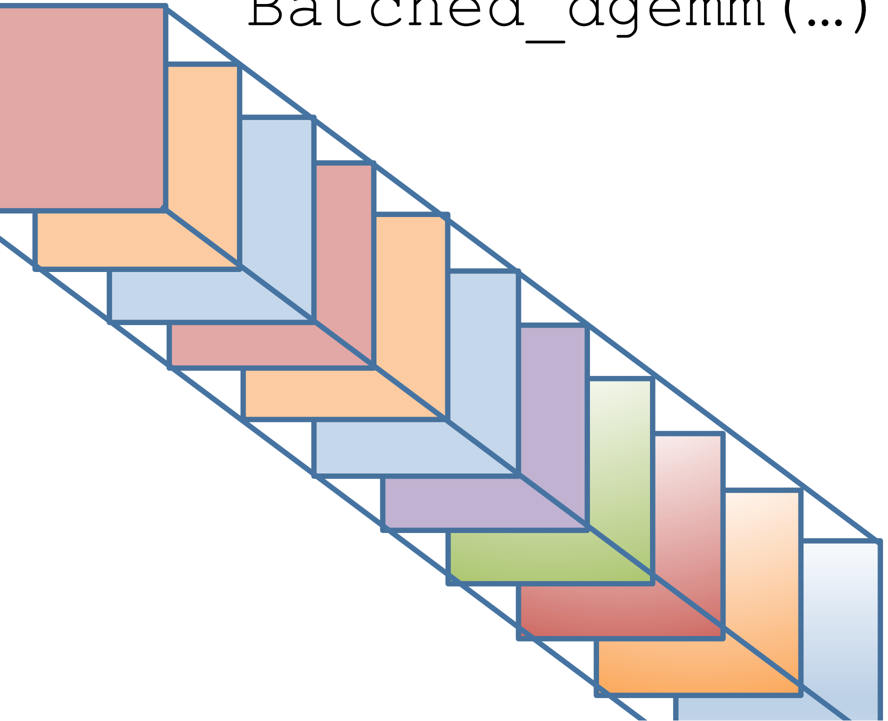
MAGMA Batched Computations

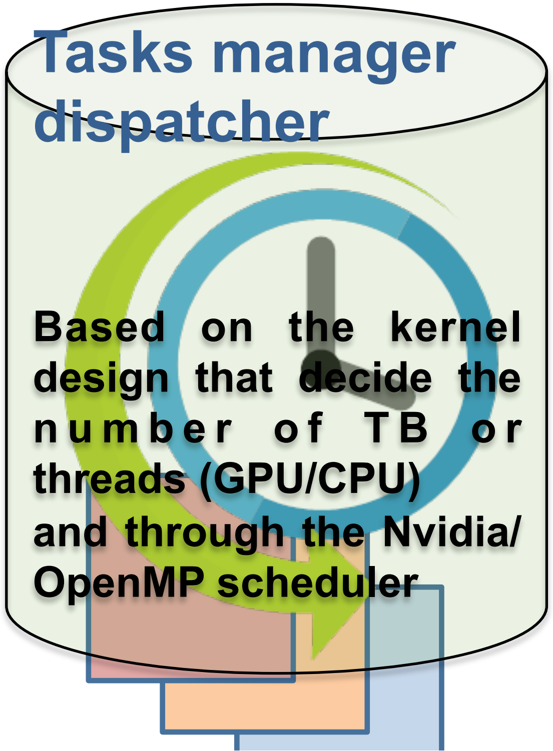
1. Batched computation

Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

- For very small matrices, assign a matrix/core (CPU) or per TB for GPU
- For medium size a matrix go to a team of cores (CPU) or many TB's (GPU)
- For large size switch to multithreads classical 1 matrix per round.

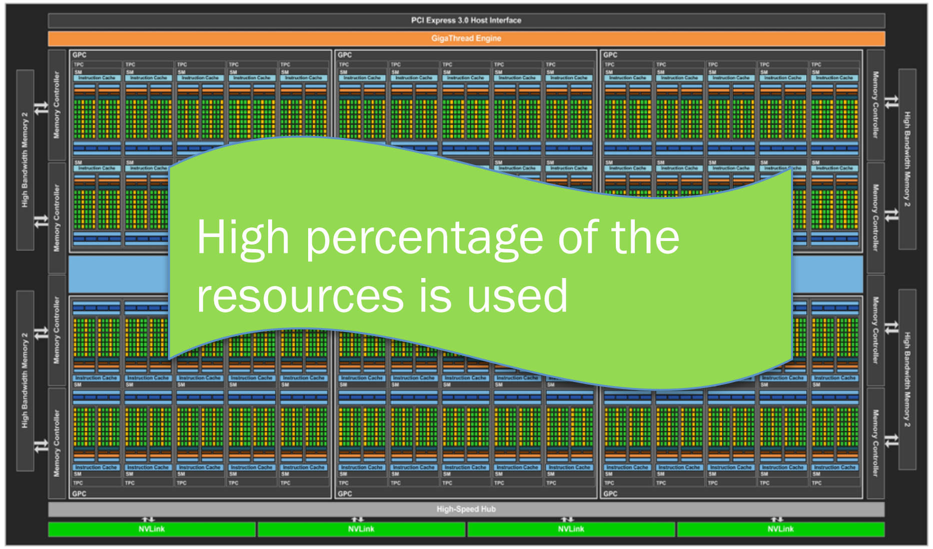
`Batched_dgemm(…)`





Tasks manager dispatcher

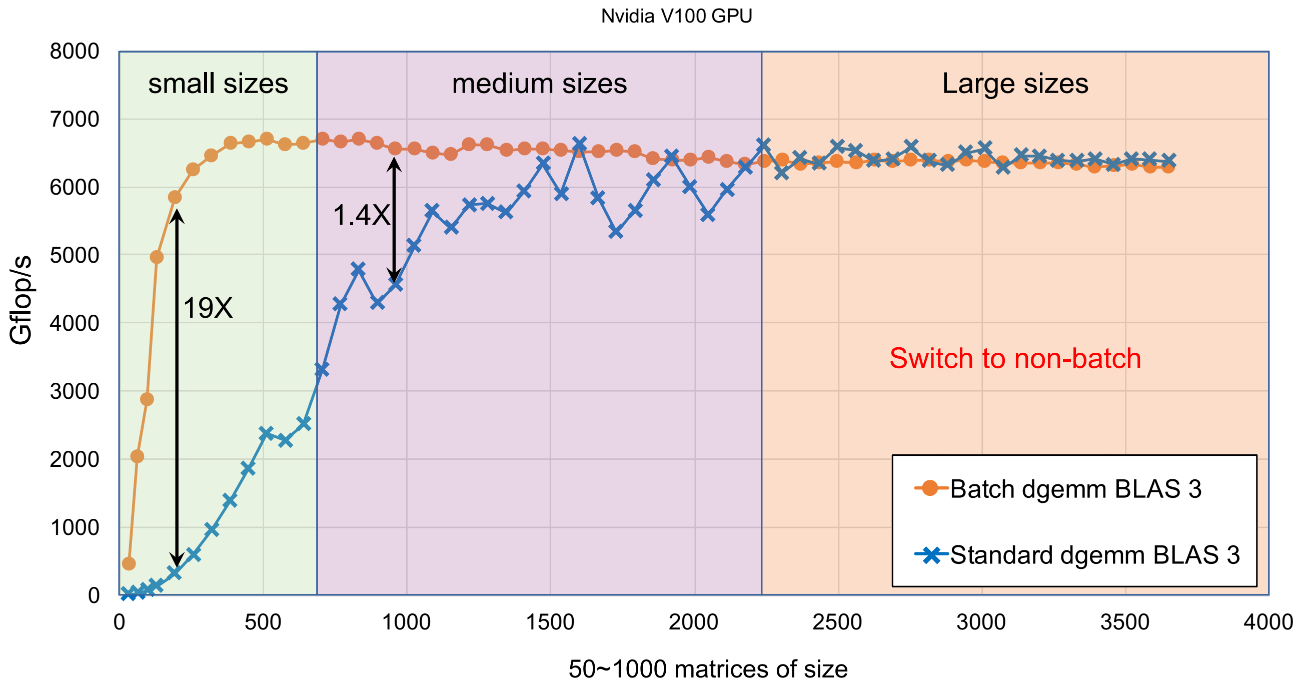
Based on the kernel design that decide the number of TB or threads (GPU/CPU) and through the Nvidia/ OpenMP scheduler



High percentage of the resources is used

How to implement fast batched DLA?

Problem sizes influence algorithms & optimization techniques

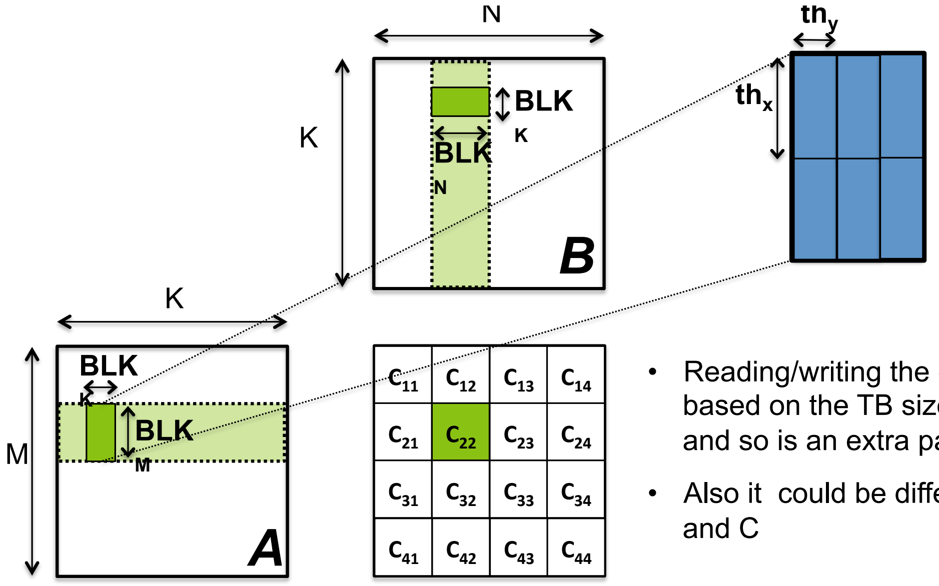


Matrix sizes (fixed) in the batch

Batch size 1,000 | Batch size 300 | Batch size 50

Kernels are designed various scenarios and parameterized for autotuning framework to find "best" performing kernels

Optimizing GEMM's: Kernel design


N
K
BLK
BLK_K
BLK_N
B
th_y
th_x
K
BLK
BLK_K
BLK_M
M
A
C_{11} C_{12} C_{13} C_{14}
C_{21} C_{22} C_{23} C_{24}
C_{31} C_{32} C_{33} C_{34}
C_{41} C_{42} C_{43} C_{44}

- Reading/writing the elements is based on the TB size (# threads) and so is an extra parameter.
- Also it could be different for A, B and C

Leveraging Half Precision in HPC on V100

MAGMA Mixed Precision algorithms

Idea: use lower precision to compute the expensive flops (LU $O(n^3)$) and then iteratively refine the solution in order to achieve the FP64 arithmetic

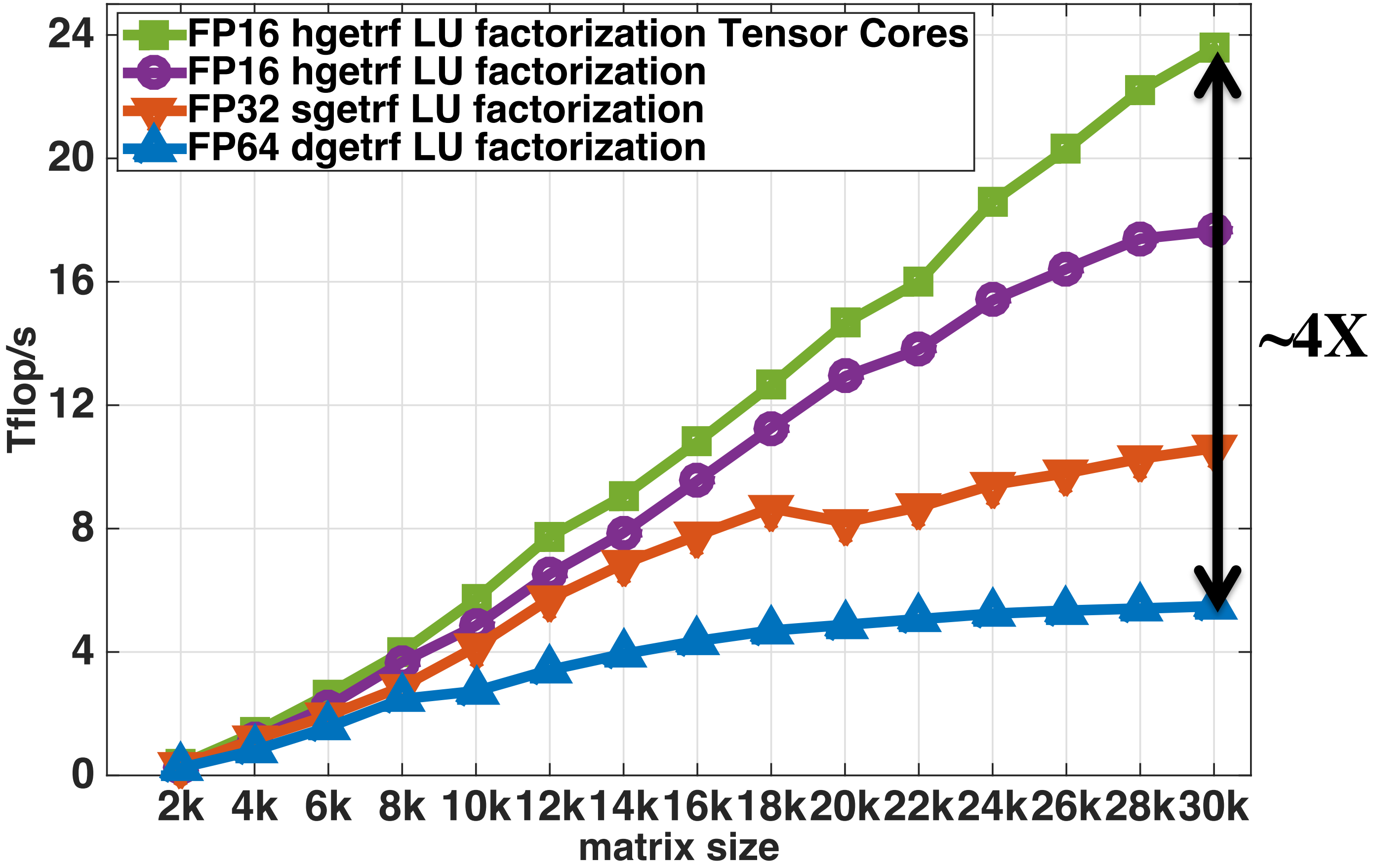
- Achieve higher performance → faster time to solution
- Reduce power consumption by decreasing the execution time → Energy Savings !!!

Reference:

1. Haidar, A., Wu, P., Tomov, S., Dongarra, J.
   Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers,
   ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ACM, Denver, Colorado, November 12-17, 2017.
2. Haidar, A., Tomov, S., Dongarra, J. , Higham, NJ,
   Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers,
   SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, November 11-16, 2018.

Leveraging Half Precision in HPC on V100 Motivation

Study of the LU factorization algorithm on Nvidia V100



- LU factorization is used to solve a linear system Ax=b

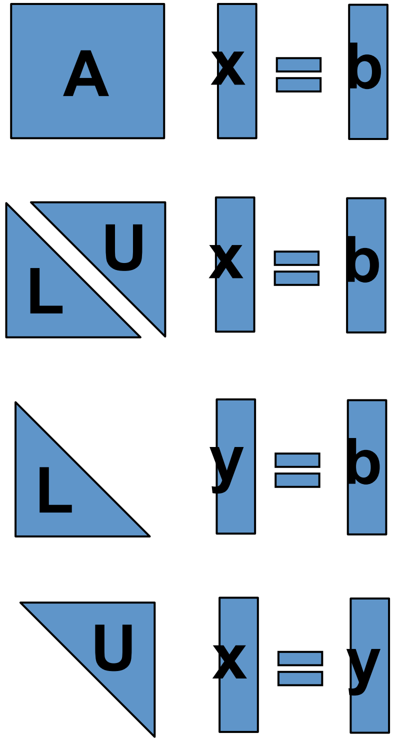
A x = b

LUx = b

Ly = b

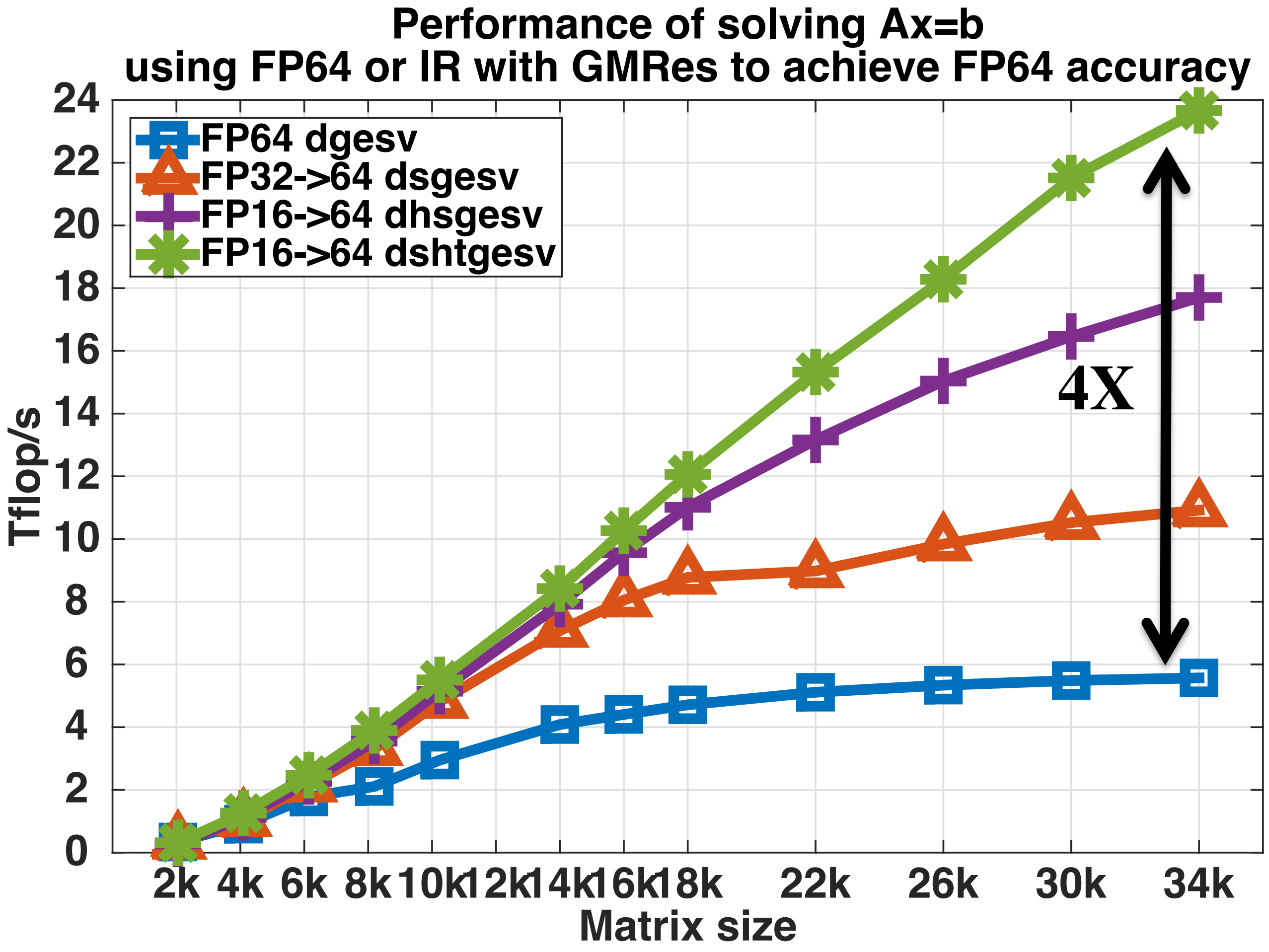
then

Ux = y



Leveraging Half Precision in HPC on V100

Performance Behavior



Flops = $2n^3/(3\ \text{time})$
meaning twice higher is twice faster

- solving Ax = b using FP64 LU
- solving Ax = b using FP32 LU and iterative refinement to achieve FP64 accuracy
- solving Ax = b using FP16 LU and iterative refinement to achieve FP64 accuracy
- solving Ax = b using FP16 Tensor Cores LU and iterative refinement to achieve FP64 accuracy

Examples / Exercises

- Every routine has an example / benchmark
- Examples / benchmarks are in directory “testing”

```
> cd testing
```

- To generate a particular test, e.g., matrix-matrix product

```
> make testing_dgemm
```

Examples / Exercises

- Solving a linear system of equations

  Ax = b

```
#include "magma_v2.h”
#include "magma_lapack.h”
…

magma_init();
…
double *hA, *hB;                                  // A is a typical array on the CPU
magma_dmalloc_pinned(&hA,  lda * N  );            // A can be allocated in pinned memory
magma_dmalloc_cpu(     &hB, nrhs* M );
…
init_matrix(opts, M, N, hA, lda);
…
magma_dgetrf( M, N, hA, lda, ipiv, &info);        // LU factorization (using CPU+GPU)
lapackf77_dgetrs( MagmaNoTransStr, &N, &nrhs,     // Solve on CPU with LAPACK
                  hA, &lda, ipiv, hB, &ldb, &info );
```

Alternatively, there is a direct MAGMA function to solve:
  see testing_dgesv.cpp and testing_dgetrf.cpp

Examples / Exercises

- Solving an eigenvalue problem
  Ax = λx

```
#include "magma_v2.h”
#include "magma_lapack.h”
…

magma_init();
…
double *hA;                                             // A is a typical array on the CPU
magma_dmalloc_pinned( &hA,  lda * N);                    // A can be allocated in pinned memory
…

magma_dsyevd( opts.jobz, opts.uplo,
              N, hA, lda, w1,
              h_work, lwork,
              iwork, liwork, &info );
```

see testing_dsyevd.cpp

Examples / Exercises

- Writing a hybrid algorithm

  Develop a hybrid CPU-GPU algorithm for this algorithm
  (Cholesky QR for a “tall-and-skinny” matrix A)

Algorithm	MATLAB	Hybrid CPU-GPU algorithm	

$G = A^T A$	G = A’*A;	cublasDgemm(...);
		magma_dgetvector(...);
			// Send G from GPU to CPU memory,
			// e.g, hG

$G = L\, L^T$	U = chol(G);	dpotrf_(...)
		magma_dsetvector(...);
			// LAPACK on CPU to do Cholesky on hG
			// send result back to the GPU

$Q = A\,(L^T)^{-1}$	Q = A * inv(U)	magma_dtrsm(...);	// Triangular matrix solve on the GPU

- First check that indeed A = QR, where Q is orthogonal ($Q^TQ = I$)
- Compare your solution to the one implemented in dgegqr_gpu.cpp

CUDA references

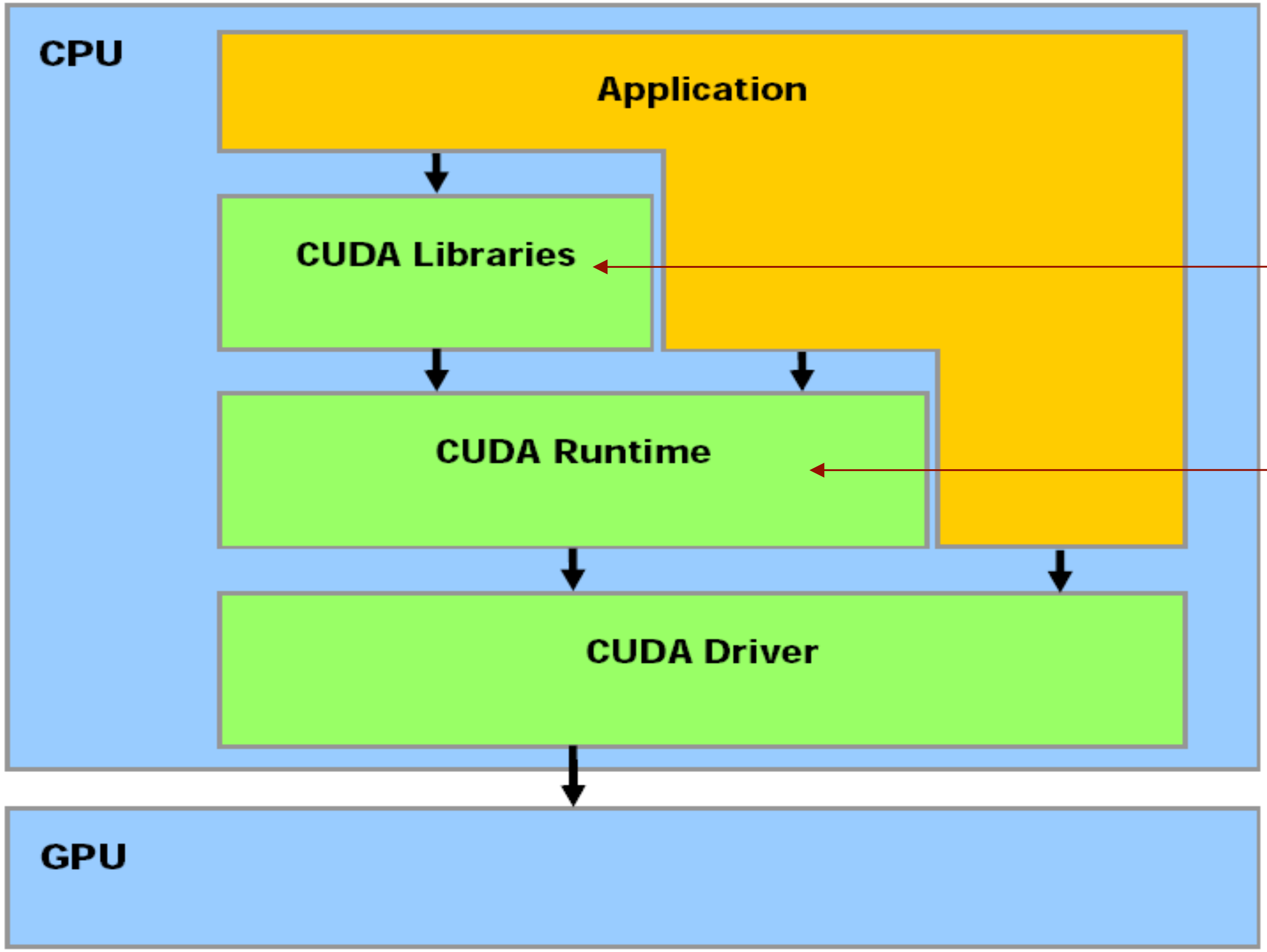
- NVIDIA CUDA Programming Guide

  https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

- CUDA Samples

  Typically installed with CUDA, e.g., in /usr/local/cuda/samples

CUDA Software Stack



CUBLAS, CUFFT, MAGMA, …

C like API

(Source: NVIDIA CUDA Programming Guide)

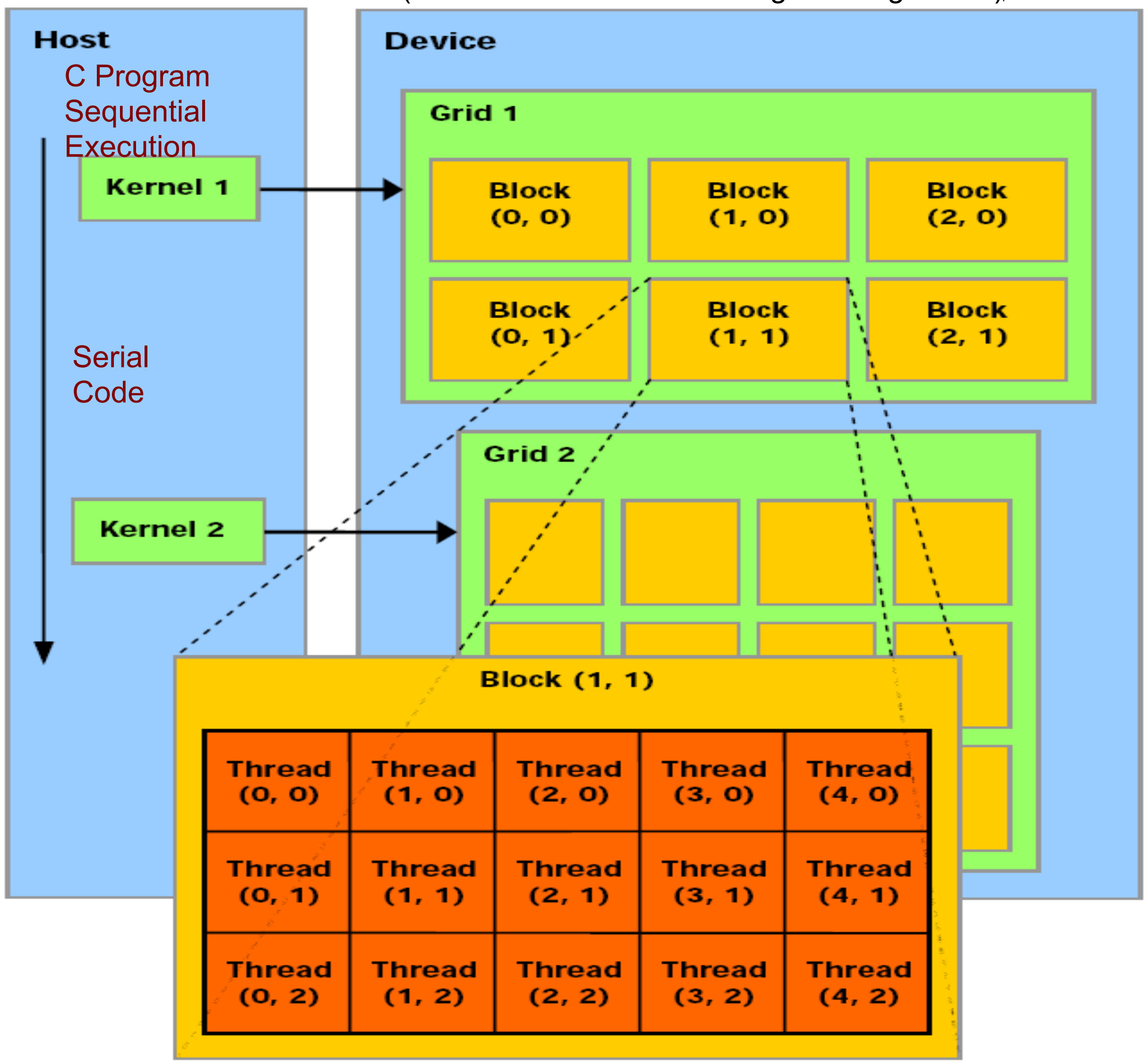
CUDA Programming Model

- Grid of thread blocks
  (blocks of the same dimension, grouped together to execute the same kernel)
- Thread block
  (a batch of threads with fast shared memory executes a kernel)
- Sequential code launches asynchronously GPU kernels

```
// set the grid and thread configuration
Dim3 dimGrid(2,3);
Dim3 dimTBlock(3,5);

// Launch the device computation
MatVec<<<dimGrid, dimTBlock>>>( . . . );
```

```
__global__ void MatVec( . . . ) {
  // Block index
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Thread index
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  . . .
}
```

(Source: NVIDIA CUDA Programming Guide)



Block (0, 0)	Block (1, 0)	Block (2, 0)
Block (0, 1)	Block (1, 1)	Block (2, 1)

Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)

Hello World!

Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

```
int main(void) {
   printf("Hello World!\n");
   return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$
```

Hello World!
with Device Code

Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

```
__global__ void mykernel(void) {
    printf("Hello World!\n");
}

int main(void) {
      mykernel<<<1,1>>>();
cudaDeviceSynchronize();
      return 0;
}
```

Hello World! with Device Code

```
__global__ void mykernel(void) {
    printf("Hello World from block %d, thread %d!\n”,
            blockIdx.x, threadIdx.x);
}

int main(void) {
      mykernel<<<10,10>>>();
      cudaDeviceSynchronize();
      return 0;
}
```

Parallel Programming in CUDA C/C++

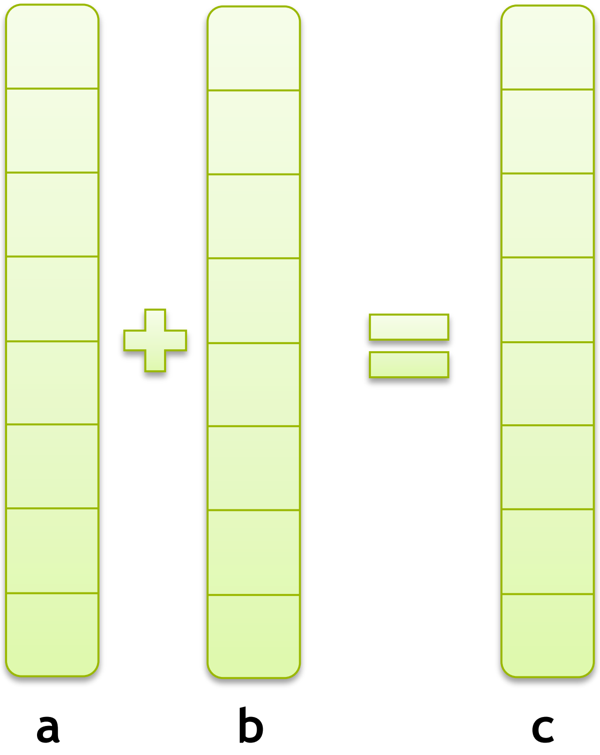
Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

- But wait… GPU computing is about massive parallelism!

- We need a more interesting example…

- We'll start by adding two integers and build up to vector addition

…



Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

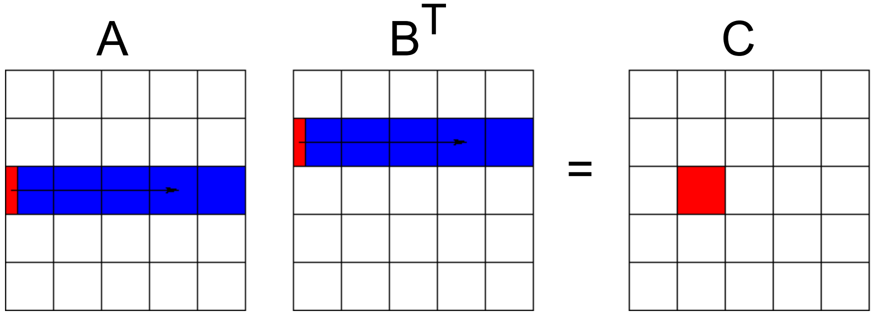
```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

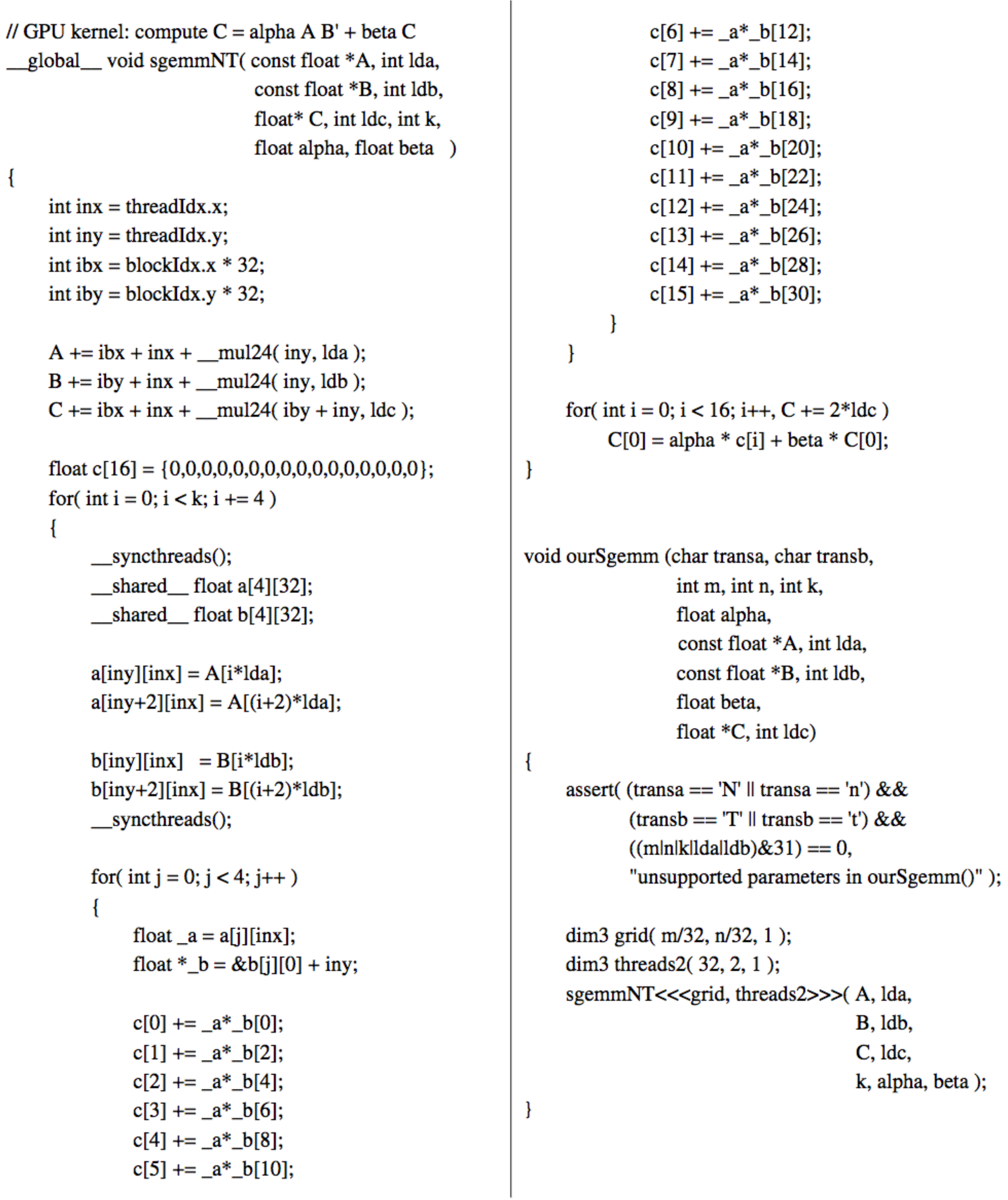
```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

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

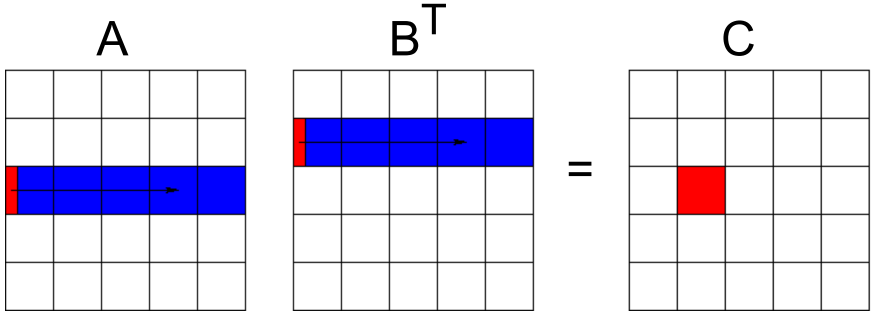


```
// GPU kernel: compute C = alpha A B' + beta C
__global__ void sgemmNT( const float *A, int lda,
                         const float *B, int ldb,
                         float* C, int ldc, int k,
                         float alpha, float beta   )
{
    int inx = threadIdx.x;
    int iny = threadIdx.y;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

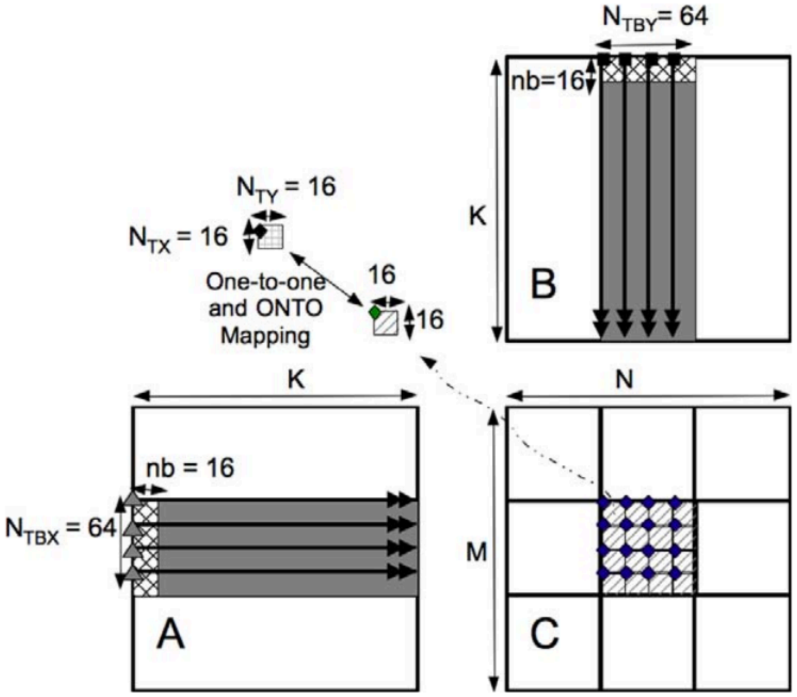
GEMM in MAGMA

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)

Implementing a DNN Framework

Daniel Nichols

Before You Start Writing Code

- Why?
  - High Performance
    - C++, C++/Python
    - Hardware accelerators
  - Portability
    - Python, C++/Python
    - Containers
  - Minor Feature
    - Extend existing open source framework
- Design
  - Write example use cases
  - Design with modularity in mind

Goals

Easy to use tensors and file I/O:

```
Tensor dataSet = io::from_csv(“data.csv”);
```

Fast tensor math backend:

```
Tensor A = …, B = …;
Tensor C = math::matmul(A, B);
```

Compute graph:

```
Op weights = …, input = …, bias = …;
Op outputOp = op::add(op::matmul(weights,
input), bias);
Tensor output = outputOp->evaluate();
```

Gradient computation:

```
GradientTable grads(outputOp);
grads.grad(outputOp, weights); /*wrt weights*/
```

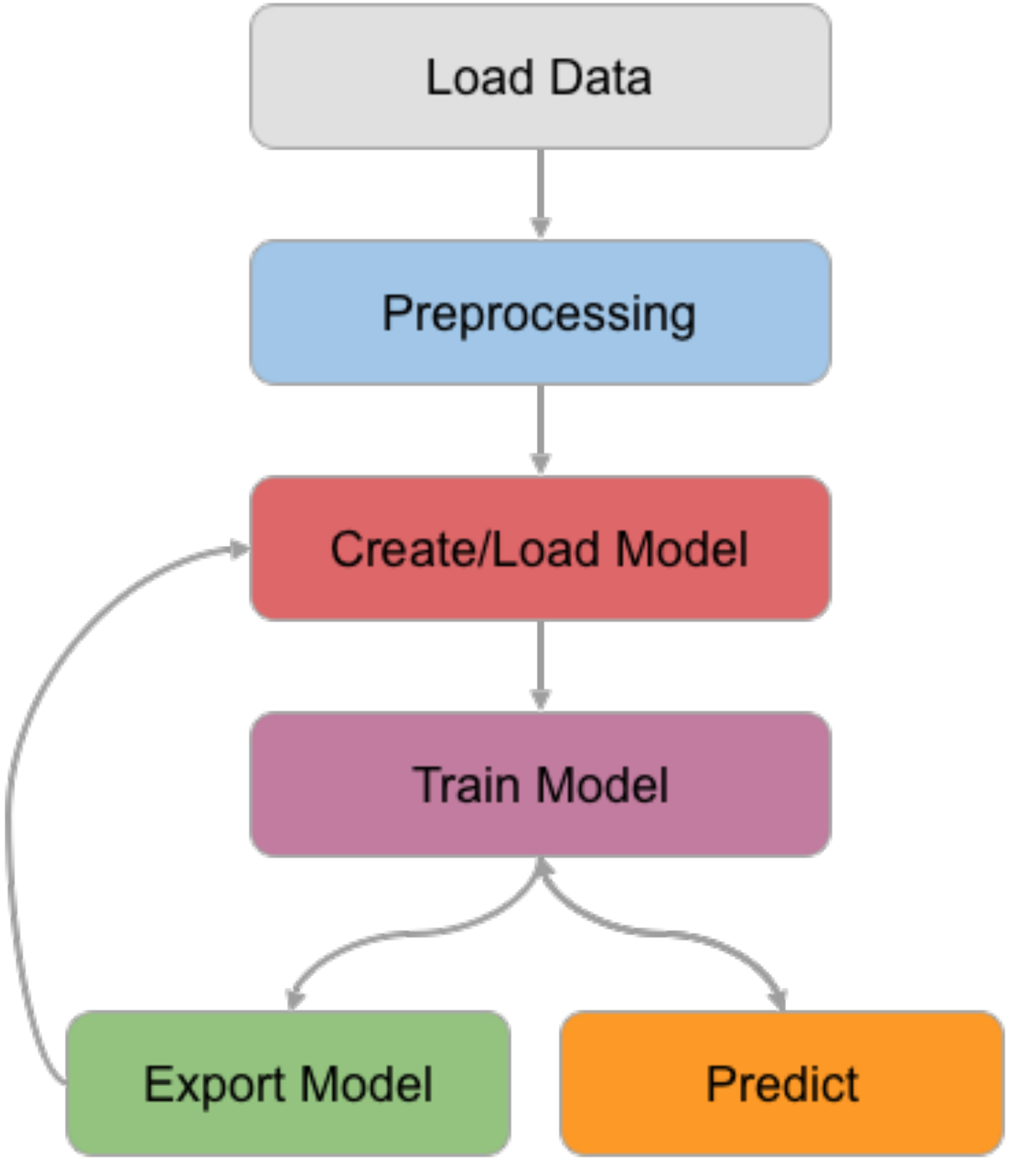
Optimize compute graphs:

```
SGDOptimizer optim(0.01);
optim.minimize(outputOp, weights);
```

Model construction:

```
Layer fc0 = layer::dense(512);
Layer fc1 = layer::dense(fc0, 128);
Layer fc2 = layer::dense(fc1, 10);
Model network({fc0, fc1, fc2}, SGD, …);

network.fit(trainDataset);
```

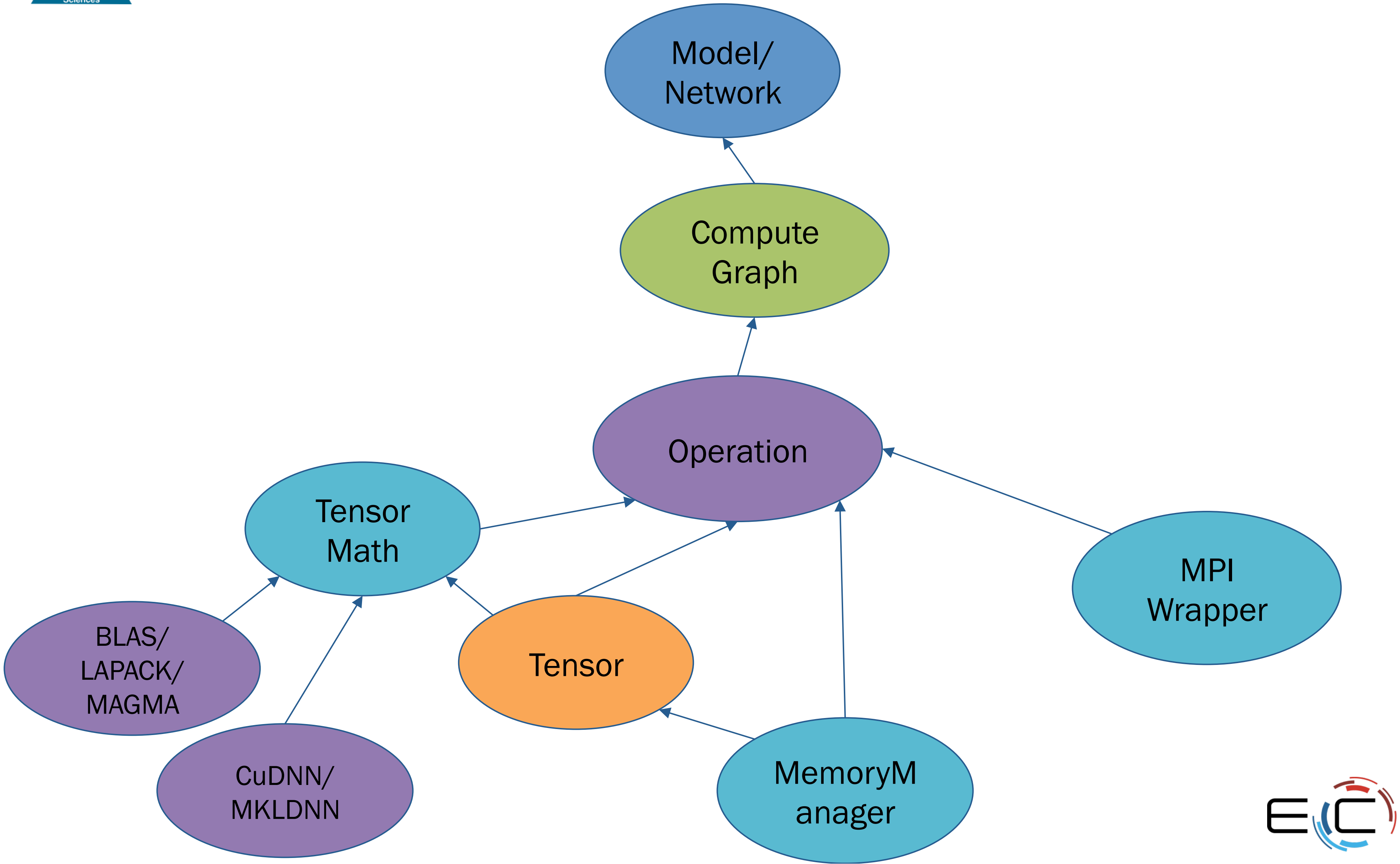
DNN Workflow



Software Structure of a DNN Framework

- **Core**
  - **Systems**
    - **Memory (CPU, GPU, Hybrid)**
    - **Communication (MPI, OpenMP)**
  - **Math**
    - **Tensor data structures and operations**
    - **Blas/Lapack, Magma**
    - **DNN accelerator (CuDNN, hipDNN, OneDNN)**
- **Frontend**
  - **Compute Graph**
  - **Optimizers**
  - **Models**

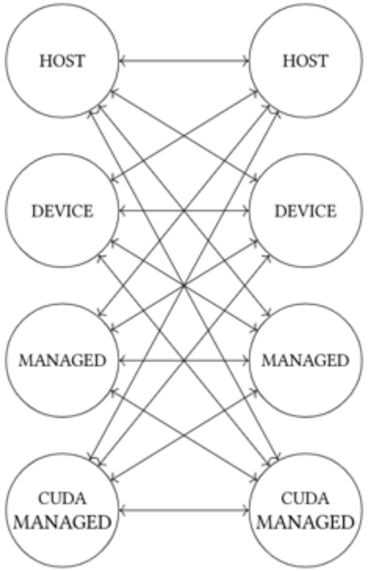
MagmaDNN’s Structure



Memory Management

Goals

- **Abstraction**
  - **Provide basic memory operations for supported hardware**
    - **copy, allocate, free, put, get**
  - **Support inter-device operations for all device combinations**
- **Virtualization**
  - **Emulate capabilities not supported by hardware**
    - **synchronized memory, async prefetching**
- **Optimized**
  - **Parameterized for auto-tuning**



Memory Management

```
class MemoryManager {
    MemoryManager(size_t n) {...}
    ~MemoryManager() {...}

     void copy(MemoryManager const& m){.}
    void swap(MemoryManager& m){...}

    template <class T>
    void set(T *arr) {...}

    template <class T>
    T* get() {...}

    void setDevice(device_t dev) {...}
    device_t getDevice() {...}
};
```

Interface

- **Simple user exposed interface**
- **Modern C++ style code**
  - **Pointers are necessary for Memory Management**
- **Can use helper classes to support combinatorial operations**

Memory Manager

```
/* (1) */
MemoryManager<T>(size_t n): data_(new T[n]) {}

/* (2) */
MemoryManager(size_t n, data_t type) {
     switch (type) {
          case FLOAT:
                this->data_ = new float[n];
          …}
}

/* (3) */
MemoryManager(size_t n, data_t type) {
    this->data_ =
     (void*) new uint8_t[n * numBytes(type)];
}
```

Three ways to store typed data

- **(1) Generics/Templates**
  - **New class for every type**
  - **Types must be known at compile time**
- **(2) Switch on every type**
  - **If data_t is enumerated, then jump tables are used**
- **(3)** Binary Large **Object (BLOB)**
  - **Store bytes and type information separately**

Memory Manager

```
auto cudaPtr = [](size_t s) { void *p;
 cudaMalloc((void **) &p, s); return p; };

auto deleter = [](float *p) { cudaFree(p); };

std::unique_ptr<float[], decltype(deleter)>((float
 *) cudaPtr( len*sizeof(float) ), deleter);
```

Make use of language features

- **std::unique_ptr<T>**
  - **Since c++11**
  - **Frees when object leaves scope**
  - **Implements proper Copy/ Swap/Move semantics**
- **std::iterator**
  - **Allows the use of std functions**
  - **Can use pointers as iterators**

Memory Management

Optimizations

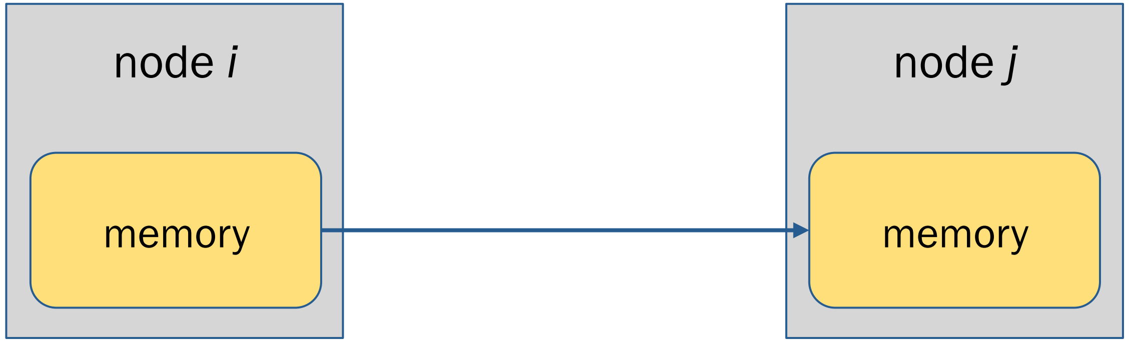
- **Asynchronous Operations**
  - **Move next set of memory into GPU while other operations are still completing**
- **Tunable parameters**
  - **Block Size**
  - **Buffering**
  - **Can be tuned to hardware at library compile time**

Communication

Communication

- **Memory**
  - **Move memory between devices/nodes seamlessly**
- **Math Operations**
  - **Allreduce, Reduce**
- **Implementation**
  - **MPI Wrappers**
  - **Hidden implementation**
  - **Well optimized**
  - **Error checking**

Communication



`MemoryManager::copy(...)`

node *i*

```
MPI_Send(data_, this->size(), …);
```

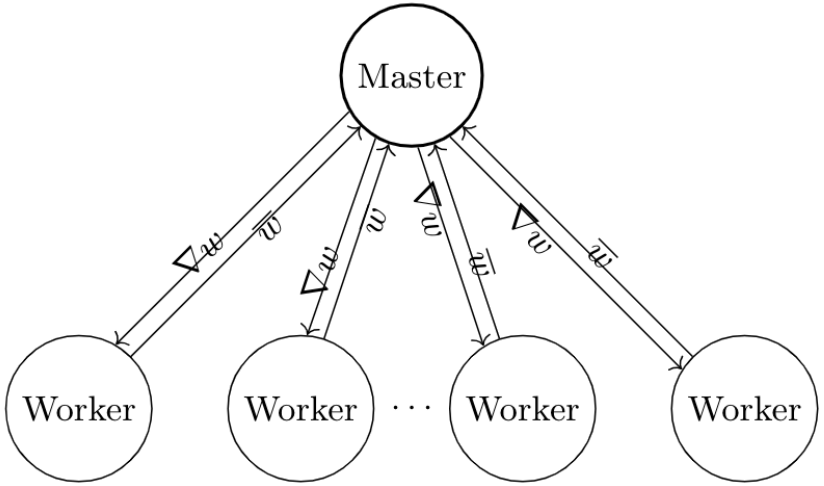
node *j*

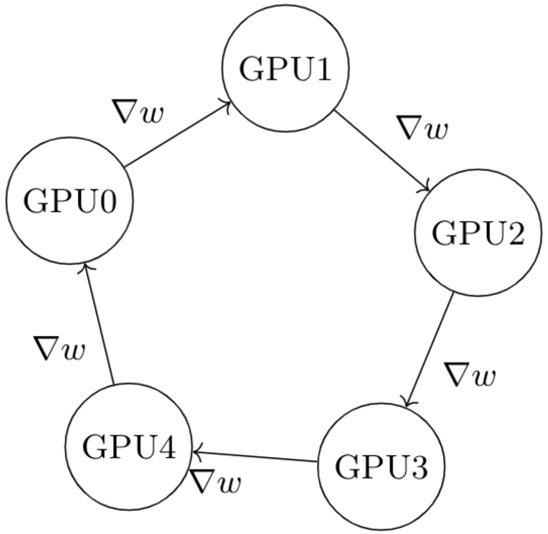
```
MPI_Recv(data_, this->size(), …);
```

Abstract Send/Receive Paradigm

- **Make communication details implicit**
- **Allow explicit customization**
- **::asyncCopy(...)**
  - **MPI_ISend(...)**
  - **MPI_IRecv(...)**
- **Check for errors!**

Communication





Important Collectives

- AllReduce(*), Reduce, Bcast, AllGather, ReduceScatter
- Use most optimal
  - MPI, Cuda-aware MPI
  - Horovod(*)
  - NCCL1 (intra-node), NCCL2 (inter-node)
- Use RDMA and GPU Direct where applicable

Communication

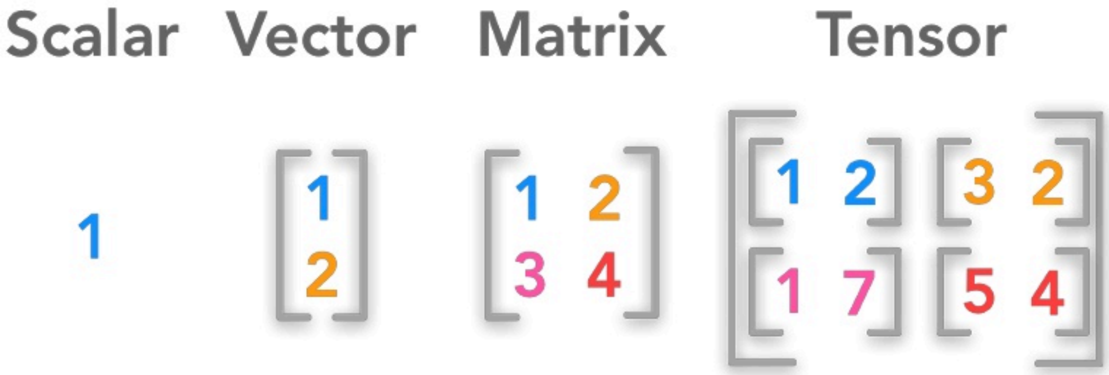
Implementation

- Wrappers for communication primitives
  - Send, Recv, Reduce, AllReduce, AllGather, Bcast, ReduceScatter
  - Allows easy change of backend (MPI, Horovod, etc…)
- Optimized
  - Tune MPI to system hardware
  - Use state-of-the-art techniques
    - Toroidal-Reduce, Pipelined Communication, Asynchronous Updates
- Error Check
  - Handle and recover from node failures
  - MPI is not fault tolerant!

Tensors

Data Structure

- Multi-Dimensional Wrapper around MemoryManager
- Conceptually *N*-dimensional, but should support 1-8 dimensions
- Equivalent to NumPy's Ndarray
- Support multiple data layouts



Tensors

```
class Tensor {
  public:
     Tensor(vector<size_t> const& shape) {.}
    Tensor(initializer_list<size_t> …) {.}

    template <class T>
    void set(vector<size_t> const& idx, T
val) {...}
    template <class T>
    T get(vector<size_t> const& idx) {...}
  private:
    MemoryManager memoryManager_;
    data_t dtype_;

    size_t getFlattenedIndex(vector<size_t>
const& idx) { … }
};
```

Implementation

- **Use memory manager to store data**
- **Tensor class provides structure to linear memory**
- **Support several data layouts**
  - **Stride from leading dimension**
  - **Stride from trailing dimension**
  - **Tiled**
- **Shallow vs Deep Copies**

Tensor Math

Design

- **Provide commonly used math operations for tensors**
- **Support all hardware**
- **Use modular backend**

Implementation

- **Instantiate versions of functions based on available libraries and hardware**
- **Clean wrappers for underlying libraries (BLAS, LAPACK, etc…)**

Tensor Math

```
template <device_t dev>
Tensor matmul<dev>(Tensor const& A, Tensor const& B);

template <>
Tensor matmul<CPU>(Tensor const& A, Tensor const& B) {
    if (A.dtype() == FLOAT_T) {
        cblas_sgemm(...);
    } else …
}

template <>
Tensor matmul<GPU>(Tensor const& A, Tensor const& B) {
    if (A.dtype() == FLOAT_T) {
        magma_sgemm(...);
    } else ...
}
```

Instantiate by Hardware/Accelerator

- **If C++, use templates**
- **Only compiles if available**

Modular Design

- **Multiple Backends**
- **Easy to switch drivers**
  - **MagmaBLAS, CuBLAS, CBLAS, MKL BLAS**
  - **oneDNN, hipDNN, cuDNN**

Common Tensor Math Functions

Matmul - `math::matmul(A, B, C)`

Dot - `math::dot(A, B)` **(gemv or gemm)**

Reduce - `math::reduce(A, Operation, axis)`

- Min, Max
- Sum
- Product
- ArgMin, ArgMax

Elementwise Unary Func - `math::unary(A, Op)`

- Negate
- Invert (element-wise)
- Square
- Softmax
- ReLU, Leaky ReLU
- Tanh
- Sigmoid

Elementwise Binary Func - `math::binary(A, B, Op)`

- Add
- Product (hadamard)
- Cross Entropy

Norm - `math::norm(A, type)`

Convolutions -

- `math::conv1d(data, filter, params…)`
- `math::conv2d(data, filter, params…)`
- `math::conv3d(data, filter, params…)`

Pooling - `math::pooling2d(data, AVG/MAX, …)`

Tensor Math

```
template <>
Tensor matmul<CPU>(Tensor const& A, Tensor const& B) {
    if (A.dtype() == FLOAT_T
          && isTallSkinny(A)
          && A.isCuda()) {

        magma_sgemm(...);

    } else if (A.dtype == DOUBLE_T
           && A.isCPU()
           && A.dataLayout() == TILE_LAYOUT) {

        slate::gemm(...);

    } else …
}
```

Optimizations

- **Use best backends**
  - **Tune at compile time**
  - **Modular design**
  - **Choose based on data size/ type**
- **Minimize call path to core function**
- **Use table or state machine to give most optimal function**

Tensor Math

```
cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t                        handle,
    const void                          *alpha,
    const cudnnTensorDescriptor_t        xDesc,
    const void                          *x,
    const cudnnTensorDescriptor_t        dyDesc,
    const void                          *dy,
    const cudnnConvolutionDescriptor_t   convDesc,
    cudnnConvolutionBwdFilterAlgo_t      algo,
    void                                *workSpace,
    size_t                         workSpaceSizeInBytes,
    const void                          *beta,
    const cudnnFilterDescriptor_t        dwDesc,
    void                                *dw
)
```

DNN Libraries

- **CuDNN, OneDNN, hipDNN**
- **Best Use**
  - **Only call core routines in training**
  - **Preallocate as much memory as possible**
  - **Error Check!**
- **Profile, profile, profile...**

Tensor Math

```
cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t                        handle,
    const void                          *alpha,
    const cudnnTensorDescriptor_t        xDesc,
    const void                          *x,
    const cudnnTensorDescriptor_t        dyDesc,
    const void                          *dy,
    const cudnnConvolutionDescriptor_t   convDesc,
    cudnnConvolutionBwdFilterAlgo_t      algo,
    void                                *workSpace,
    size_t                         workSpaceSizeInBytes,
    const void                          *beta,
    const cudnnFilterDescriptor_t        dwDesc,
    void                                *dw
)
```

```
cudnnStatus_t cudnnCreate(cudnnHandle_t *handle)
```

DNN Libraries

- **CuDNN, OneDNN, hipDNN**
- **Best Use**
  - **Only call core routines in training**
  - **Preallocate as much memory as possible**
  - **Error Check!**
- **Profile, profile, profile...**

Tensor Math

```
cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t                      handle,
    const void                        *alpha,
    const cudnnTensorDescriptor_t      xDesc,
    const void                        *x,
    const cudnnTensorDescriptor_t      dyDesc,
    const void                        *dy,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionBwdFilterAlgo_t    algo,
    void                              *workSpace,
    size_t                       workSpaceSizeInBytes,
    const void                        *beta,
    const cudnnFilterDescriptor_t      dwDesc,
    void                              *dw
)
```

```
cudnnStatus_t cudnnCreate(cudnnHandle_t *handle)
```

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

  - **Only call core routines in**

    as possible
  - **Error Check!**
- **Profile, profile, profile...**

Tensor Math

```
cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t                           handle,
    const void                             *alpha,
    const cudnnTensorDescriptor_t           xDesc,
    const void                             *x,
    const cudnnTensorDescriptor_t           dyDesc,
    const void                             *dy,
    const cudnnConvolutionDescriptor_t      convDesc,
    cudnnConvolutionBwdFilterAlgo_t         algo,
    void                                   *workSpace,
    size_t                             workSpaceSizeInBytes,
    const void                             *beta,
    const cudnnFilterDescriptor_t           dwDesc,
    void                                   *dw
)
```

```
cudnnStatus_t cudnnCreate(cudnnHandle_t *handle)
```

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

- Only call core routines in

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

as possible

```
cudnnStatus_t cudnnCreateConvolutionDescr
    cudnnConvolutionDescriptor_t *convDes
```

-

r Math

```
cudnnStatus_t
cudnnGetConvolutionBackwardDataAlgorithm_v7(
    cudnnHandle_t                              handle,
    const cudnnFilterDescriptor_t              wDesc,
    const cudnnTensorDescriptor_t              dyDesc,
    const cudnnConvolutionDescriptor_t       convDesc,
    const cudnnTensorDescriptor_t              dxDesc,
    const int                      requestedAlgoCount,
    int                            *returnedAlgoCount,
    cudnnConvolutionBwdDataAlgoPerf_t
*perfResults)
```

```
    const cudnnTensorDescriptor_           yDesc,
    const void                             y,
    const cudnnConvolutionDescriptor_      convDesc,
    cudnnConvolutionBwdFilterAlgo_t       algo,
    void                                  *workSpace,
    size_t                           workSpaceSizeInBytes,
    const void                            *beta,
    const cudnnFilterDescriptor_t          dwDesc,
    void                                  *dw
)
```

```
nnStatus_t cudnnCreate(cudnnHandle_t *handle)
```

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

- Only call core routines in

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

as possible

```
cudnnStatus_t cudnnCreateConvolutionDescr
    cudnnConvolutionDescriptor_t *convDes
```

-

r Math

```
cudnnStatus_t
cudnnGetConvolutionBackwardDataAlgorithm_v7(
    cudnnHandle_t                          handle,
    const cudnnFilterDescriptor_t          wDesc,
    const cudnnTensorDescriptor_t          dyDesc,
    const cudnnConvolutionDescriptor_t   convDesc,
    const cudnnTensorDescriptor_t          dxDesc,
    const int                    requestedAlgoCount,
    int                          *returnedAlgoCount,
    cudnnConvolutionBwdDataAlgoPerf_t
*perfResults)
```

```
    const void
    const cudnnTensorDescriptor_            Desc,
    const void                              y,
    const cudnnConvolutionDescriptor_      onvDesc,
    cudnnConvolutionBwdFilterAlgo_t       algo,
    void                                *workSpace,
    size_t                          workSpaceSizeInBytes,
    const void                          *beta,
    const cudnnFilterDescriptor_t         dwDesc,
    void                                *dw
)
```

```
nnStatus_t cudnnCreate(cudnnHandle_t *handle)
```

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

- **Only call core routines in**

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

```
cudnnStatus_t
cudnnGetConvolutionBackwardDataWorkspaceSiz
    cudnnHandle_t                          handle
    const cudnnFilterDescriptor_t          wDesc
    const cudnnTensorDescriptor_t          dyDes
    const cudnnConvolutionDescriptor_t convD
    const cudnnTensorDescriptor_t        dxDesc,
    cudnnConvolutionBwdDataAlgo_t        algo,
    size_t                            *sizeInByte
```

Math

```
cudnnStatus_t
cudnnGetConvolutionBackwardDataAlgorithm_v7(
    cudnnHandle_t                          handle,
    const cudnnFilterDescriptor_t          wDesc,
    const cudnnTensorDescriptor_t          dyDesc,
    const cudnnConvolutionDescriptor_t   convDesc,
    const cudnnTensorDescriptor_t          dxDesc,
    const int                        requestedAlgoCount,
    int                              *returnedAlgoCount,
    cudnnConvolutionBwdDataAlgoPerf_t
*perfResults)
```

```
nnStatus_t cudnnCreate(cudnnHandle_t *handle)
```

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

- **Only call core routines in**

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

```
     const cudnnTensorDescriptor_             Desc,
     const void                              y,
     const cudnnConvolutionDescriptor_       onvDesc,
     cudnnConvolutionBwdFilterAlgo_t          algo,
     void                                    *workSpace,
     size_t                            workSpaceSizeInBytes,
     const void                              *beta,
     const cudnnFilterDescriptor_t            dwDesc,
     void                                    *dw
```

```
cudnnStatus_t
cudnnGetConvolutionBackwardDataWorkspaceSize
    cudnnHandle_t                              handle
    const cudnnFilterDescriptor_t               wDesc
    const cudnnTensorDescriptor_t               dyDes
    const cudnnConvolutionDescriptor_t convD
    const cudnnTensorDescriptor_t             dxDesc
    cudnnConvolutionBwdDataAlgo_t              algo,
    size_t                                  *sizeInByte
```

```
cudnnStatus_t cudnnCreateFilterDescriptor(
    cudnnFilterDescriptor_t *filterDesc)
```

Frontend

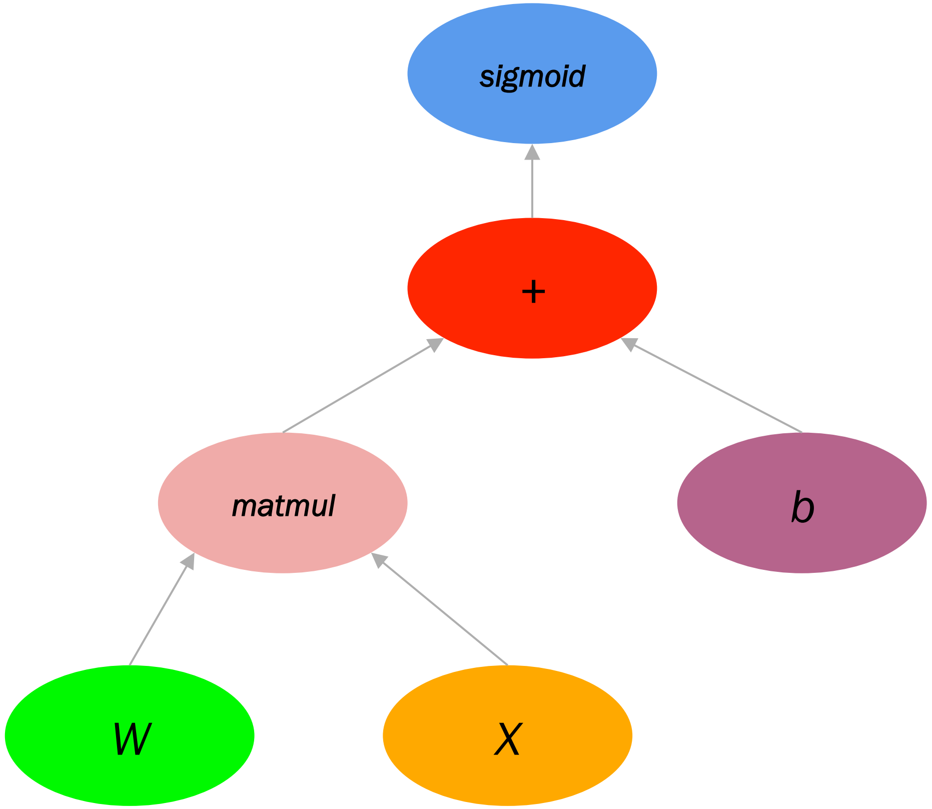
- Core (memory, tensors, communication, math kernels)
    - Completely separate from frontend
    - Correct & Fast
- Frontend (Compute Graph, Optimizers, Models)
    - Hardware agnostic
    - Clean interface
    - Use core API

Compute Graph

Compute Graph

- Represents training control flow/ math operations
- Directed Acyclic Graph
- Eager vs. Non-Eager
- Represents every computation

$$\sigma(W.X + b)$$



Compute Graph

```
class Operation {
    virtual Tensor eval() = 0;

    /* Compute gradient wrt x
        G is incoming backprop gradient */
    virtual Tensor grad(Operation *x, Tensor const& G) = 0;

    vector<Operation*> children_;
    Operation *parent_;
};
```

Base Class

- ***Operation* in MagmaDNN**
- **Offers eval/grad functions**
- **Abstract**

Compute Graph

```
class MatmulOp : public Operation {
    MatmulOp(Operation *A, Operation *B) :
           children_({A,B}), A_(A), B_(B) {}

    Tensor eval() {
        return math::gemm(A->eval(), B->eval());
    }
    Tensor grad(Operation *x, Tensor const& G) {
        if (x == A_)
            return math::gemm(G, B_->eval(), B_TRANSPOSE);
        else
            return math::gemm(A_->eval(), G, A_TRANSPOSE);
     }
private:
    Operation *A_, *B_;
};
```

Example Matmul Operation

eval()

- **Evaluates children**
- **Returns tensor with computed result**

grad()

- **Computes gradient wrt an input**
- **Used in backpropagation algorithm**

Compute Graph

```
/* (1) */
Operation *matmul(Operation *A, Operation *B) {
    return new MatmulOp(A, B);
}

/* (2) */
typedef std::unique_ptr<Operation> Op;
Op&& matmul(Op A, Op B) {
    return std::make_unique<MatmulOp>(A.get(), B.get());
}
```

Usability

- **Define wrapper functions to construct nodes behind scene**
- **Pointers necessary for Data Structures**
- **Use C++ to manage memory**

Compute Graph

```
Tensor xTensor, wTensor, bTensor;

auto x = var(xTensor);
auto w = var(wTensor);
auto b = var(bTensor);

auto result = add(matmul(w, x), b);
Tensor resultTensor = result->eval();
```

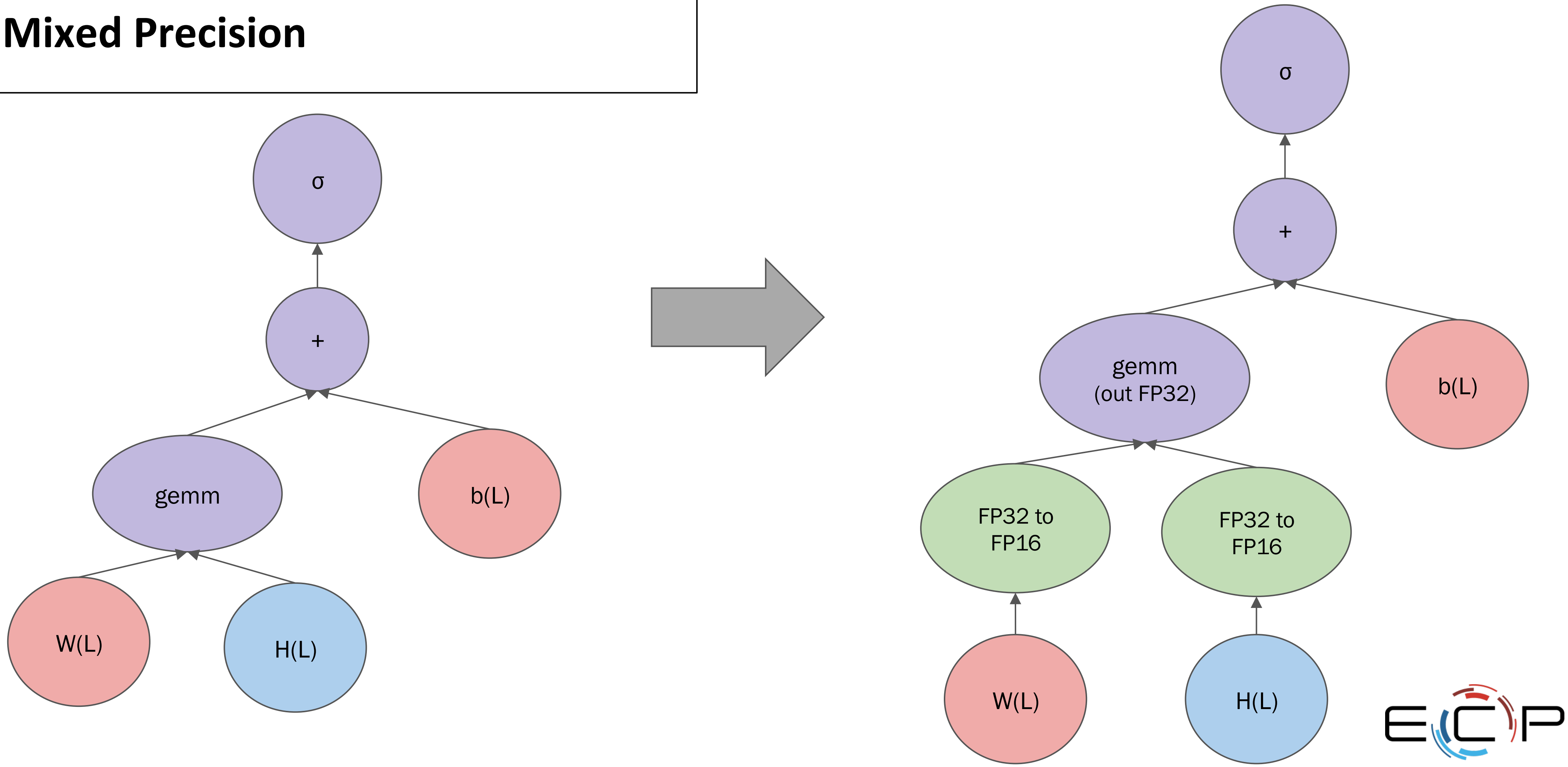
Usability

- **Create *Variable* operation as entry points to graph**
- **End user can now construct graph easily with wrapper functions**
- **Must eval to get final result**

Compute Graph

Optimizations

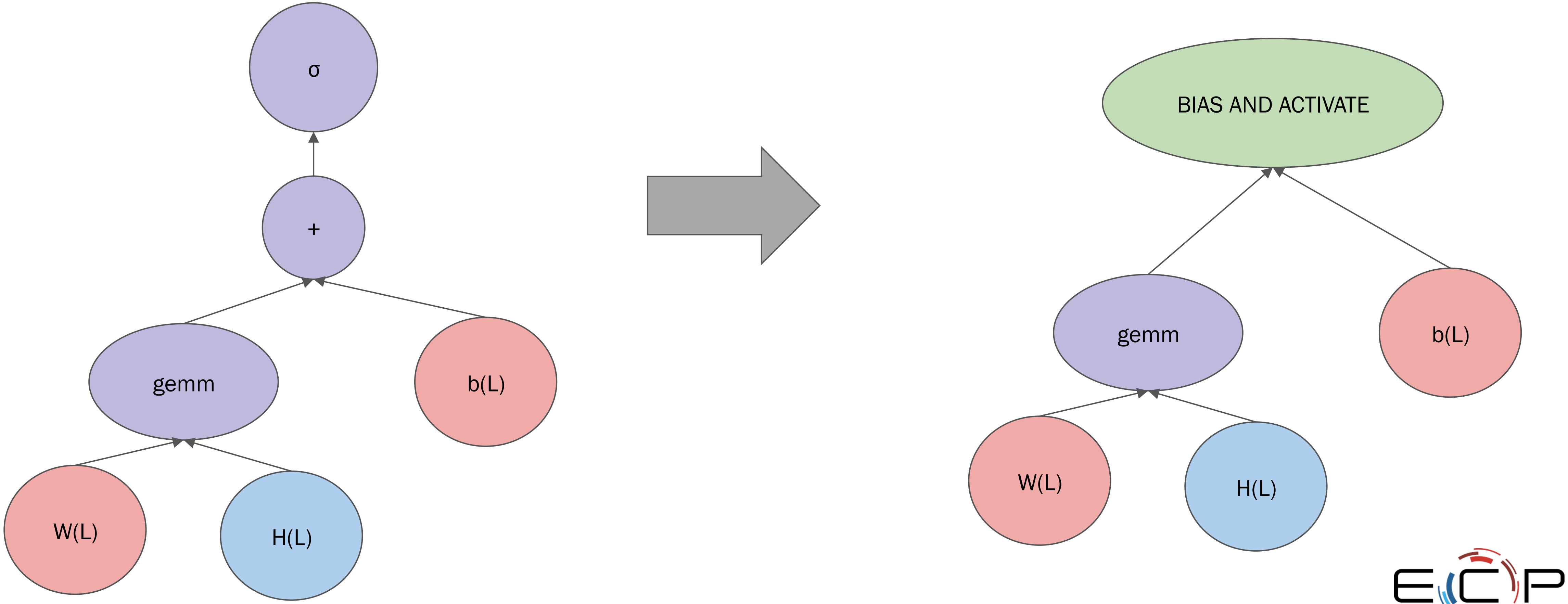
- **Mixed Precision**



Compute Graph

Optimizations

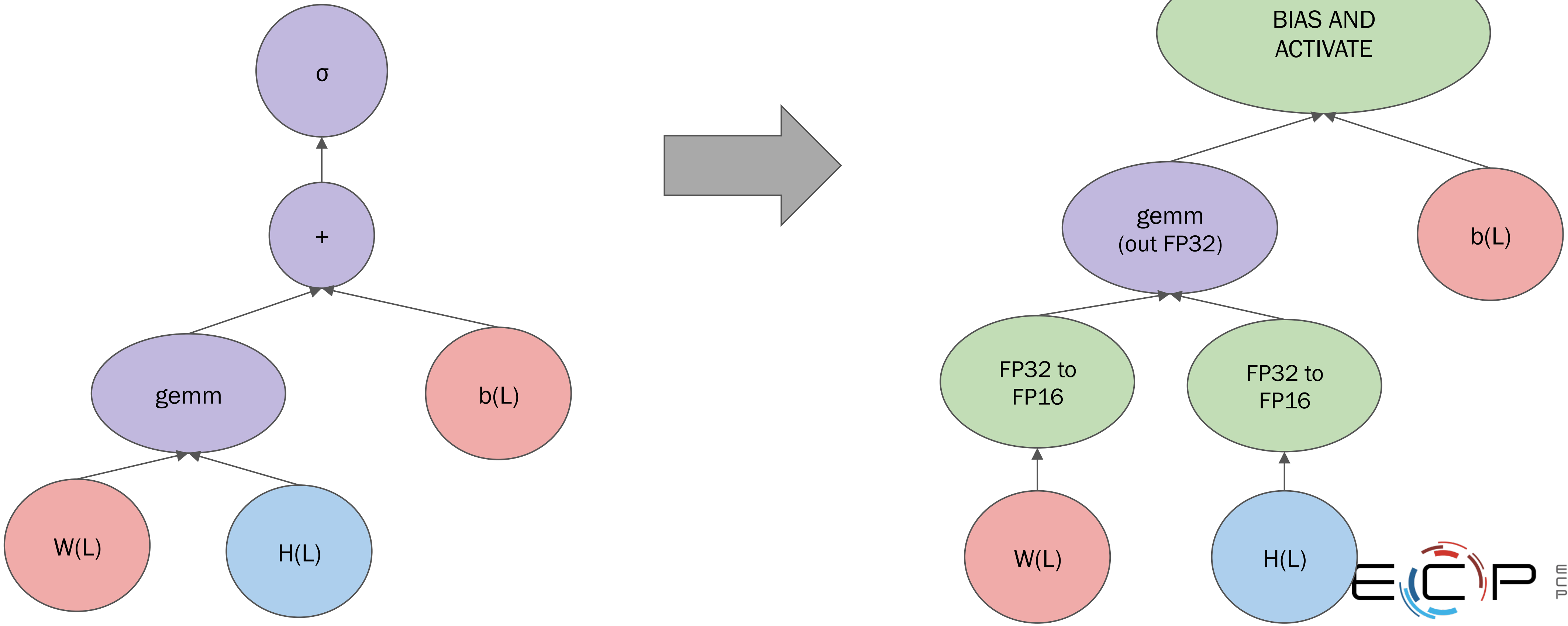
- **Fused Operations**



Compute Graph

Optimizations

- **All optimizations**
- **Tunable**



Compute Graph

```
map<Op, Tensor> getGradTable(x, G, T):
     map<Op, Tensor> gradTable
     gradTable[x] = 1
     for operation in Graph:
        setGradient(operation, Graph, trainable, gradTable)
     return gradTable

setGradient(x, G, T, table):
    grad = table[x]
    if grad exists:
        return grad
    G = 0
    for consumer in x.consumers():
        setGradient(child, G, T, table)
        G = G + consumer->grad(x, grad)
    table[x] = G
```

Computing Gradients

- **Simplified version**
- **Dynamic Programming**
- **Only works for 1 output**

Other Implementations and Discussions

- **See “Deep Learning” by Ian Goodfellow**
- **See “autograd” or “jax”**

Optimizer

Interface

- **Minimize compute graph node w.r.t. an input node**
- **ex. min pow(X, 2) w.r.t. x**
- **Support iterative methods**

Abstraction

- **Generic optimizer class used by fit functions**
- **Support many types SGD, SGD+Nesterov Momentum, Adam, RMSProp, …**
- **Distributed Optimizers**

Implementation

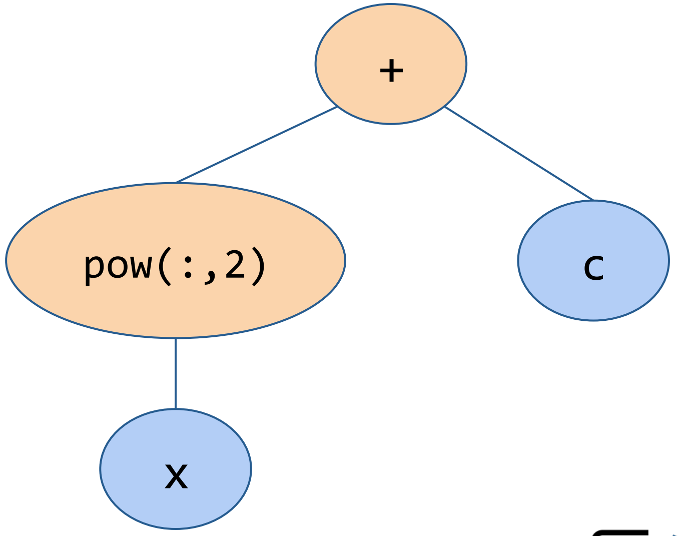
- **Use existing tensor math core**
- **Easy hyperparameter manipulation**

Optimizer

```
auto x = op::var<float>("x", NONE);
auto c = op::var<float>("c", {CONSTANT, -2.0f});

optimizer::GradientDescent opt(0.05);

opt.minimize( op::add(op::pow(x, 2), c), {x});
```

Example Use Case

- **Minimize any compute graph**
- **Easy for end user**



Optimizer

```
class Optimizer {
    Optimizer(params…) {}

    virtual void minimize(Op x, Op wrt) {
        for (1 … limit) {
            step(x, wrt);
           }
    }

    virtual void step(Op x, Op wrt) = 0;
};
```

The Interface

- **“Minimize”**
  - **Where to implement minimization routine**
  - **Minimizes x with respect to wrt**
- **“Step”**
  - **A single step for iterative methods**
- **Use Grad Table to minimize for gradient based methods**

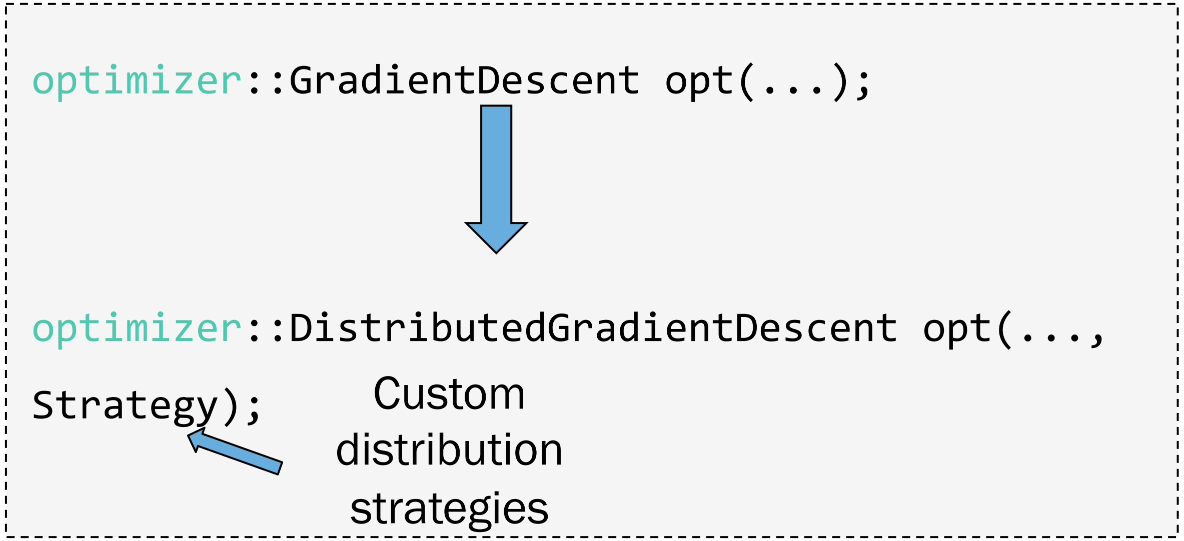
Optimizer

```
class SGD : public Optimizer {
    Optimizer(float lr): lr_(lr) {}

    virtual void step(Op x, Op wrt) override {
        auto grads = getGradTable(x, wrt,{x});

        math::sgd_update(x, grads[wrt], lr_);
    }
};
```

An Example Optimizer -- SGD

- Use grad table to get gradients
- Math Core kernels to update
- Use a learning rate
- Only need to override step function

Optimizer



```
optimizer::GradientDescent opt(...);

optimizer::DistributedGradientDescent opt(...,
Strategy);
```

Custom distribution strategies

Easy Change To Run Distributed

- User only has to change optimizer type
- Uses communication backend
  - Best performance
  - Best portability
- Still need to run with mpirun

Model

Tie Everything Together

- Everything needed to run a DNN is implemented
- Wrap DNN utilities in model class for users
- Can also support other ML models

Implementation

- Use rest of backend and frontend to implement
- Should just "work"

Optimization

- Use Core API

Layers

```
class Layer {
    virtual Op output() = 0;
};

class FullyConnected : public Layer {
    FullyConnected(Op input, int hiddenUnits, bool
useBias = True) {
        weights_ = Var({input.shape(0), hiddenUnits});
     }

     virtual Op output() {
        Op tmp = matmul(weights_, input);
        return (use_bias) ? add(tmp, bias) : tmp;
     }

     Op weights_;
};
```

The Layer Class

- **A helper class**
- **Help construct multiple compute graph nodes at once**
- **More familiar construct to data scientist**
- **Define helper functions like with operations**

Layers

```
Op input = Var({batchSize, inputFeatures});

Layer *fc1 = fullyConnected(input, 512, true);
Layer *act1 = activation(fc1->output(), RELU);

Layer *fc2 = fullyconnected(act1->output(), 32);
Layer *act2 = activation(fc2->output(), RELU);
```

Example Constructing Layers

- **act2->output() is top of compute graph**
- **act2->output()->eval() is a working neural network**
- **Almost there!**

Model

```
class Model {
    virtual void summary() = 0;

    virtual void fit(Tensor &x, Tensor &y,…)=0;
    virtual Tensor predict(Tensor &x) = 0;
};
```

The Base Model

- **Every Model has a “fit” or “train” function**
  - **Where learning happens**
- **Every model has a “predict” function**
  - **To use the model for actual classification tasks**

Model

```
class NeuralNetwork : public Model {
  …

  virtual void train(X, Y, epochs, optimizer) {
    for i … epochs {
      for n … batchesPerEpoch {
      input_.copy(batches.get(n));

      optimizer.minimize(lossFunc->output(),{weights});


    }
    }
  }
};
```

Training Routine (Abbreviated)

- Put loss function on last layer
- For every batch
    - Copy data into input
    - Run optimizer
- Network is training!
- Can also record loss, accuracy, and training time here

Model

```
NeuralNetwork nn(layers);

nn.fit(xData, yData, nEpochs, SGD);
```

Example Use Case

- Training routines are now easy
- Remove error prone code

Software Structure of a DNN Framework

- Core
  - Systems
    - Memory (CPU, GPU, Hybrid)
    - Communication (MPI, OpenMP)
  - Math
    - Tensor data structures and operations
    - Blas/Lapack, Magma
    - DNN accelerator (CuDNN, hipDNN, OneDNN)
- Frontend
  - Compute Graph
  - Optimizers
  - Models

DNN Programming

MagmaDNN 1.2

Rocco Febbo

Introduction | Examples | Core DNN Concepts

Development Version

https://github.com/MagmaDNN/magmadnn

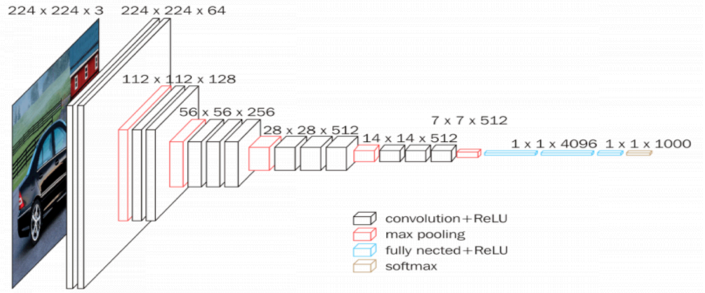
Release Version

https://bitbucket.org/icl/magmadnn/src/master/

DNN Programming Structure Exercises

What's New in MagmaDNN 1.2

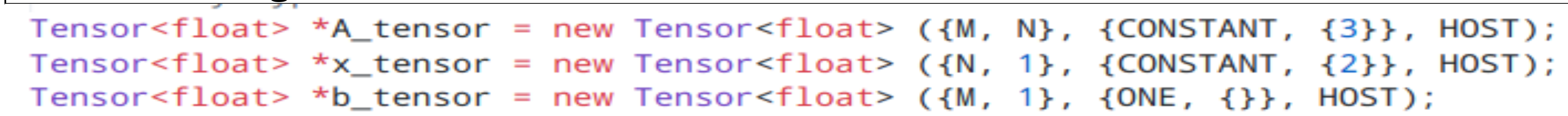
- oneDNN(MKLDNN) Support
  - fully connected, convolutional, and pooling layers
- CMake Build System
- New Examples
  - CNN
  - ResNet
  - AlexNet
  - LeNet
  - MNIST and CIFAR interactive
  - VGG16
  - Tensor Math
- C++ Style Formatter
- Modularized Distributed Optimizer
- CIFAR10, CIFAR100, MNIST dataloaders
- Cuda Stream
- Model Summary
- Spack Package Manager Support



DNN Programming Structure Exercises

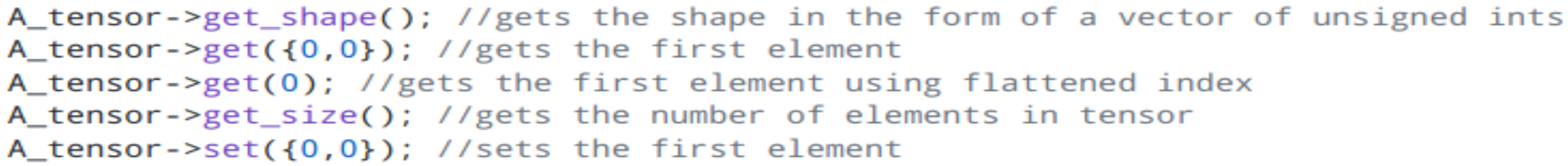
Tensors

MagmaDNN primarily interfaces with memory in the form of a Tensor. The shape of the tensor is defined when it is created along with how it should be filled and what device it should be stored on.



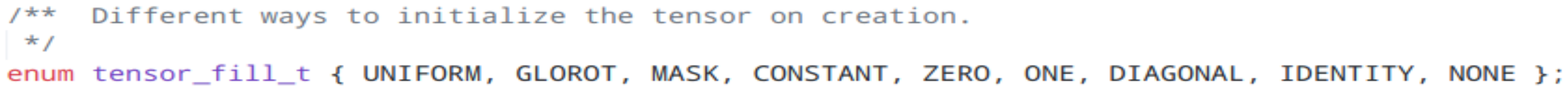
```
Tensor<float> *A_tensor = new Tensor<float> ({M, N}, {CONSTANT, {3}}, HOST);
Tensor<float> *x_tensor = new Tensor<float> ({N, 1}, {CONSTANT, {2}}, HOST);
Tensor<float> *b_tensor = new Tensor<float> ({M, 1}, {ONE, {}}, HOST);
```

The Tensor interface has many built-in functions. A few are shown below.



```
A_tensor->get_shape(); //gets the shape in the form of a vector of unsigned ints
A_tensor->get({0,0}); //gets the first element
A_tensor->get(0); //gets the first element using flattened index
A_tensor->get_size(); //gets the number of elements in tensor
A_tensor->set({0,0}); //sets the first element
```

Tensors can be initialized using one of the nine predefined formats as shown below.



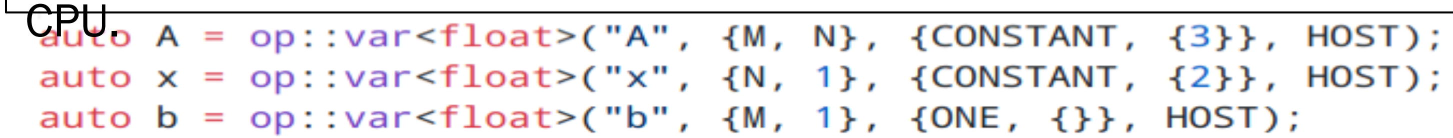
```
/**  Different ways to initialize the tensor on creation.
 */
enum tensor_fill_t { UNIFORM, GLOROT, MASK, CONSTANT, ZERO, ONE, DIAGONAL, IDENTITY, NONE };
```

DNN Programming Structure Exercises

Compute Graph

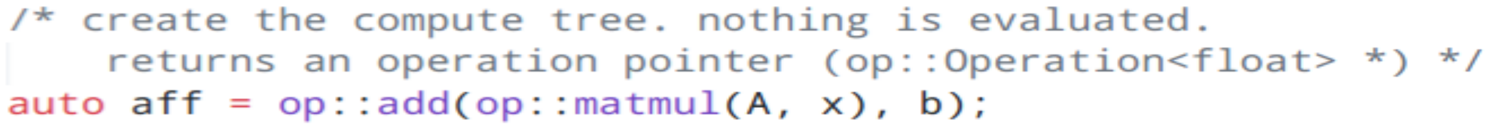
MagmaDNN performs computations by creating a compute graph then evaluating it. This has the benefit of allowing the compute graph to be optimized before the calculation is evaluated. Below is a tensor calculation of Ax+b.

First the variables are created with initialized values. Here A is an M by N matrix filled with the value 3. ‘x’ is a vector of length N filled with 2. ‘b’ is a vector of length M filled with the value 1. Each of these variables are stored on system RAM due to the ‘HOST’ specification. This means that any calculations will be performed by the CPU.



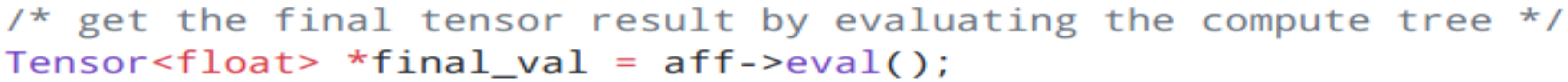
```
auto A = op::var<float>("A", {M, N}, {CONSTANT, {3}}, HOST);
auto x = op::var<float>("x", {N, 1}, {CONSTANT, {2}}, HOST);
auto b = op::var<float>("b", {M, 1}, {ONE, {}}, HOST);
```

Here an operation using those three variables is defined.



```
/* create the compute tree. nothing is evaluated.
   returns an operation pointer (op::Operation<float> *) */
auto aff = op::add(op::matmul(A, x), b);
```

Now, the operation is evaluated on the CPU.

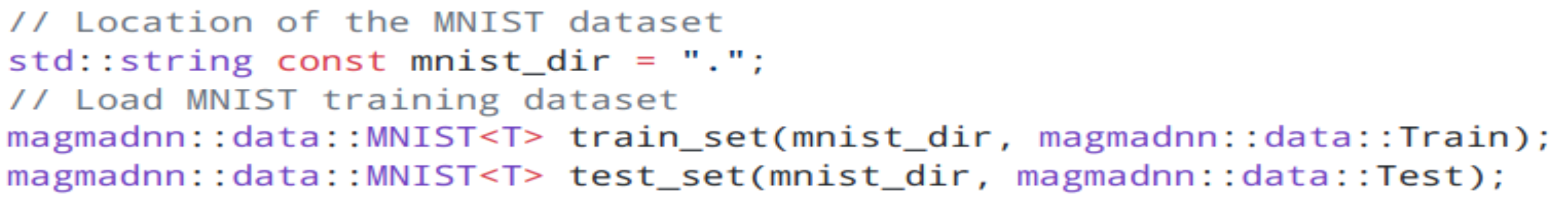


```
/* get the final tensor result by evaluating the compute tree */
Tensor<float> *final_val = aff->eval();
```

DNN Programming Structure Exercises

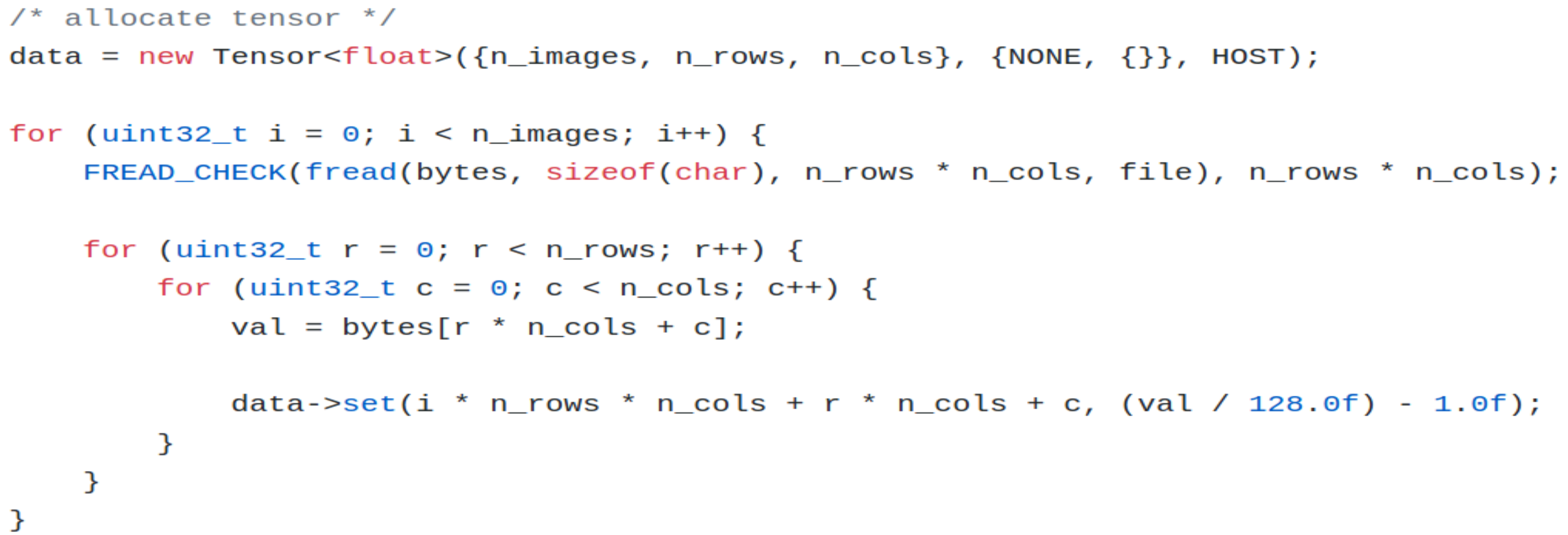
Reading Data

When building your own neural network, creating a data loader interface can be useful. In MagmaDNN, there are built-in interfaces for loading the MNIST datasets and CIFAR datasets.



```
// Location of the MNIST dataset
std::string const mnist_dir = ".";
// Load MNIST training dataset
magmadnn::data::MNIST<T> train_set(mnist_dir, magmadnn::data::Train);
magmadnn::data::MNIST<T> test_set(mnist_dir, magmadnn::data::Test);
```

Data can also be loaded manually using the tensor interfaces. When a tensor is initialized, it’s shape and memory type are specified

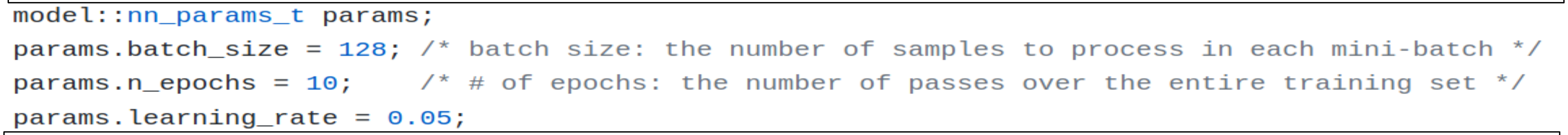


```
/* allocate tensor */
data = new Tensor<float>({n_images, n_rows, n_cols}, {NONE, {}}, HOST);

for (uint32_t i = 0; i < n_images; i++) {
    FREAD_CHECK(fread(bytes, sizeof(char), n_rows * n_cols, file), n_rows * n_cols);

    for (uint32_t r = 0; r < n_rows; r++) {
        for (uint32_t c = 0; c < n_cols; c++) {
            val = bytes[r * n_cols + c];

            data->set(i * n_rows * n_cols + r * n_cols + c, (val / 128.0f) - 1.0f);
        }
    }
}
```

DNN Programming Structure Exercises

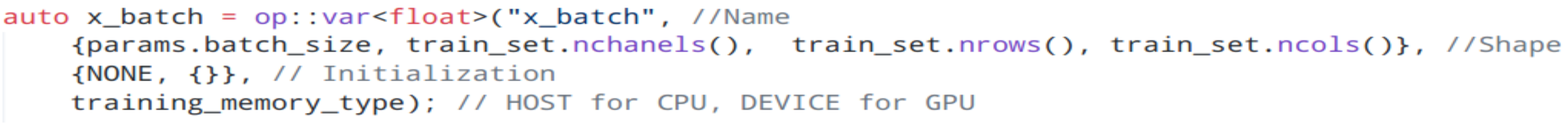
Model Parameters and Network Input

The parameters for batch size, epochs, and learning rate are defined in a struct and passed to the model when it is created.



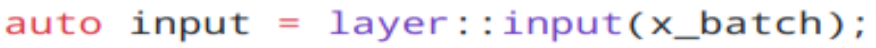
```
model::nn_params_t params;
params.batch_size = 128; /* batch size: the number of samples to process in each mini-batch */
params.n_epochs = 10;    /* # of epochs: the number of passes over the entire training set */
params.learning_rate = 0.05;
```

An operation is defined for the input to the model with the needed shape. The memory type selects which device our network will train on.



```
auto x_batch = op::var<float>("x_batch", //Name
    {params.batch_size, train_set.nchanels(),  train_set.nrows(), train_set.ncols()}, //Shape
    {NONE, {}}, // Initialization
    training_memory_type); // HOST for CPU, DEVICE for GPU
```

This will be passed into the first layer of our network.



```
auto input = layer::input(x_batch);
```

DNN Programming Structure Exercises

Activation and Fit Functions

As of version 1.2 MagmaDNN has four activation functions, four optimizers, and two loss functions.

```
enum activation_t { SIGMOID, TANH, RELU, SOFTMAX };
```

```
enum loss_t { CROSS_ENTROPY, MSE };
```

```
enum optimizer_t {
     SGD,
     ADAGRAD,
     RMSPROP,
     ADAM,
};
```

Sigmoid

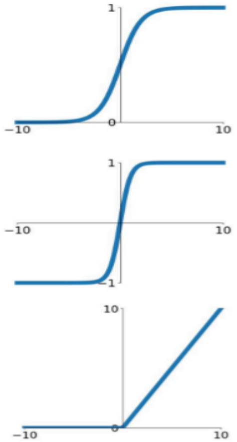
$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh

$\tanh(x)$

ReLU

$\max(0, x)$



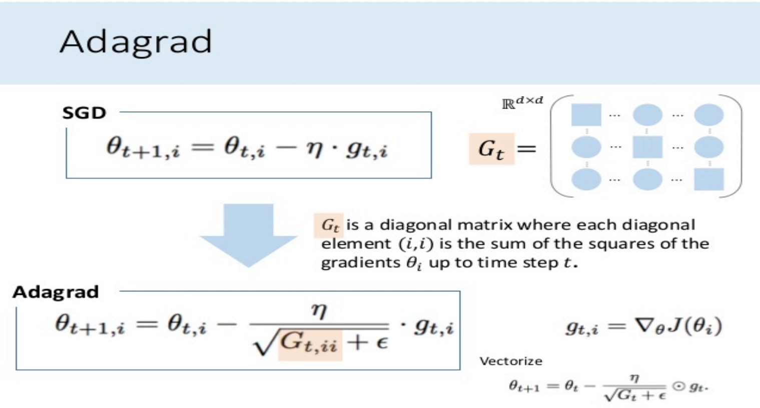
Adam

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

RMSProp

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)\left(\frac{\delta C}{\delta w}\right)^2$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{E[g^2]_t}} \frac{\delta C}{\delta w}$$



Adagrad

SGD

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

$G_t =$ $\mathbb{R}^{d\times d}$

G_t is a diagonal matrix where each diagonal element (i,i) is the sum of the squares of the gradients θ_i up to time step t.

Adagrad

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$$g_{t,i} = \nabla_\theta J(\theta_i)$$

Vectorize

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

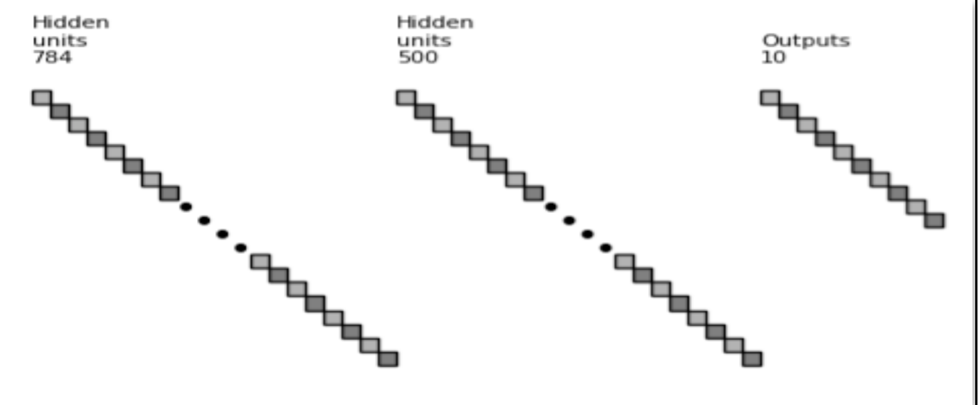
image: https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

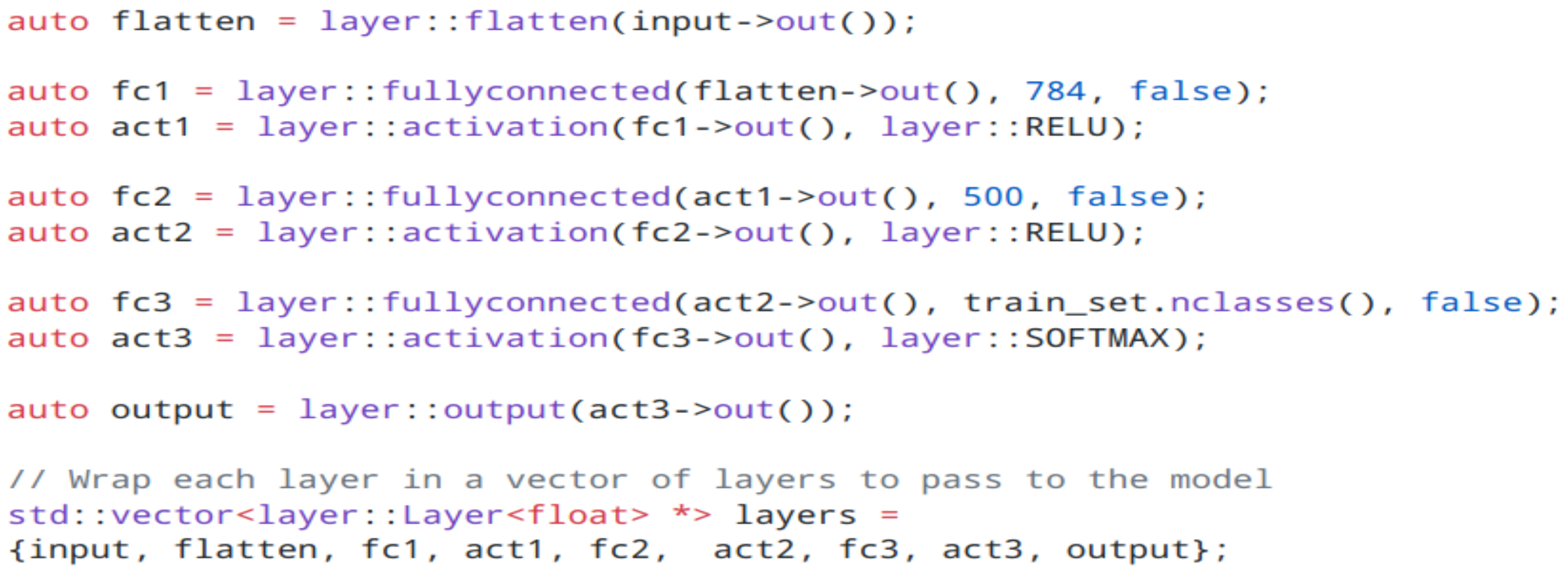
DNN Programming Structure Exercises

Now, the structure of our network can be defined!

For this network, we will use 2 hidden layers to create a multi-layer perceptron network. First we flatten the input since it's likely multidimensional. When each layer is defined, the input to that layer is provided. This is usually the output of the previous layer. Then, the number of hidden units is declared and if a bias for that layer is needed. The layers are then put in a vector to pass to the model.

Simple MLP Network



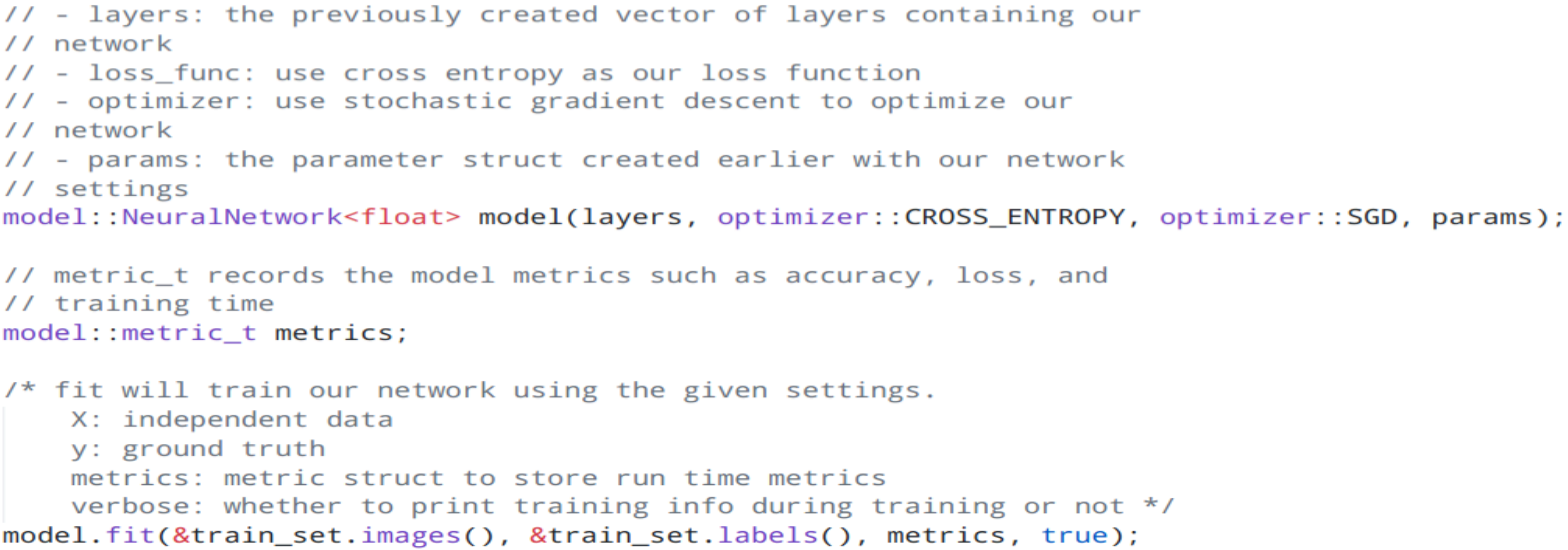


```
auto flatten = layer::flatten(input->out());

auto fc1 = layer::fullyconnected(flatten->out(), 784, false);
auto act1 = layer::activation(fc1->out(), layer::RELU);

auto fc2 = layer::fullyconnected(act1->out(), 500, false);
auto act2 = layer::activation(fc2->out(), layer::RELU);

auto fc3 = layer::fullyconnected(act2->out(), train_set.nclasses(), false);
auto act3 = layer::activation(fc3->out(), layer::SOFTMAX);

auto output = layer::output(act3->out());

// Wrap each layer in a vector of layers to pass to the model
std::vector<layer::Layer<float> *> layers =
{input, flatten, fc1, act1, fc2,  act2, fc3, act3, output};
```

DNN Programming Structure Exercises

Fitting the Model

The layers, loss function, optimizer, and model parameters are specified when a model is initialized. The network begins training when the "fit" member function is called using the loaded data.



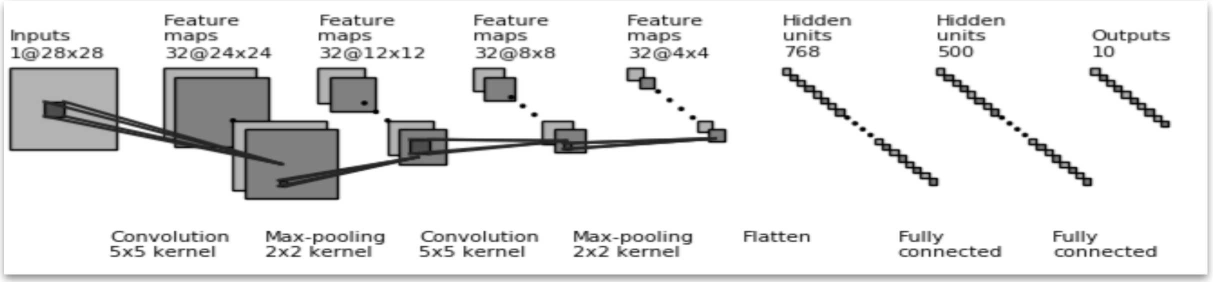
```
// - layers: the previously created vector of layers containing our
// network
// - loss_func: use cross entropy as our loss function
// - optimizer: use stochastic gradient descent to optimize our
// network
// - params: the parameter struct created earlier with our network
// settings
model::NeuralNetwork<float> model(layers, optimizer::CROSS_ENTROPY, optimizer::SGD, params);

// metric_t records the model metrics such as accuracy, loss, and
// training time
model::metric_t metrics;

/* fit will train our network using the given settings.
    X: independent data
    y: ground truth
    metrics: metric struct to store run time metrics
    verbose: whether to print training info during training or not */
model.fit(&train_set.images(), &train_set.labels(), metrics, true);
```

DNN Programming Structure Exercises

Creating a Convolutional Neural Network

Creating a convolutional neural network is as simple as adding convolutional layers to a network. In the following code we will create a network with 2 convolutional layers each followed by a max pooling layer. We will then flatten the output of the pooling layer and pass it to 2 MLP layers.

```
auto conv2d1 = layer::conv2d(input->out(), {5, 5}, 32, {0, 0});
auto act1 = layer::activation(conv2d1->out(), layer::RELU);
auto pool1 = layer::pooling(act1->out(), {2, 2}, {0, 0}, {2, 2});

auto conv2d2 = layer::conv2d(pool1->out(), {5, 5}, 32, {0, 0});
auto act2 = layer::activation(conv2d2->out(), layer::RELU);
auto pool2 = layer::pooling(act2->out(), {2, 2}, {0, 0}, {2, 2});

auto flatten = layer::flatten(pool2->out());

auto fc1 = layer::fullyconnected(flatten->out(), 768, true);
auto act3 = layer::activation(fc1->out(), layer::RELU);

auto fc2 = layer::fullyconnected(act3->out(), 500, true);
auto act4 = layer::activation(fc2->out(), layer::RELU);

auto fc3 = layer::fullyconnected(act4->out(), n_classes, false);
auto act5 = layer::activation(fc3->out(), layer::SOFTMAX);

auto output = layer::output(act5->out());
```

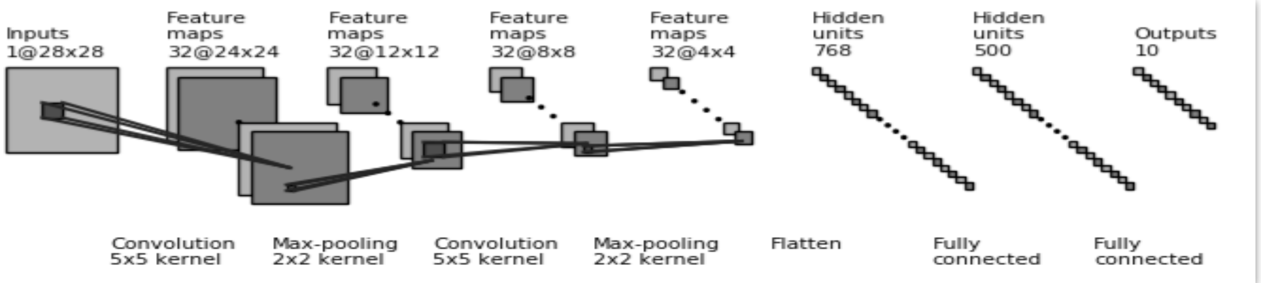


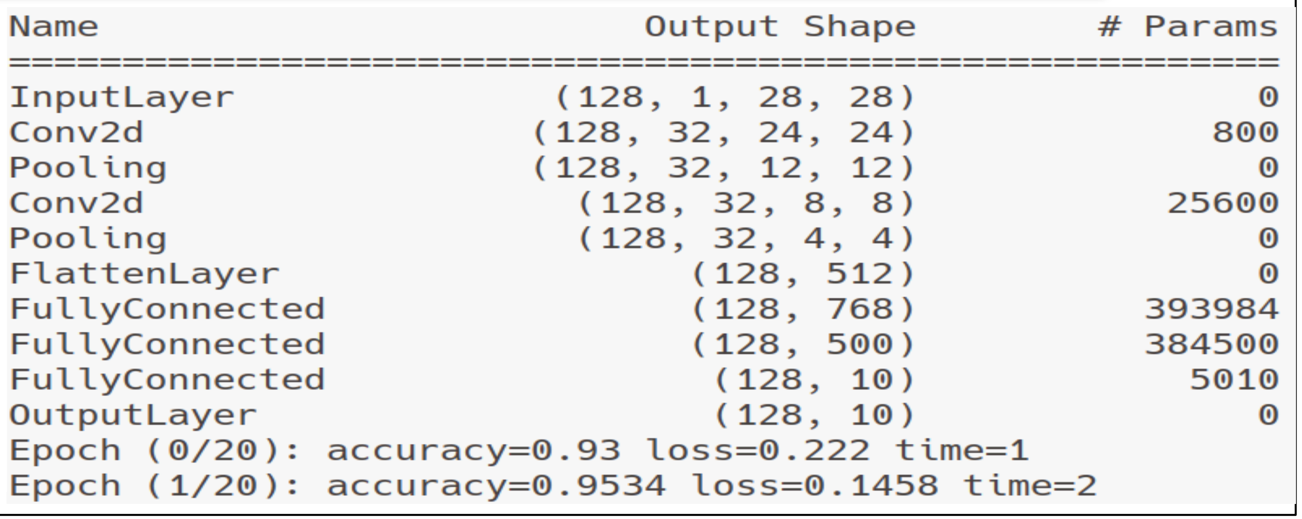
DNN Programming Structure Exercises

Neural Network Model Summary

Using the network from the previous example, MagmaDNN can output a model summary similar to Tensorflow. When the summary member function is called, each layer is shown in order with the output shape of that layer shown in the middle column and the number of parameters within that layer shown on the right column.

```
std::vector<layer::Layer<float> *> layers =
   {input,
    conv2d1, act1,
    pool1,
    conv2d2, act2,
    pool2,
    flatten,
    fc1,   act3,
    fc2,  act4,
    fc3,  act5,
    output};

model::NeuralNetwork<float> model(layers,
    optimizer::CROSS_ENTROPY,
    optimizer::SGD, params);

model::metric_t metrics;

model.summary();

model.fit(images_host, labels_host, metrics, true);
```





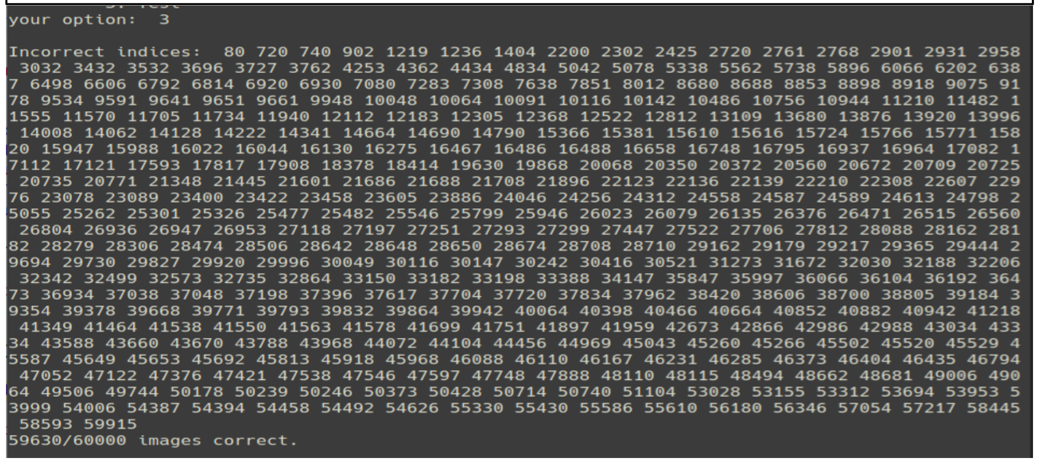
```
Name                              Output Shape          # Params
==================================================================
InputLayer                     (128, 1, 28, 28)                0
Conv2d                        (128, 32, 24, 24)              800
Pooling                       (128, 32, 12, 12)                0
Conv2d                          (128, 32, 8, 8)            25600
Pooling                         (128, 32, 4, 4)                0
FlattenLayer                         (128, 512)                0
FullyConnected                       (128, 768)           393984
FullyConnected                       (128, 500)           384500
FullyConnected                        (128, 10)             5010
OutputLayer                           (128, 10)                0
Epoch (0/20): accuracy=0.93 loss=0.222 time=1
Epoch (1/20): accuracy=0.9534 loss=0.1458 time=2
```

DNN Programming Structure Exercises

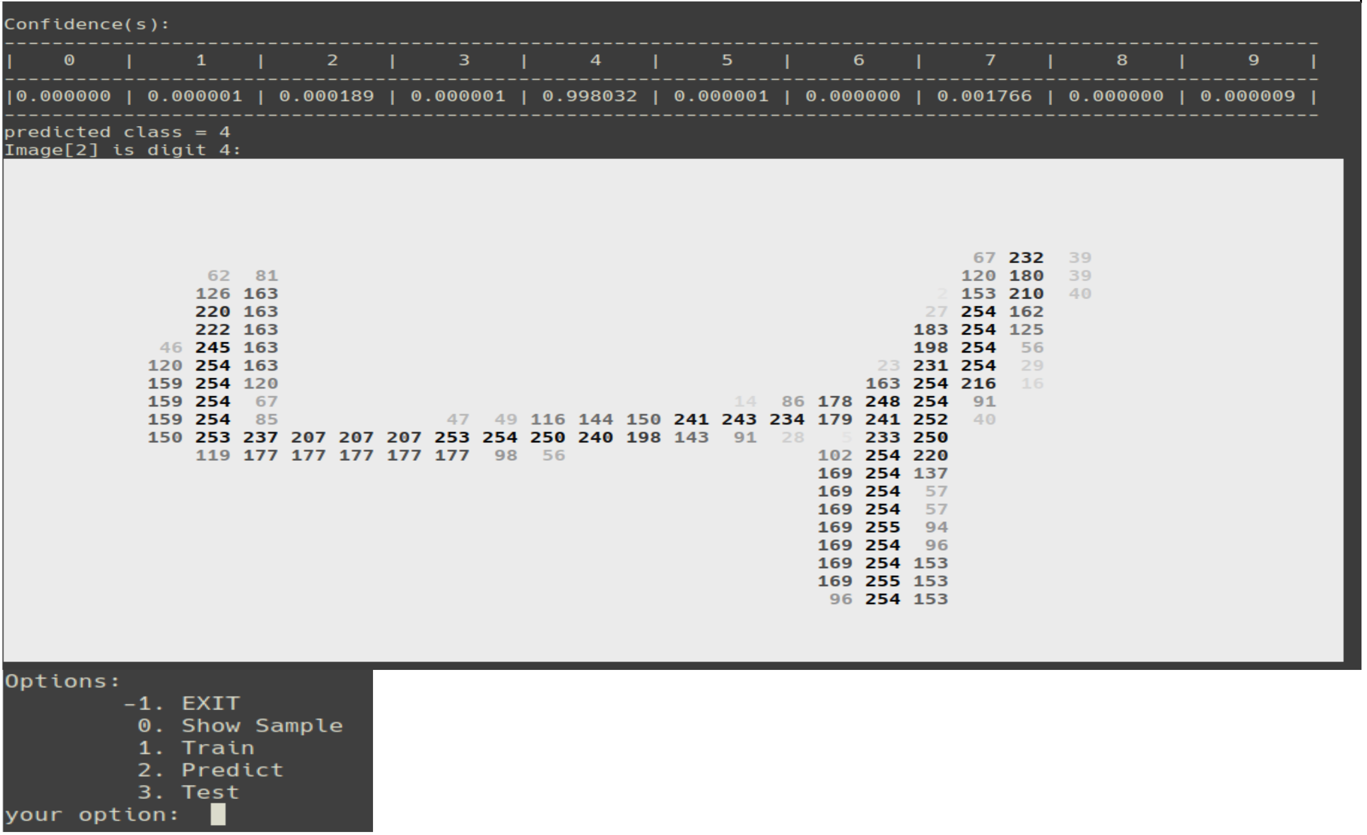
Interactive examples: MNIST

Included as one of the examples in MagmaDNN is the MNIST interactive example. This example gives you an untrained neural network with the options to:

- Show Sample
  - show a user selected image in the terminal
- Train
  - trains the network
- Predict
  - predict a specific sample with confidence matrix
- Test
  - Show image indexes with bad predictions



```
         3. Test
your option:  3

Incorrect indices:  80 720 740 902 1219 1236 1404 2200 2302 2425 2720 2761 2768 2901 2931 2958
 3032 3432 3532 3696 3727 3762 4253 4362 4434 4834 5042 5078 5338 5562 5738 5896 6066 6202 638
7 6498 6606 6792 6814 6920 6930 7080 7283 7308 7638 7851 8012 8680 8688 8853 8898 8918 9075 91
78 9534 9591 9641 9651 9661 9948 10048 10064 10091 10116 10142 10486 10756 10944 11210 11482 1
1555 11570 11705 11734 11940 12112 12183 12305 12368 12522 12812 13109 13680 13876 13920 13996
 14008 14062 14128 14222 14341 14664 14690 14790 15366 15381 15610 15616 15724 15766 15771 158
20 15947 15988 16022 16044 16130 16275 16467 16486 16488 16658 16748 16795 16937 16964 17082 1
7112 17121 17593 17817 17908 18378 18414 19630 19868 20068 20350 20372 20560 20672 20709 20725
 20735 20771 21348 21445 21601 21686 21688 21708 21896 22123 22136 22139 22210 22308 22607 229
76 23078 23089 23400 23422 23458 23605 23886 24046 24256 24312 24558 24587 24589 24613 24798 2
5055 25262 25301 25326 25477 25482 25546 25799 25946 26023 26079 26135 26376 26471 26515 26560
 26804 26936 26947 26953 27118 27197 27251 27293 27299 27447 27522 27706 27812 28088 28162 281
82 28279 28306 28474 28506 28642 28648 28650 28674 28708 28710 29162 29179 29217 29365 29444 2
9694 29730 29827 29920 29996 30049 30116 30147 30242 30416 30521 31273 31672 32030 32188 32206
 32342 32499 32573 32735 32864 33150 33182 33198 33388 34147 35847 35997 36066 36104 36192 364
73 36934 37038 37048 37198 37396 37617 37704 37720 37834 37962 38420 38606 38700 38805 39184 3
9354 39378 39668 39771 39793 39832 39864 39942 40064 40398 40466 40664 40852 40882 40942 41218
 41349 41464 41538 41550 41563 41578 41699 41751 41897 41959 42673 42866 42986 42988 43034 433
34 43588 43660 43670 43788 43968 44072 44104 44456 44969 45043 45260 45266 45502 45520 45529 4
5587 45649 45653 45692 45813 45918 45968 46088 46110 46167 46231 46285 46373 46404 46435 46794
 47052 47122 47376 47421 47538 47546 47597 47748 47888 48110 48115 48494 48662 48681 49006 490
64 49506 49744 50178 50239 50246 50373 50428 50714 50740 51104 53028 53155 53312 53694 53953 5
3999 54006 54387 54394 54458 54492 54626 55330 55430 55586 55610 56180 56346 57054 57217 58445
 58593 59915
59630/60000 images correct.
```



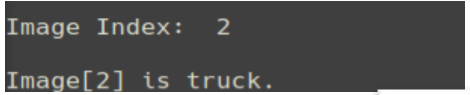
```
Confidence(s):
---------------------------------------------------------------------------------------------------------------
|    0    |     1    |     2    |     3    |     4    |     5    |     6    |     7    |     8    |     9    |
---------------------------------------------------------------------------------------------------------------
|0.000000 | 0.000001 | 0.000189 | 0.000001 | 0.998032 | 0.000001 | 0.000000 | 0.001766 | 0.000000 | 0.000009 |
---------------------------------------------------------------------------------------------------------------
predicted class = 4
Image[2] is digit 4:

                                                                                67 232  39
                62  81                                                         120 180  39
               126 163                                                       2 153 210  40
               220 163                                                      27 254 162
               222 163                                                     183 254 125
            46 245 163                                                     198 254  56
           120 254 163                                                  23 231 254  29
           159 254 120                                                 163 254 216  16
           159 254  67                                      14  86 178 248 254  91
           159 254  85              47  49 116 144 150 241 243 234 179 241 252  40
           150 253 237 207 207 207 253 254 250 240 198 143  91  28   5 233 250
               119 177 177 177 177 177  98  56                     102 254 220
                                                                   169 254 137
                                                                   169 254  57
                                                                   169 254  57
                                                                   169 255  94
                                                                   169 254  96
                                                                   169 254 153
                                                                   169 255 153
                                                                    96 254 153

Options:
        -1. EXIT
         0. Show Sample
         1. Train
         2. Predict
         3. Test
your option:
```

DNN Programming Structure Exercises

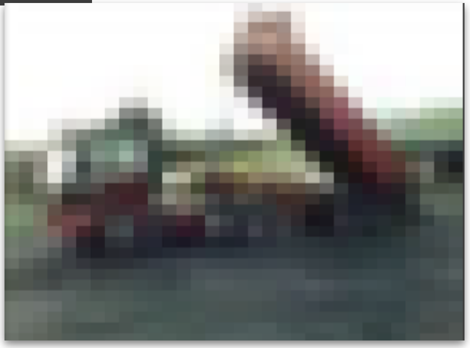
Interactive examples: CIFAR10

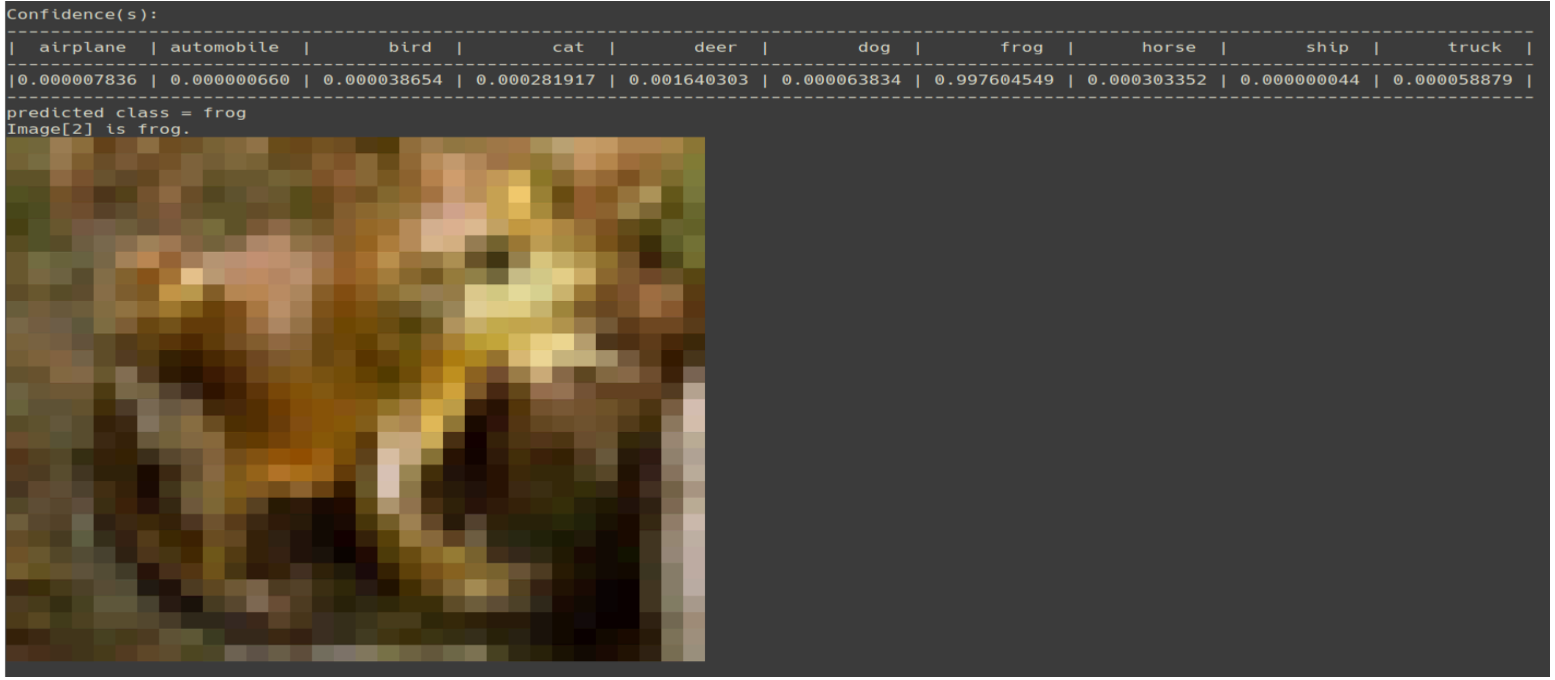
Included as one of the examples in MagmaDNN is the CIFAR10 interactive examples. These examples give you an untrained neural network with the options to:

- Show Sample
  - show a user selected image in the terminal
- Train
  - trains the network
- Predict
  - predict a specific sample with confidence matrix

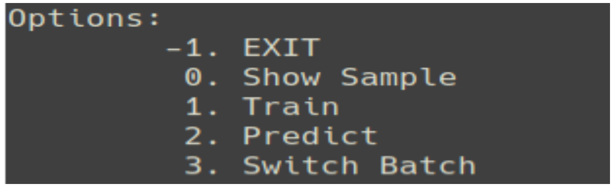


```
Image Index:  2

Image[2] is truck.
```





```
Confidence(s):
-------------------------------------------------------------------------------------------------------------------------------------------
|  airplane  | automobile  |         bird  |          cat  |         deer  |          dog  |         frog  |        horse  |         ship  |        truck  |
-------------------------------------------------------------------------------------------------------------------------------------------
|0.000007836 | 0.000000660 | 0.000038654 | 0.000281917 | 0.001640303 | 0.000063834 | 0.997604549 | 0.000303352 | 0.000000044 | 0.000058879 |
-------------------------------------------------------------------------------------------------------------------------------------------
predicted class = frog
Image[2] is frog.
```



```
Options:
        -1. EXIT
         0. Show Sample
         1. Train
         2. Predict
         3. Switch Batch
```

DNN Programming Structure Exercises

Building MagmaDNN

When building MagmaDNN there are several build options at your disposal. Some of which dertermine the use of external APIs such as CuDNN, Magma, openBLAS, oneDNN, etc. The Cuda, CuDNN, and CuBLAS libaries require an Nvidia GPU to be installed on the system. Likewise, in order to use oneDNN (MKLDNN) an Intel CPU is required to be installed on the system.

MagmaDNN Built-in Functionality

If you are building MagmaDNN on a system incompatible with these libraries, you can use MagmaDNN's built-in tools to perform complex operations on your CPU. The default build behavior will automatically build MagmaDNN this way. Also for compatability, MagmaDNN includes support for building using an included makefile and CMake.

Makefile

1. Make a copy of the make.inc example
2. Adjust parameters accordingly
3. Type 'make install'
4. Type 'make examples' to build examples

CMake

1. Adjust CMakeLists.txt parameters accordingly
2. Create a directory to contain build files (ex. 'mkdir ../build.magmadnn')
3. cd into that directory
4. Type 'cmake ../magmadnn'
5. Type 'make install'

DNN Programming Structure Exercises

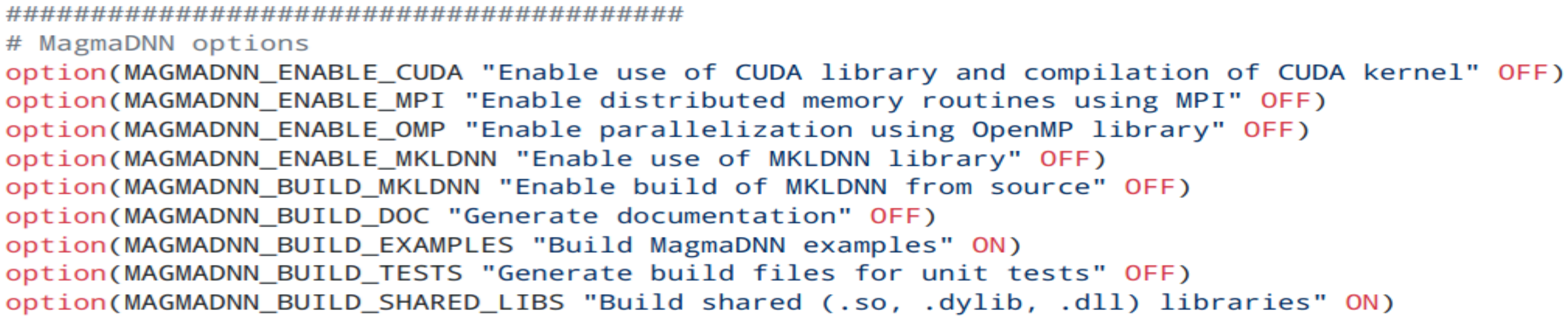
MagmaDNN Hardware Support

Operation	CPU(native)	CPU(MKLDNN)	GPU(Cuda)
adagrad	SUPPORTED	UNSUPPORTED	SUPPORTED
adam	SUPPORTED	UNSUPPORTED	SUPPORTED
rmsprop	SUPPORTED	UNSUPPORTED	SUPPORTED
SGD	SUPPORTED	UNSUPPORTED	SUPPORTED
add	SUPPORTED	UNSUPPORTED	SUPPORTED
argmax	SUPPORTED	UNSUPPORTED	UNSUPPORTED
batch norm	UNSUPPORTED	UNSUPPORTED	SUPPORTED
Bias Add	SUPPORTED	UNSUPPORTED	SUPPORTED
tensor concat	SUPPORTED	UNSUPPORTED	UNSUPPORTED
conv2d	SUPPORTED	SUPPORTED	SUPPORTED
cross entropy	SUPPORTED	UNSUPPORTED	SUPPORTED
div	SUPPORTED	UNSUPPORTED	SUPPORTED
dot	SUPPORTED	UNSUPPORTED	SUPPORTED
dropout	SUPPORTED	UNSUPPORTED	SUPPORTED
flatten	SUPPORTED	UNSUPPORTED	SUPPORTED
linear	SUPPORTED	SUPPORTED	SUPPORTED
log	SUPPORTED	UNSUPPORTED	SUPPORTED
matmul	SUPPORTED	UNSUPPORTED	SUPPORTED
MSE	SUPPORTED	UNSUPPORTED	SUPPORTED
pooling	UNSUPPORTED	SUPPORTED	SUPPORTED
pow	SUPPORTED	UNSUPPORTED	SUPPORTED
product	SUPPORTED	UNSUPPORTED	SUPPORTED
relu	SUPPORTED	UNSUPPORTED	SUPPORTED
sigmoid	SUPPORTED	UNSUPPORTED	SUPPORTED
softmax	SUPPORTED	UNSUPPORTED	SUPPORTED
sum	SUPPORTED	UNSUPPORTED	SUPPORTED
tanh	SUPPORTED	UNSUPPORTED	SUPPORTED
transpose	SUPPORTED	UNSUPPORTED	SUPPORTED
reduce sum	SUPPORTED	UNSUPPORTED	SUPPORTED
scalar product	SUPPORTED	UNSUPPORTED	SUPPORTED
tensor fill	SUPPORTED	UNSUPPORTED	SUPPORTED

DNN Programming Structure Exercises

Building MagmaDNN with CMake: CMakeLists.txt Options

When building MagmaDNN with CMake the build options are specified in CMakeLists.txt



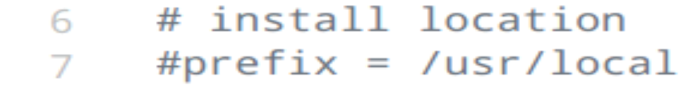
```
########################################
# MagmaDNN options
option(MAGMADNN_ENABLE_CUDA "Enable use of CUDA library and compilation of CUDA kernel" OFF)
option(MAGMADNN_ENABLE_MPI "Enable distributed memory routines using MPI" OFF)
option(MAGMADNN_ENABLE_OMP "Enable parallelization using OpenMP library" OFF)
option(MAGMADNN_ENABLE_MKLDNN "Enable use of MKLDNN library" OFF)
option(MAGMADNN_BUILD_MKLDNN "Enable build of MKLDNN from source" OFF)
option(MAGMADNN_BUILD_DOC "Generate documentation" OFF)
option(MAGMADNN_BUILD_EXAMPLES "Build MagmaDNN examples" ON)
option(MAGMADNN_BUILD_TESTS "Generate build files for unit tests" OFF)
option(MAGMADNN_BUILD_SHARED_LIBS "Build shared (.so, .dylib, .dll) libraries" ON)
```

As with CMake there is no need to specify library locations.

DNN Programming Structure Exercises

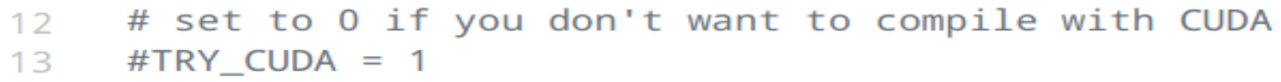
Building MagmaDNN: Makefile Options

The install location is specified as the prefix parameter



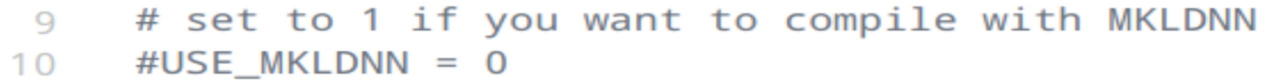
```
# install location
#prefix = /usr/local
```

When using the makefile build system MagmaDNN will defaults attempt to build using Cuda unless explicitly specified.



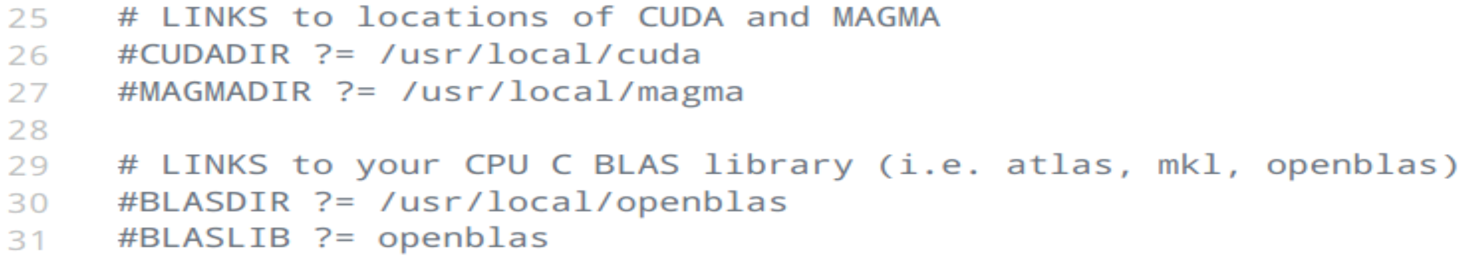
```
# set to 0 if you don't want to compile with CUDA
#TRY_CUDA = 1
```

MagmaDNN defaults to not compiling with MKLDNN.



```
# set to 1 if you want to compile with MKLDNN
#USE_MKLDNN = 0
```

To compile with Cuda, Magma, and a BLAS lib the library locations should be specified here.

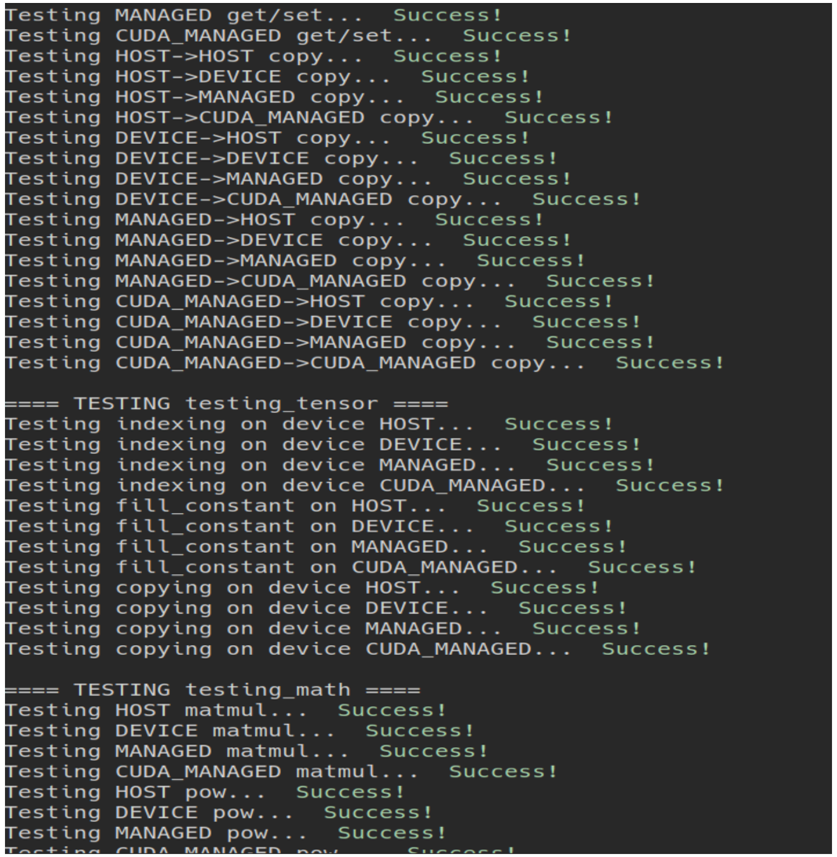


```
# LINKS to locations of CUDA and MAGMA
#CUDADIR ?= /usr/local/cuda
#MAGMADIR ?= /usr/local/magma

# LINKS to your CPU C BLAS library (i.e. atlas, mkl, openblas)
#BLASDIR ?= /usr/local/openblas
#BLASLIB ?= openblas
```

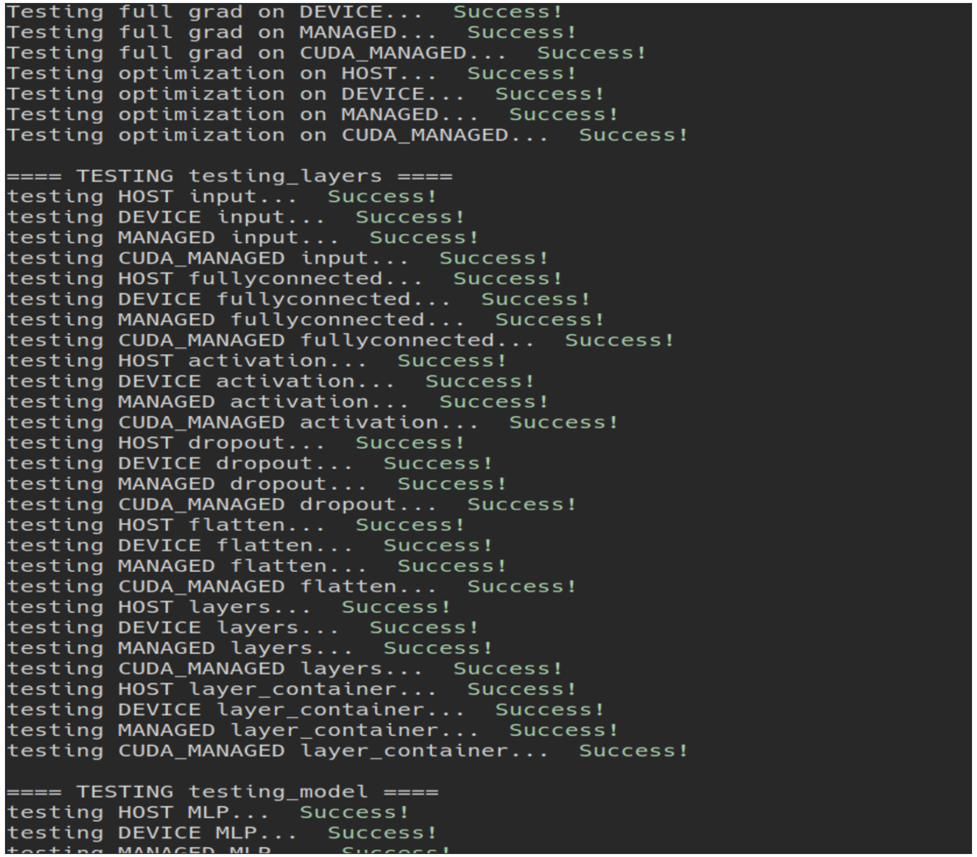
DNN Programming Structure Exercises

Building MagmaDNN: Running Tests

MagmaDNN has a set of tests you can run after compiling to ensure correctness of computations and that you are ready to build a neural network.



```
Testing MANAGED get/set...   Success!
Testing CUDA_MANAGED get/set...   Success!
Testing HOST->HOST copy...   Success!
Testing HOST->DEVICE copy...   Success!
Testing HOST->MANAGED copy...   Success!
Testing HOST->CUDA_MANAGED copy...   Success!
Testing DEVICE->HOST copy...   Success!
Testing DEVICE->DEVICE copy...   Success!
Testing DEVICE->MANAGED copy...   Success!
Testing DEVICE->CUDA_MANAGED copy...   Success!
Testing MANAGED->HOST copy...   Success!
Testing MANAGED->DEVICE copy...   Success!
Testing MANAGED->MANAGED copy...   Success!
Testing MANAGED->CUDA_MANAGED copy...   Success!
Testing CUDA_MANAGED->HOST copy...   Success!
Testing CUDA_MANAGED->DEVICE copy...   Success!
Testing CUDA_MANAGED->MANAGED copy...   Success!
Testing CUDA_MANAGED->CUDA_MANAGED copy...   Success!

==== TESTING testing_tensor ====
Testing indexing on device HOST...   Success!
Testing indexing on device DEVICE...   Success!
Testing indexing on device MANAGED...   Success!
Testing indexing on device CUDA_MANAGED...   Success!
Testing fill_constant on HOST...   Success!
Testing fill_constant on DEVICE...   Success!
Testing fill_constant on MANAGED...   Success!
Testing fill_constant on CUDA_MANAGED...   Success!
Testing copying on device HOST...   Success!
Testing copying on device DEVICE...   Success!
Testing copying on device MANAGED...   Success!
Testing copying on device CUDA_MANAGED...   Success!

==== TESTING testing_math ====
Testing HOST matmul...   Success!
Testing DEVICE matmul...   Success!
Testing MANAGED matmul...   Success!
Testing CUDA_MANAGED matmul...   Success!
Testing HOST pow...   Success!
Testing DEVICE pow...   Success!
Testing MANAGED pow...   Success!
Testing CUDA_MANAGED pow...   Success!
```



```
Testing full grad on DEVICE...   Success!
Testing full grad on MANAGED...   Success!
Testing full grad on CUDA_MANAGED...   Success!
Testing optimization on HOST...   Success!
Testing optimization on DEVICE...   Success!
Testing optimization on MANAGED...   Success!
Testing optimization on CUDA_MANAGED...   Success!

==== TESTING testing_layers ====
testing HOST input...   Success!
testing DEVICE input...   Success!
testing MANAGED input...   Success!
testing CUDA_MANAGED input...   Success!
testing HOST fullyconnected...   Success!
testing DEVICE fullyconnected...   Success!
testing MANAGED fullyconnected...   Success!
testing CUDA_MANAGED fullyconnected...   Success!
testing HOST activation...   Success!
testing DEVICE activation...   Success!
testing MANAGED activation...   Success!
testing CUDA_MANAGED activation...   Success!
testing HOST dropout...   Success!
testing DEVICE dropout...   Success!
testing MANAGED dropout...   Success!
testing CUDA_MANAGED dropout...   Success!
testing HOST flatten...   Success!
testing DEVICE flatten...   Success!
testing MANAGED flatten...   Success!
testing CUDA_MANAGED flatten...   Success!
testing HOST layers...   Success!
testing DEVICE layers...   Success!
testing MANAGED layers...   Success!
testing CUDA_MANAGED layers...   Success!
testing HOST layer_container...   Success!
testing DEVICE layer_container...   Success!
testing MANAGED layer_container...   Success!
testing CUDA_MANAGED layer_container...   Success!

==== TESTING testing_model ====
testing HOST MLP...   Success!
testing DEVICE MLP...   Success!
testing MANAGED MLP...   Success!
```

DNN Programming Structure Exercises

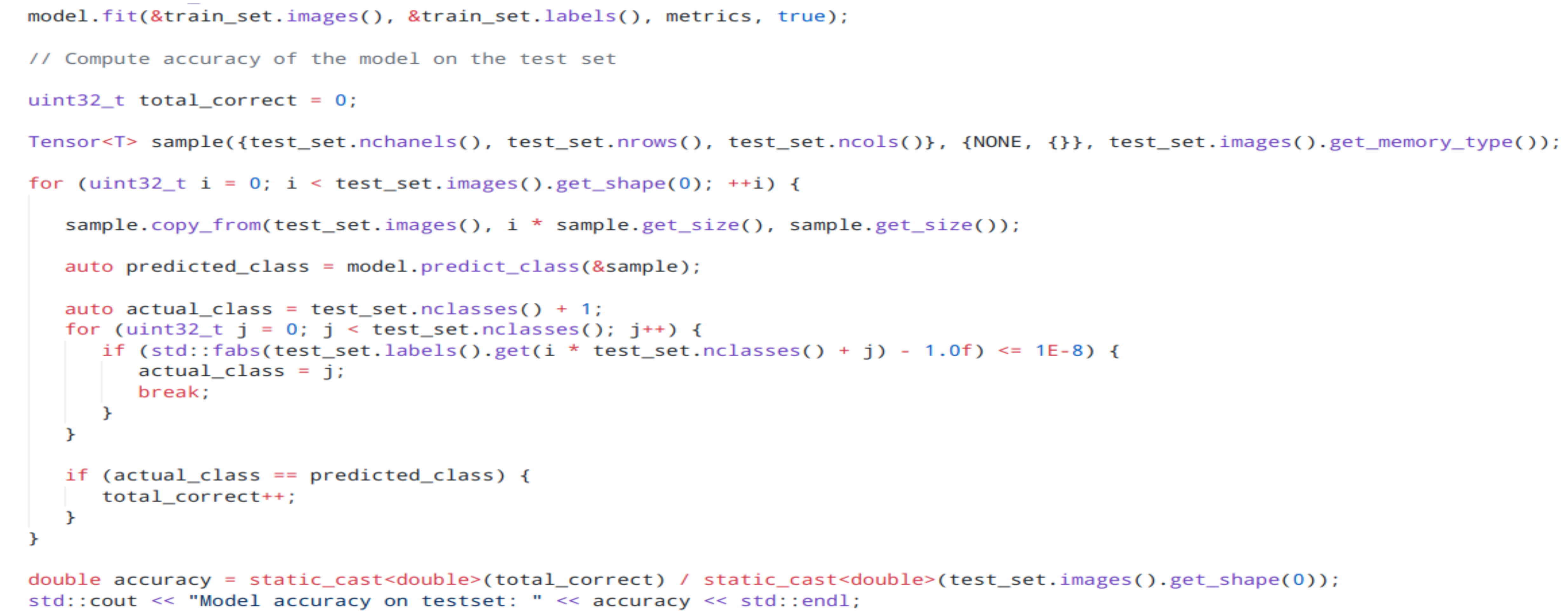
Compiling and Training the Network

1. **Install MagmaDNN**
2. **Include MagmaDNN headers in your cpp file**
3. **Link MagmaDNN and any external libraries**

```
g++ conv.cpp -I/opt/cuda/include/ -I/usr/local/magma/include/ -L/opt/cuda/lib/ -L/usr/local/magma/lib -lmagmadnn -lcublas -
lcudnn -lmagma -o train

./train

Preparing to read 60000 images with size 28 x 28 ...
finished reading images.
Preparing to read 60000 labels with 10 classes ...
finished reading labels.
Epoch (0/20): accuracy=0.93 loss=0.222 time=1
Epoch (1/20): accuracy=0.9534 loss=0.1458 time=2
Epoch (2/20): accuracy=0.9638 loss=0.1126 time=3
Epoch (3/20): accuracy=0.97 loss=0.09276 time=4
...
Epoch (19/20): accuracy=0.9915 loss=0.0241 time=19
Final Training Metrics: accuracy=0.9915 loss=0.0241 time=19
```

DNN Programming Structure Exercises

Testing a Trained Model with a Test Set



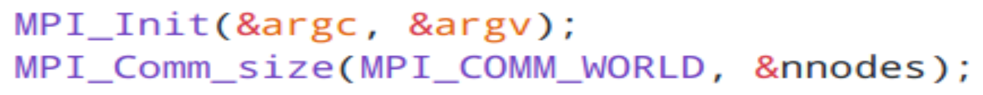
```
model.fit(&train_set.images(), &train_set.labels(), metrics, true);

// Compute accuracy of the model on the test set

uint32_t total_correct = 0;

Tensor<T> sample({test_set.nchanels(), test_set.nrows(), test_set.ncols()}, {NONE, {}}, test_set.images().get_memory_type());

for (uint32_t i = 0; i < test_set.images().get_shape(0); ++i) {

   sample.copy_from(test_set.images(), i * sample.get_size(), sample.get_size());

   auto predicted_class = model.predict_class(&sample);

   auto actual_class = test_set.nclasses() + 1;
   for (uint32_t j = 0; j < test_set.nclasses(); j++) {
      if (std::fabs(test_set.labels().get(i * test_set.nclasses() + j) - 1.0f) <= 1E-8) {
         actual_class = j;
         break;
      }
   }

   if (actual_class == predicted_class) {
      total_correct++;
   }
}

double accuracy = static_cast<double>(total_correct) / static_cast<double>(test_set.images().get_shape(0));
std::cout << "Model accuracy on testset: " << accuracy << std::endl;
```

DNN Programming Structure Exercises

Designing a Distributed Network

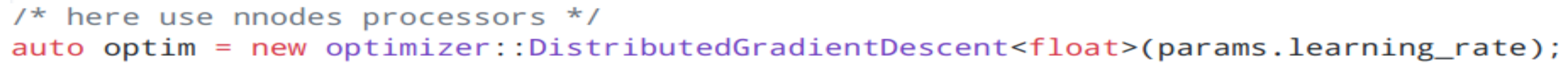
MagmaDNN has built-in support to perform distributed training using MPI by averaging the neural network weights across each node after each batch for data parallelism.

MPI_Init is called before magmadnn_init()

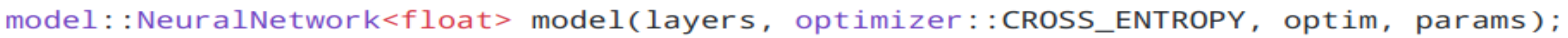


```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nnodes);
```

After the network architecture is defined, an optimizer is created using the DistributedGradientDescent class and passed to the model.

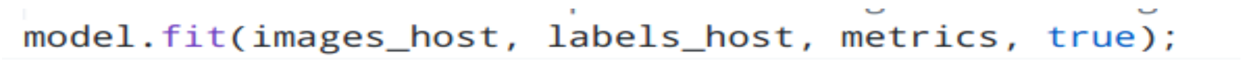


```
/* here use nnodes processors */
auto optim = new optimizer::DistributedGradientDescent<float>(params.learning_rate);
```



```
model::NeuralNetwork<float> model(layers, optimizer::CROSS_ENTROPY, optim, params);
```

The fit member function is called and training begins across all nodes.



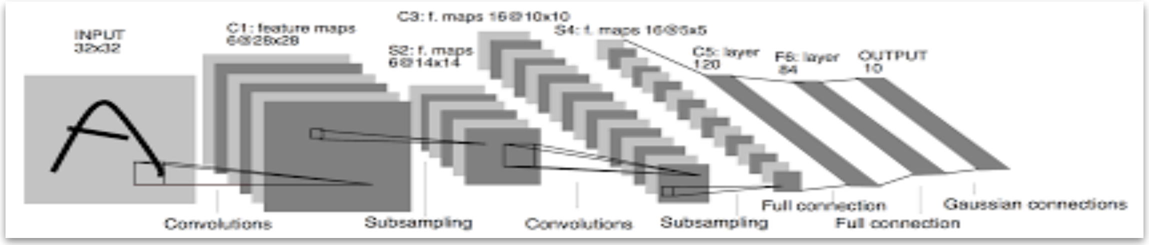
```
model.fit(images_host, labels_host, metrics, true);
```

DNN Programming Structure Examples

LeNet5

One of the examples included in MagmaDNN is a LeNet network. The image below shows how the layers are defined in MagmaDNN.

```
auto conv2d1 = layer::conv2d(input->out(), {5, 5}, 32, {0, 0}, {1, 1}, {1, 1});
auto act1 = layer::activation(conv2d1->out(), layer::TANH);
auto pool1 = layer::pooling(act1->out(), {2, 2}, {0, 0}, {2, 2}, AVERAGE_POOL);

auto conv2d2 = layer::conv2d(pool1->out(), {5, 5}, 32, {0, 0}, {1, 1}, {1, 1});
auto act2 = layer::activation(conv2d2->out(), layer::TANH);
auto pool2 = layer::pooling(act2->out(), {2, 2}, {0, 0}, {2, 2}, AVERAGE_POOL);

auto flatten = layer::flatten(pool2->out());

auto fc1 = layer::fullyconnected(flatten->out(), 120, true);
auto act3 = layer::activation(fc1->out(), layer::TANH);

auto fc2 = layer::fullyconnected(act3->out(), 84, true);
auto act4 = layer::activation(fc2->out(), layer::TANH);

auto fc3 = layer::fullyconnected(act4->out(), train_set.nclasses(), false);
auto act5 = layer::activation(fc3->out(), layer::SOFTMAX);
```

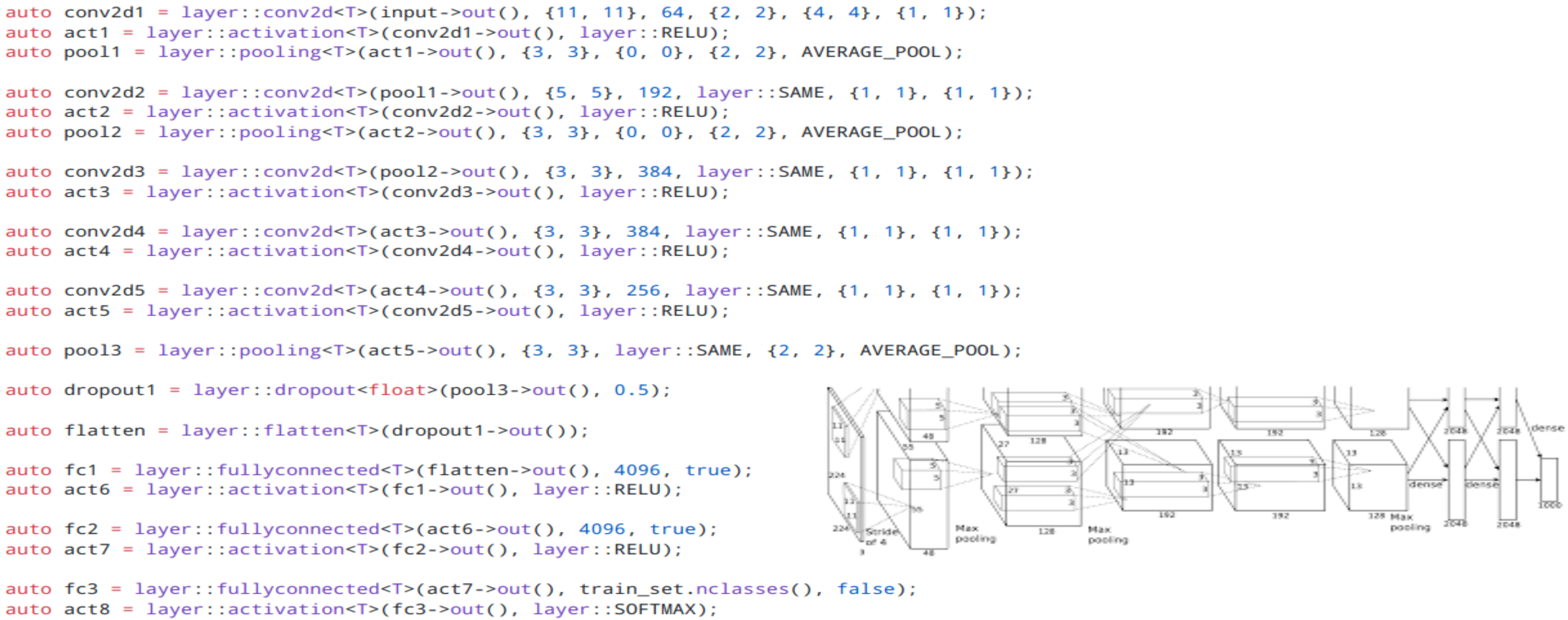


http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

DNN Programming Structure Examples

AlexNet

Another example network in MagmaDNN is AlexNet. This network trains on the CIFAR100 dataset.



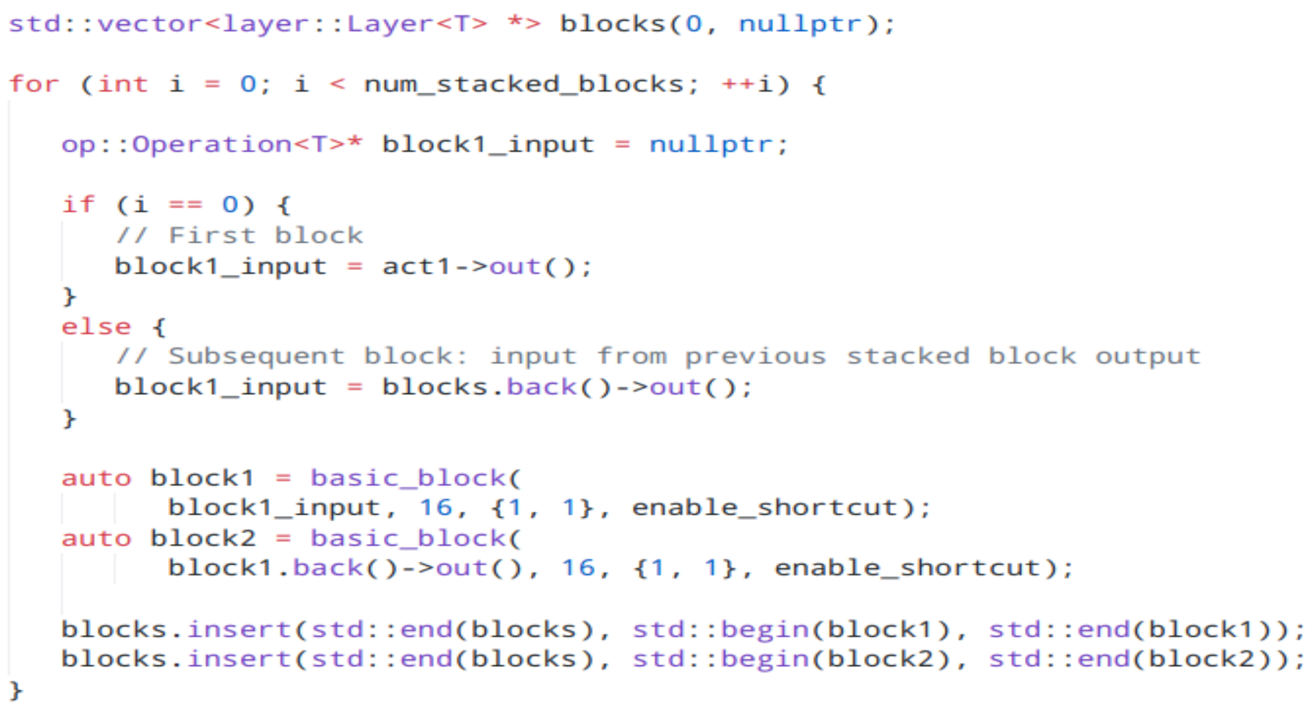
```
auto conv2d1 = layer::conv2d<T>(input->out(), {11, 11}, 64, {2, 2}, {4, 4}, {1, 1});
auto act1 = layer::activation<T>(conv2d1->out(), layer::RELU);
auto pool1 = layer::pooling<T>(act1->out(), {3, 3}, {0, 0}, {2, 2}, AVERAGE_POOL);

auto conv2d2 = layer::conv2d<T>(pool1->out(), {5, 5}, 192, layer::SAME, {1, 1}, {1, 1});
auto act2 = layer::activation<T>(conv2d2->out(), layer::RELU);
auto pool2 = layer::pooling<T>(act2->out(), {3, 3}, {0, 0}, {2, 2}, AVERAGE_POOL);

auto conv2d3 = layer::conv2d<T>(pool2->out(), {3, 3}, 384, layer::SAME, {1, 1}, {1, 1});
auto act3 = layer::activation<T>(conv2d3->out(), layer::RELU);

auto conv2d4 = layer::conv2d<T>(act3->out(), {3, 3}, 384, layer::SAME, {1, 1}, {1, 1});
auto act4 = layer::activation<T>(conv2d4->out(), layer::RELU);

auto conv2d5 = layer::conv2d<T>(act4->out(), {3, 3}, 256, layer::SAME, {1, 1}, {1, 1});
auto act5 = layer::activation<T>(conv2d5->out(), layer::RELU);

auto pool3 = layer::pooling<T>(act5->out(), {3, 3}, layer::SAME, {2, 2}, AVERAGE_POOL);

auto dropout1 = layer::dropout<float>(pool3->out(), 0.5);

auto flatten = layer::flatten<T>(dropout1->out());

auto fc1 = layer::fullyconnected<T>(flatten->out(), 4096, true);
auto act6 = layer::activation<T>(fc1->out(), layer::RELU);

auto fc2 = layer::fullyconnected<T>(act6->out(), 4096, true);
auto act7 = layer::activation<T>(fc2->out(), layer::RELU);

auto fc3 = layer::fullyconnected<T>(act7->out(), train_set.nclasses(), false);
auto act8 = layer::activation<T>(fc3->out(), layer::SOFTMAX);
```

https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

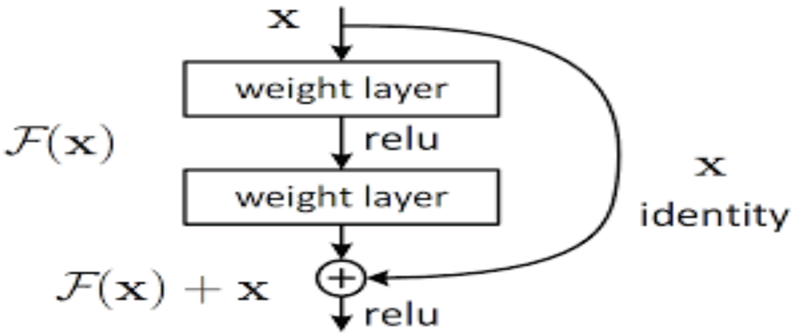
DNN Programming Structure Examples

ResNet

The ResNet example in MagmaDNN uses basic blocks to create the network.



```
std::vector<layer::Layer<T> *> blocks(0, nullptr);

for (int i = 0; i < num_stacked_blocks; ++i) {

    op::Operation<T>* block1_input = nullptr;

    if (i == 0) {
        // First block
        block1_input = act1->out();
    }
    else {
        // Subsequent block: input from previous stacked block output
        block1_input = blocks.back()->out();
    }

    auto block1 = basic_block(
            block1_input, 16, {1, 1}, enable_shortcut);
    auto block2 = basic_block(
            block1.back()->out(), 16, {1, 1}, enable_shortcut);

    blocks.insert(std::end(blocks), std::begin(block1), std::end(block1));
    blocks.insert(std::end(blocks), std::begin(block2), std::end(block2));
}
```

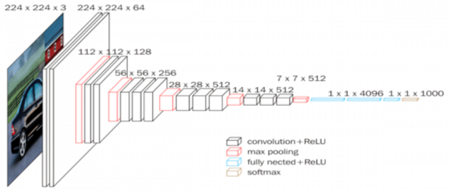


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

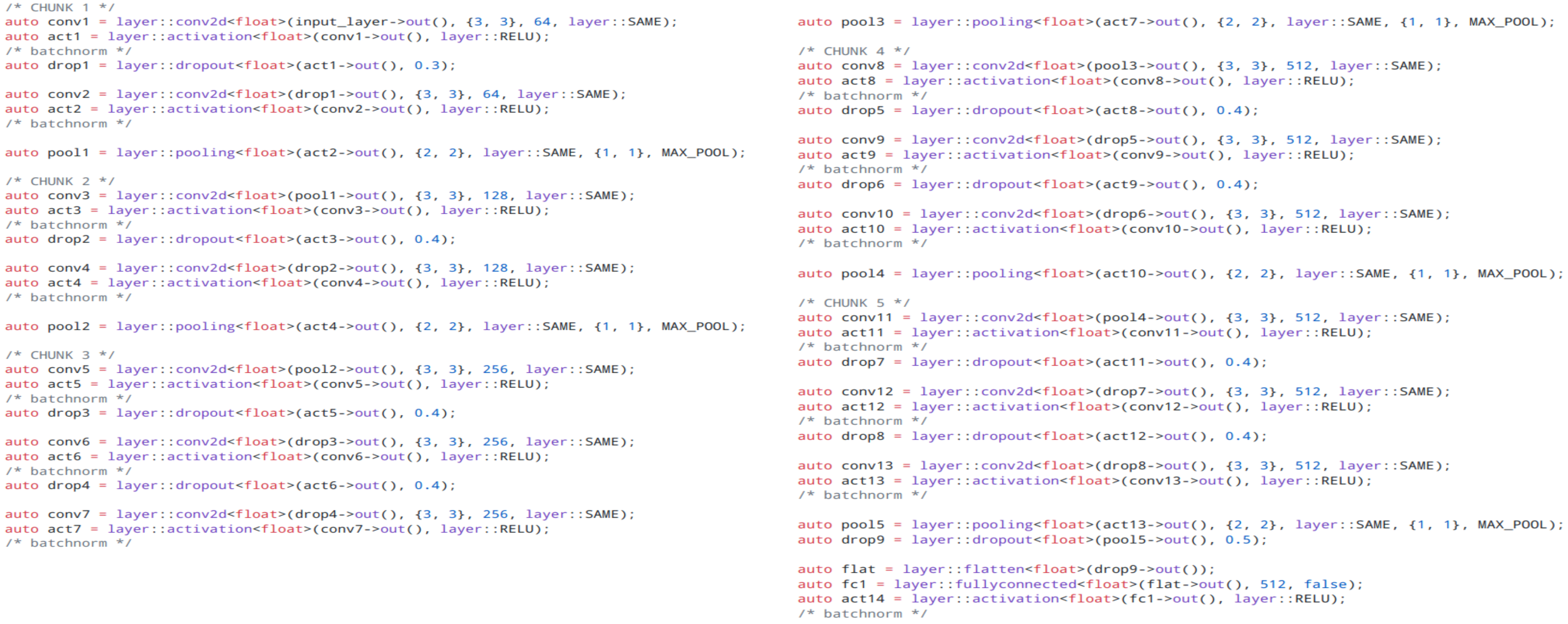
image: https://neurohive.io/en/popular-networks/resnet/

DNN Programming Structure Examples

VGG16



The VGG16 example in MagmaDNN layer structure is outlined below.



```
/* CHUNK 1 */
auto conv1 = layer::conv2d<float>(input_layer->out(), {3, 3}, 64, layer::SAME);
auto act1 = layer::activation<float>(conv1->out(), layer::RELU);
/* batchnorm */
auto drop1 = layer::dropout<float>(act1->out(), 0.3);

auto conv2 = layer::conv2d<float>(drop1->out(), {3, 3}, 64, layer::SAME);
auto act2 = layer::activation<float>(conv2->out(), layer::RELU);
/* batchnorm */

auto pool1 = layer::pooling<float>(act2->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);

/* CHUNK 2 */
auto conv3 = layer::conv2d<float>(pool1->out(), {3, 3}, 128, layer::SAME);
auto act3 = layer::activation<float>(conv3->out(), layer::RELU);
/* batchnorm */
auto drop2 = layer::dropout<float>(act3->out(), 0.4);

auto conv4 = layer::conv2d<float>(drop2->out(), {3, 3}, 128, layer::SAME);
auto act4 = layer::activation<float>(conv4->out(), layer::RELU);
/* batchnorm */

auto pool2 = layer::pooling<float>(act4->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);

/* CHUNK 3 */
auto conv5 = layer::conv2d<float>(pool2->out(), {3, 3}, 256, layer::SAME);
auto act5 = layer::activation<float>(conv5->out(), layer::RELU);
/* batchnorm */
auto drop3 = layer::dropout<float>(act5->out(), 0.4);

auto conv6 = layer::conv2d<float>(drop3->out(), {3, 3}, 256, layer::SAME);
auto act6 = layer::activation<float>(conv6->out(), layer::RELU);
/* batchnorm */
auto drop4 = layer::dropout<float>(act6->out(), 0.4);

auto conv7 = layer::conv2d<float>(drop4->out(), {3, 3}, 256, layer::SAME);
auto act7 = layer::activation<float>(conv7->out(), layer::RELU);
/* batchnorm */
```

```
auto pool3 = layer::pooling<float>(act7->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);

/* CHUNK 4 */
auto conv8 = layer::conv2d<float>(pool3->out(), {3, 3}, 512, layer::SAME);
auto act8 = layer::activation<float>(conv8->out(), layer::RELU);
/* batchnorm */
auto drop5 = layer::dropout<float>(act8->out(), 0.4);

auto conv9 = layer::conv2d<float>(drop5->out(), {3, 3}, 512, layer::SAME);
auto act9 = layer::activation<float>(conv9->out(), layer::RELU);
/* batchnorm */
auto drop6 = layer::dropout<float>(act9->out(), 0.4);

auto conv10 = layer::conv2d<float>(drop6->out(), {3, 3}, 512, layer::SAME);
auto act10 = layer::activation<float>(conv10->out(), layer::RELU);
/* batchnorm */

auto pool4 = layer::pooling<float>(act10->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);

/* CHUNK 5 */
auto conv11 = layer::conv2d<float>(pool4->out(), {3, 3}, 512, layer::SAME);
auto act11 = layer::activation<float>(conv11->out(), layer::RELU);
/* batchnorm */
auto drop7 = layer::dropout<float>(act11->out(), 0.4);

auto conv12 = layer::conv2d<float>(drop7->out(), {3, 3}, 512, layer::SAME);
auto act12 = layer::activation<float>(conv12->out(), layer::RELU);
/* batchnorm */
auto drop8 = layer::dropout<float>(act12->out(), 0.4);

auto conv13 = layer::conv2d<float>(drop8->out(), {3, 3}, 512, layer::SAME);
auto act13 = layer::activation<float>(conv13->out(), layer::RELU);
/* batchnorm */

auto pool5 = layer::pooling<float>(act13->out(), {2, 2}, layer::SAME, {1, 1}, MAX_POOL);
auto drop9 = layer::dropout<float>(pool5->out(), 0.5);

auto flat = layer::flatten<float>(drop9->out());
auto fc1 = layer::fullyconnected<float>(flat->out(), 512, false);
auto act14 = layer::activation<float>(fc1->out(), layer::RELU);
/* batchnorm */
```

DNN Programming

MagmaDNN 1.2

Introduction | Examples | Core DNN Concepts

Development Version

https://github.com/MagmaDNN/magmadnn

Release Version

https://bitbucket.org/icl/magmadnn/src/master/