Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks

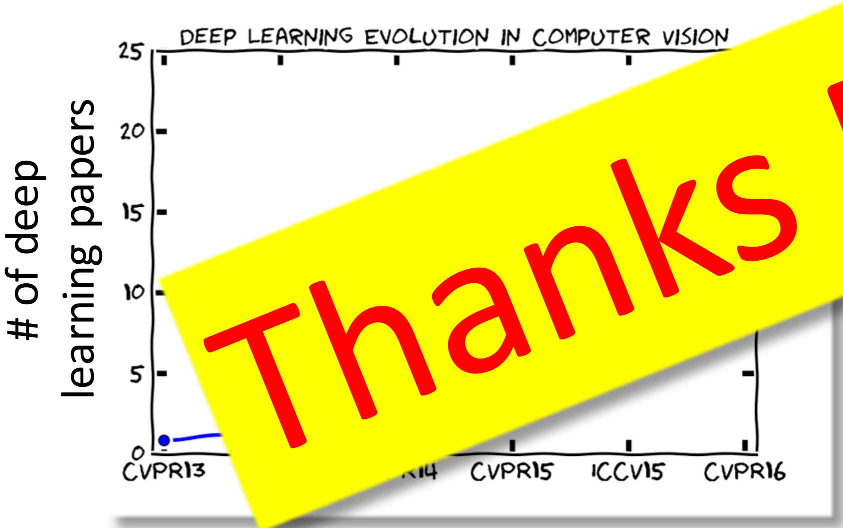
Leonard Truong[Φ, τ], Rajkishore Barik[τ], Ehsan Totoni[τ], Hai Liu[τ], Chick Markley[Φ], Armando Fox[Φ], Tatiana Shpeisman[τ]

[τ]Intel Labs, [Φ]UC Berkeley

Deep Learning Research is Thriving

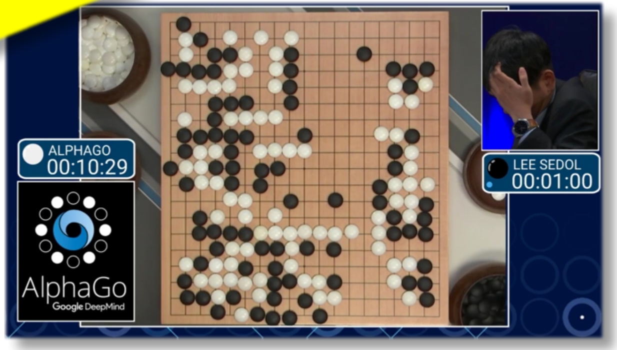
Thanks Ben Zorn

Academia



http://jponttuset.github.io/xkcd-deep-learning/

bing

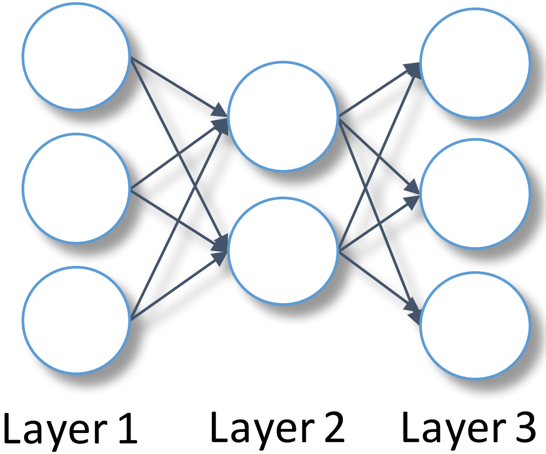


Deep Learning 101

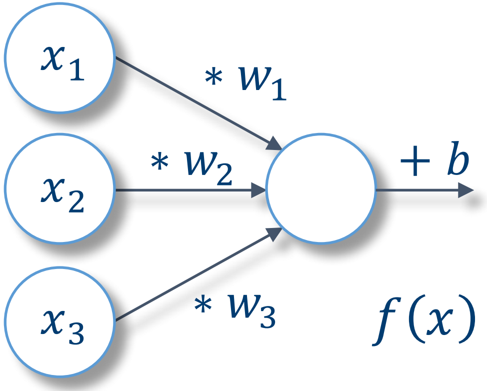
Neural networks: family of biologically inspired, machine-learning models

Neurons are organized into **layers**

DNN Architecture/Model: a specific configuration of layers



WeightedNeuron: fundamental building block of neural networks



$$f(x) = b + \sum_i w_i * x_i$$

Parameters $\{w_1, w_2, w_3, b\}$ are learned

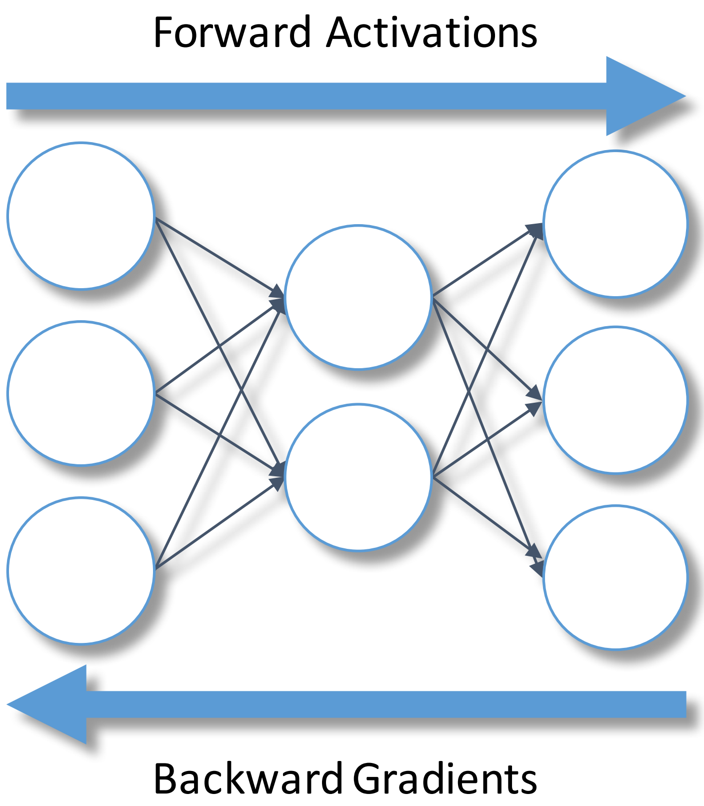
Deep Learning 101

Training neural networks

For each training sample

- **Forward propagation**
- **Back propagation**
- Update parameters

Typically process the entire training data set multiple times



Deep Learning Research

New Layers
New Network Architectures

What's the best programming environment for deep learning research?

Dropout

AlexNet

Maxout

Inception Architectures

Parametric Rectified Linear Unit

Batch Normalization

Residual Learning Architectures

2013 2014 2015 2016

A very incomplete list of deep learning advancements

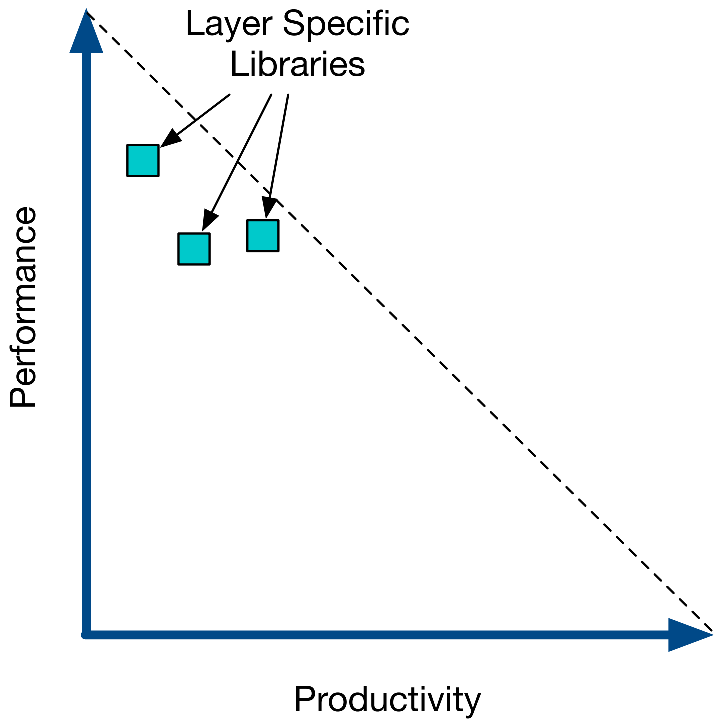
Can take **days to weeks** to evaluate a new idea

- AlexNet 5-6 days [Krizhevsky et al. NIPS '12]
- VGG 2-3 weeks [Simonyan et al. ICLR '15]

Deep Learning Software Landscape

Layer Specific Libraries

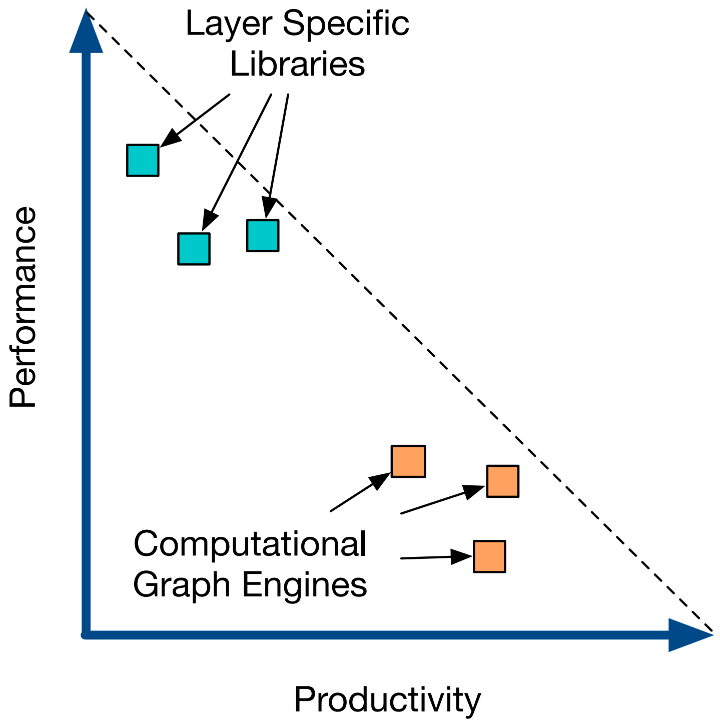
- Provide a function for each layer
- **Examples**: cuDNN, Caffe, NNPACK, …
- Good performance
- Hard to program
- **Cannot fuse across layers**



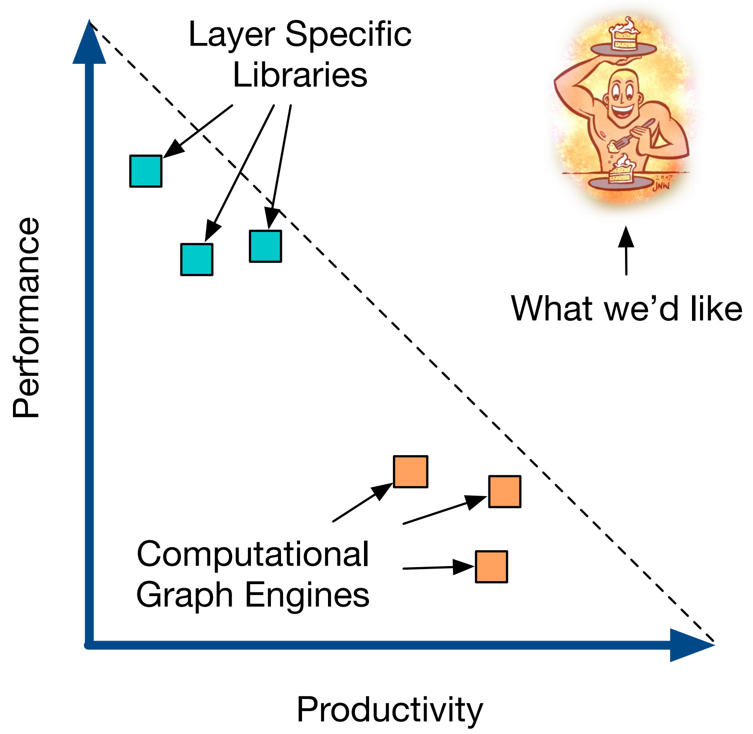
Deep Learning Software Landscape

Computational Graph Engines

- DNNs can be expressed as graphs of general operations
- **Examples**: Theano, CNTK, TensorFlow, …
- Easier to program
- Sacrifice performance
- **Many rely on bindings to cuDNN**



Deep Learning Software Landscape



Introducing Latte

DSL

- Language for describing DNNs as a system of interconnected neurons

Compiler

- Constructs an implicit data-flow graph from user description
- Synthesizes and optimizes an implementation of the data-flow graph

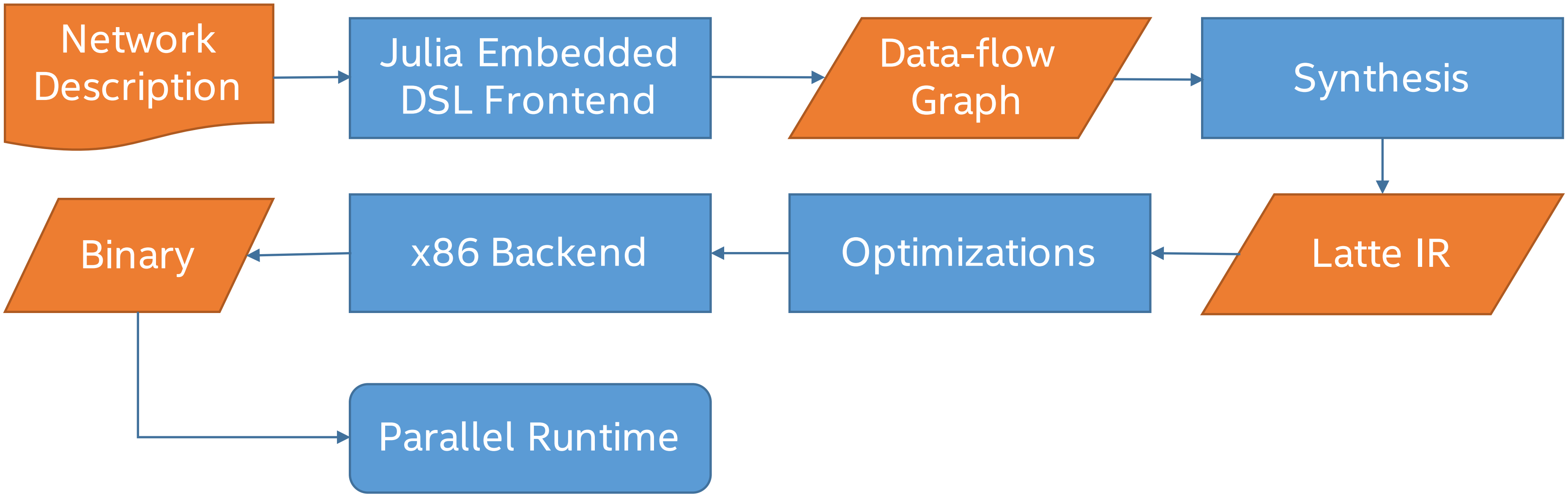
Runtime

- Supports distributed memory parallelism for data-parallel training

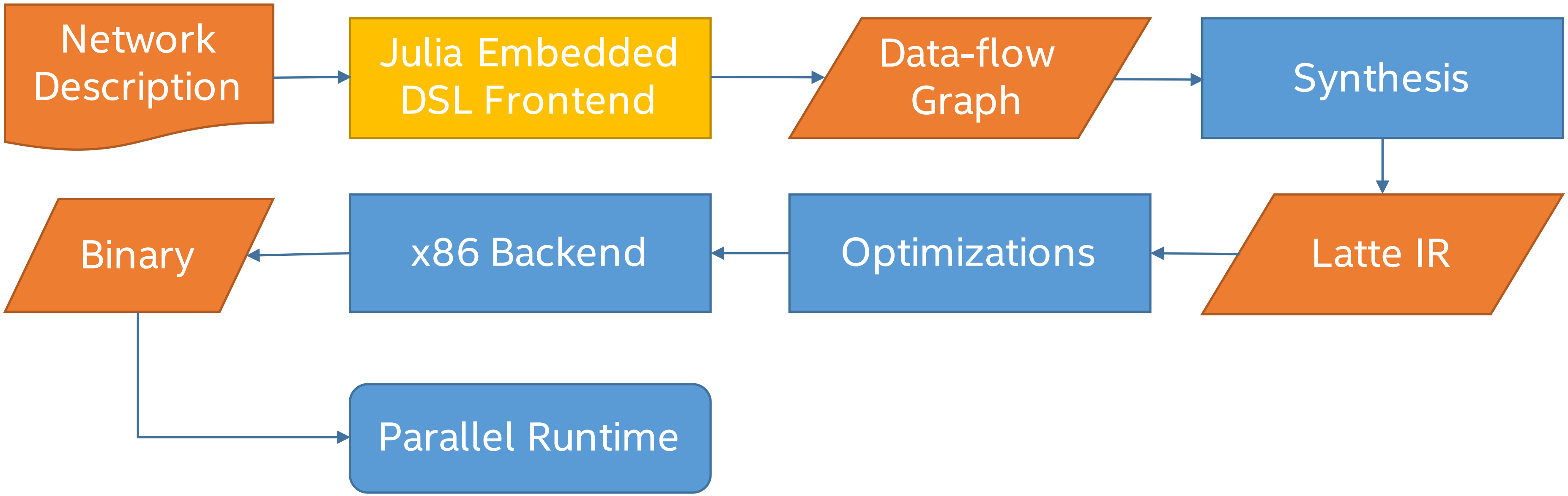
Competitive Performance

- 3-6x speedup over Caffe (C++/MKL) on a single node

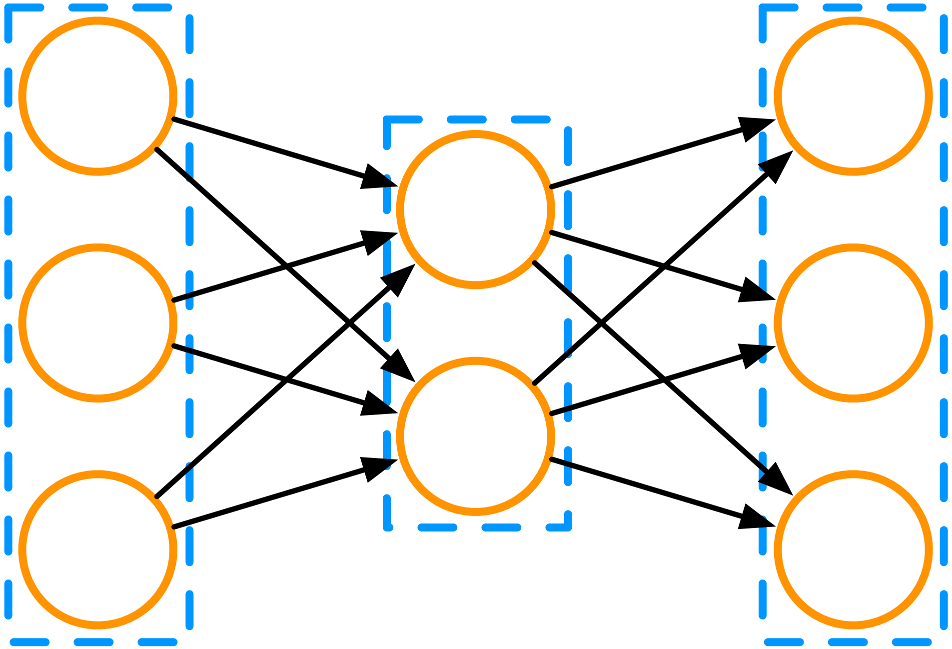
Latte System Diagram



Latte DSL Frontend



Latte DSL: Introduction



Language Constructs

Ensemble

Neuron

Connection

Embedded in julia

Latte DSL: Neuron

Neuron – Primitive, abstract data-type

```
type Neuron
    value    :: Float32
    ∇        :: Float32
    inputs   :: Vector{Float32}
    ∇inputs  :: Vector{Float32}
end
```

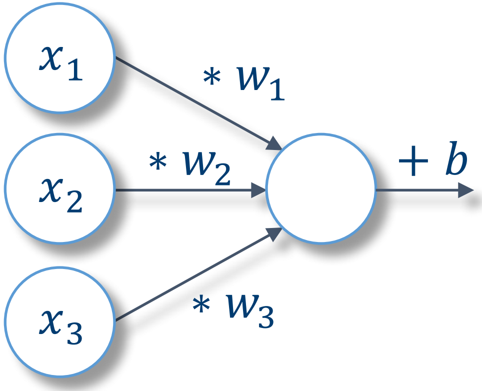
User can

- Define additional fields for internal state (optional)
- Implement ***forward*** and ***backward*** functions (required)

Latte DSL: Neuron Example

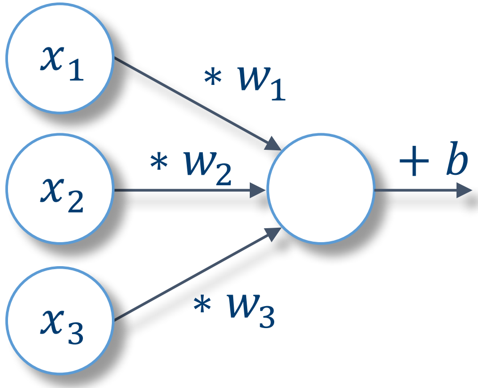
WeightedNeuron adds additional fields for learning weights and bias values

```
@neuron type WeightedNeuron <: Neuron
    weights  :: Vector{Float32}
    ∇weights :: Vector{Float32}
    bias     :: Float32
    ∇bias    :: Float32
end
```



- weights holds $\{w_1, w_2, w_3\}$, ∇weights holds gradients
- bias holds $\{b\}$, ∇bias holds gradient
- **Important**: Inherits value, ∇, inputs, and ∇inputs from Neuron

Latte DSL: Neuron Forward Example



The forward propagation of a **WeightedNeuron**

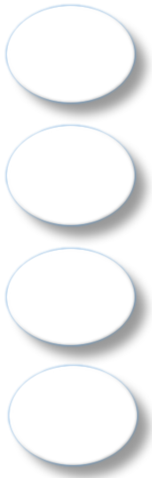
$$f(x) = b + \sum_i w_i * x_i$$

```
@neuron forward(neuron::WeightedNeuron) do
    for i in 1:length(neuron.inputs)
        neuron.value += neuron.weights[i] * neuron.inputs[i]
    end
    neuron.value += neuron.bias
end
```

Latte DSL: Ensemble

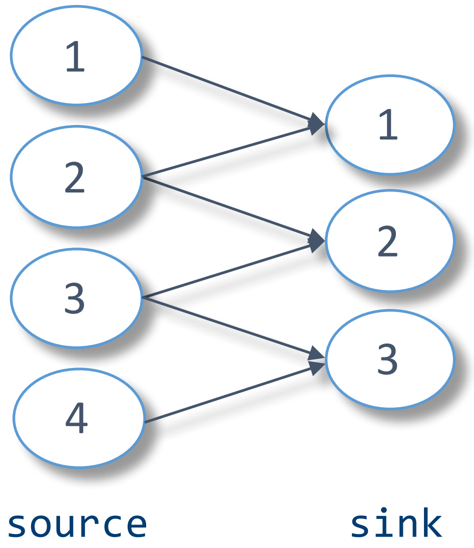
An **Ensemble** is an **N**-dimensional array of a Neuron subtype **T**

```
type Ensemble{T <: Neuron, N}
    name         :: Symbol
    neurons      :: Array{T,N}
    connections  :: Vector{Connection}
    ...
end
```



Latte DSL: Connections

Connections between a *source* ensemble and a *sink ensemble* are defined using a **mapping function**




```
add_connections(net, source, sink, (i) -> (i:i+1,))
```

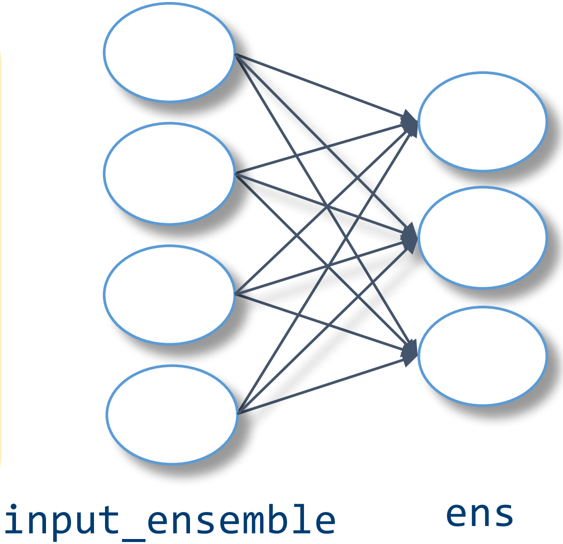
3 3:4

Bringing it all together: Fully Connected Layer

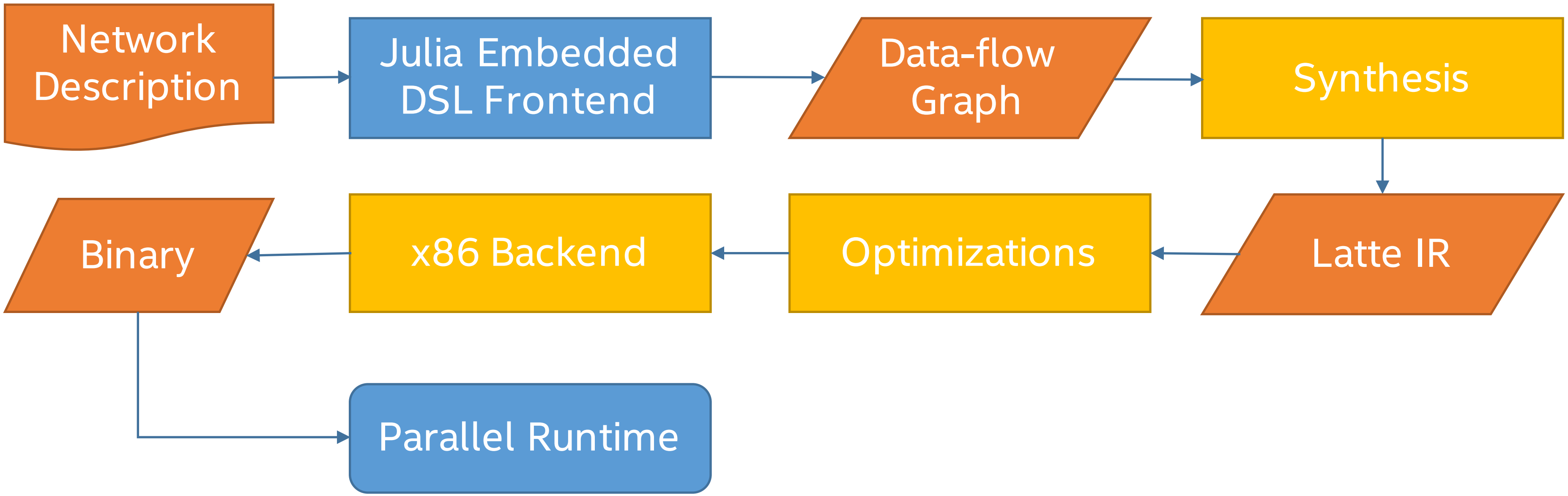
```
# Create a 1-D array of `num_outputs` WeightedNeurons
neurons = [WeightedNeuron(weights[:, i], ∇weights[:, i],
                          bias[:, i]   , ∇bias[:, i])
           for i in 1:num_outputs]

# Construct the ensemble
ens = Ensemble(net, name, neurons)
```

```
# Connect each neuron in input_ensemble
add_connections(net, input_ensemble, ens,
                (i) -> (1:d for d in size(input_ensemble))
```



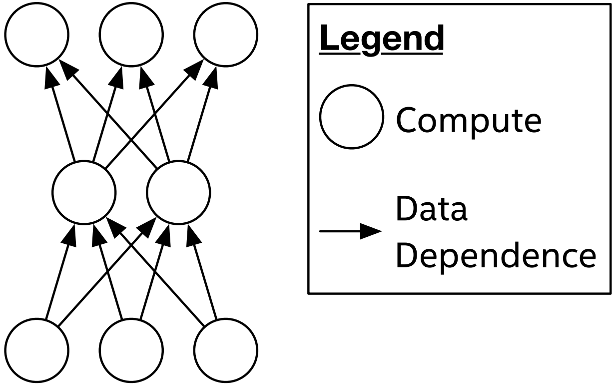
Latte Compiler



Latte Compiler: Synthesis

User implicitly describes a **data-flow graph (DFG)**

- Stored implicitly by mapping functions
- Ensembles provide a natural partitioning of the DFG



Data-flow Synthesis

- Copy output values of connected neurons into an input buffer for an ensemble

Compute Synthesis

- Perform the computation of neurons in an ensemble

Distributed Memory Communication

- Asynchronously synchronize gradients

Latte Compiler: Optimizations

Cross Layer Fusion (Covered in this talk)

- Use dependence information from data-flow graph to fuse loops across layers

Shared Variable Optimization (See paper)

- Compiler analysis discovers values that are shared between neurons

Library Kernel Pattern Matching (See paper)

- Replace matrix-multiplication loop nest patterns with MKL call

General Loop Optimizations (See paper)

- Tiling, Fusion

Latte Compiler: Cross Layer Fusion Example

Conv Layer

ReLU Layer

Pooling Layer

```
for y_tile in 1:TILE_SIZE:height
  gemm('T', 'N', TILE_SIZE*width, n_filters, n_inputs,
       conv1input[n], conv1weights, conv1[n])
for y_tile in 1:TILE_SIZE:height
  for c = 1:n_filters,
      y = y_tile:y_tile+TILE_SIZE,
      x = 1:width
    conv1[x, y, c] = max(conv1[x, y, c], 0.0)
for y_tile in 1:TILE_SIZE:height/2
  for c = 1:n_filters,
      y = y = y_tile:y_tile+TILE_SIZE,
      x = 1:width
    for p = 1:2, q = 1:2
      poolinput[p*2+q, x, y, c] = conv1[x+q, y+p, c]
    maxval = -Inf
    for i = 1:pool_size
      maxval = max(poolinput[i, x, y, c], maxval)
    pool1[x, y, c] = maxval
```

Easily Fused

Different Loop Lengths

Possible Fusion Preventing Dependency

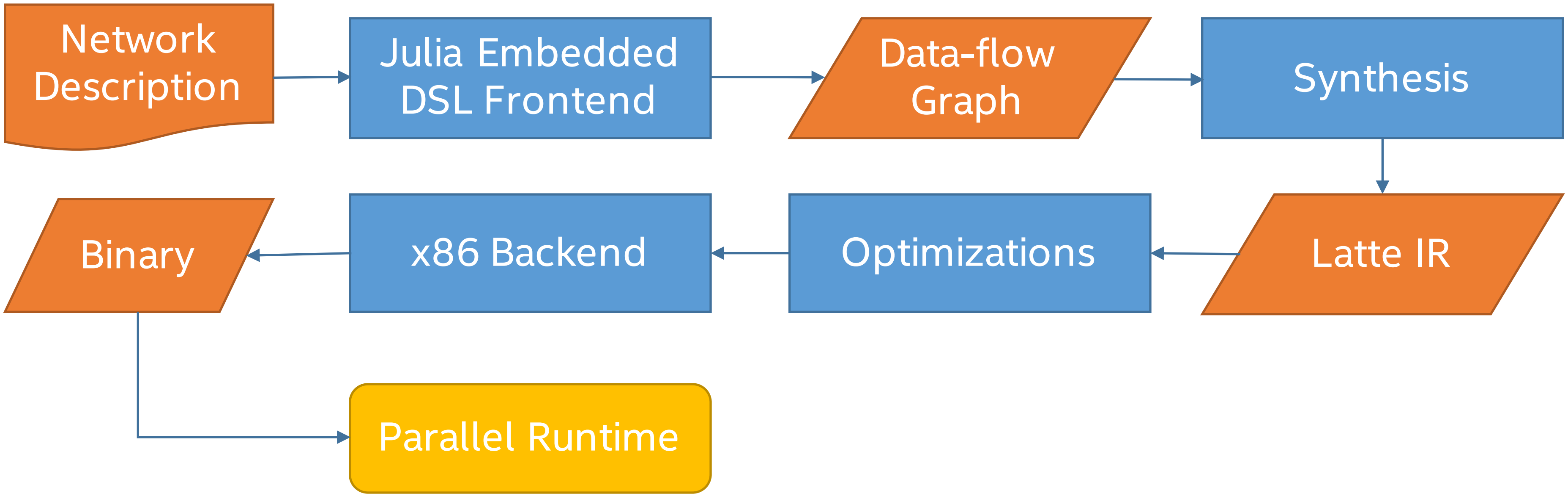
Latte Compiler: Cross Layer Fusion Example

Key Takeaways

- Fusing across layers is only possible in a dynamic compiler, **static libraries cannot do this**
- Dependence information already provided by data-flow graph

```
for y_tile in 1:TILE_SIZE:height/2
  gemm('T', 'N', TILE_SIZE*2*width,
       n_filters, n_inputs, conv1input[n],
       conv1weights, conv1[n])
  for c = 1:n_filters,
     for y = y_tile:y_tile+TILE_SIZE*2,
       x = 1:width
       conv1[x, y, c] = max(conv1[x, y, c],
                            0.0)
     for y = y_tile:y_tile+TILE_SIZE,
       x = 1:width
       maxval = -Inf
       for p = 1:2, q = 1:2
         maxval = max(conv1[x+q, y+p, c],
                      maxval)
       pool1[x, y, c] = maxval
```

Latte Runtime



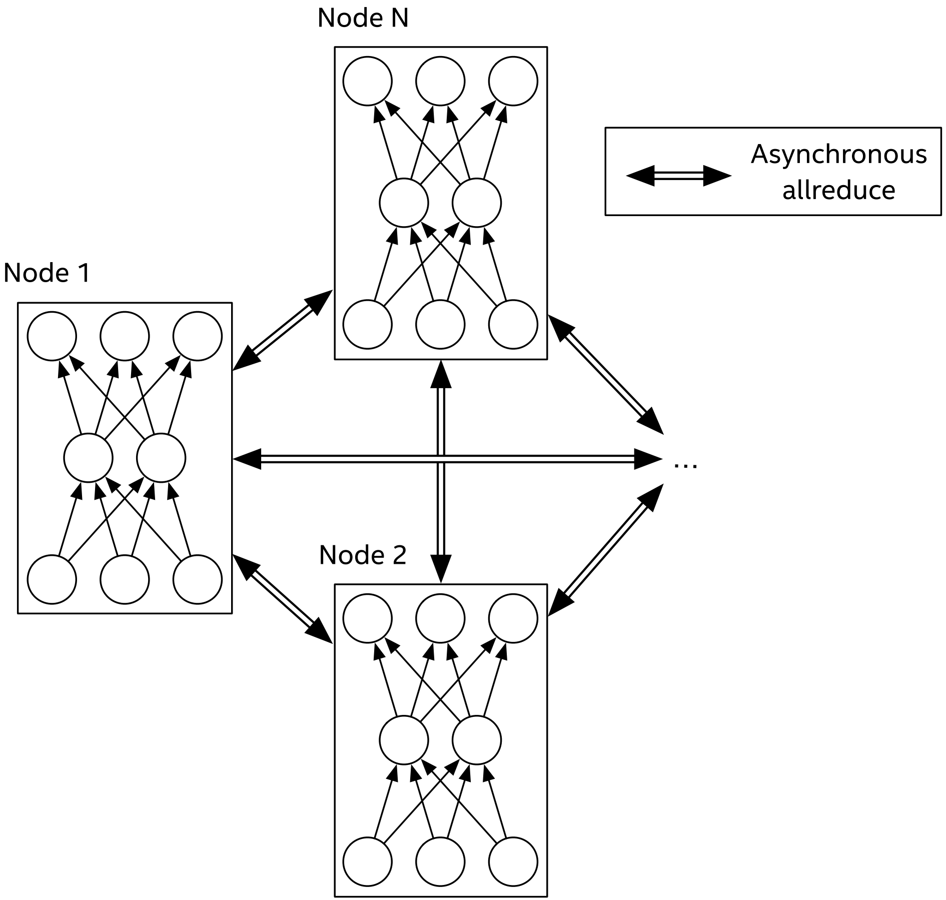
Latte Runtime

Manages data-parallel training on distributed memory systems

- Gradients reduction is overlapped with forward/backward phases

Supports automatic offloading to accelerators (Xeon Phi)

- Data movement to and from accelerator is overlapped with host compute



Evaluating Latte

Compared performance against open-source frameworks

- Caffe [C++/MKL], Mocha.jl [Julia/MKL] (see paper for Mocha results)
- **Training throughput** used as metric

Platforms in this talk:

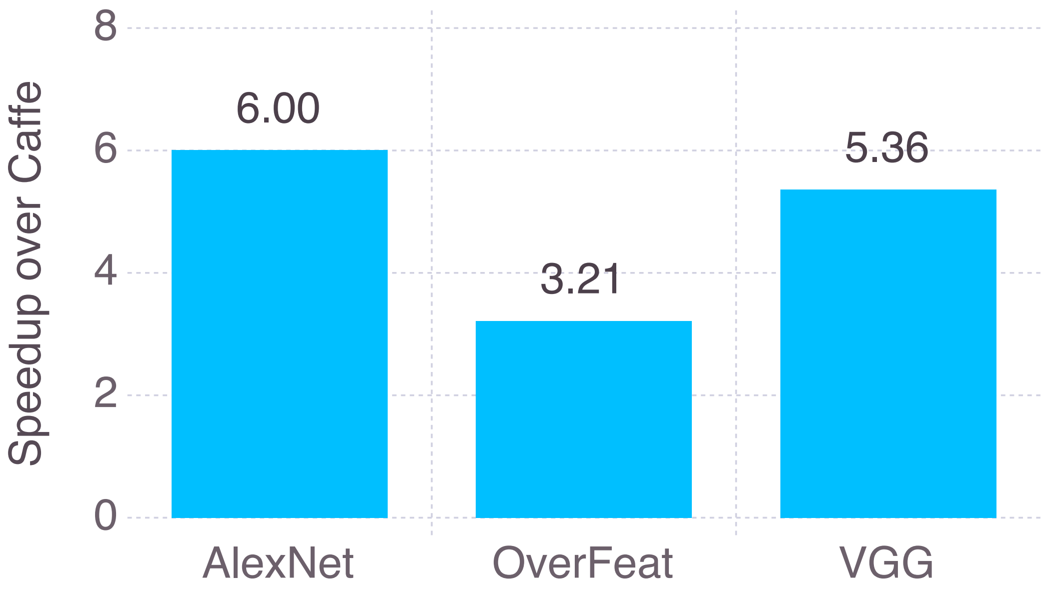
- Single node : Dual-socket Intel Xeon E5-2699 v3 (36 cores)
- Supercomputer : NERSC Cori Phase 1 (info at nersc.gov)

More Platforms (see paper):

- Accelerator Offloading (Xeon + Xeon Phi), Commodity Cluster

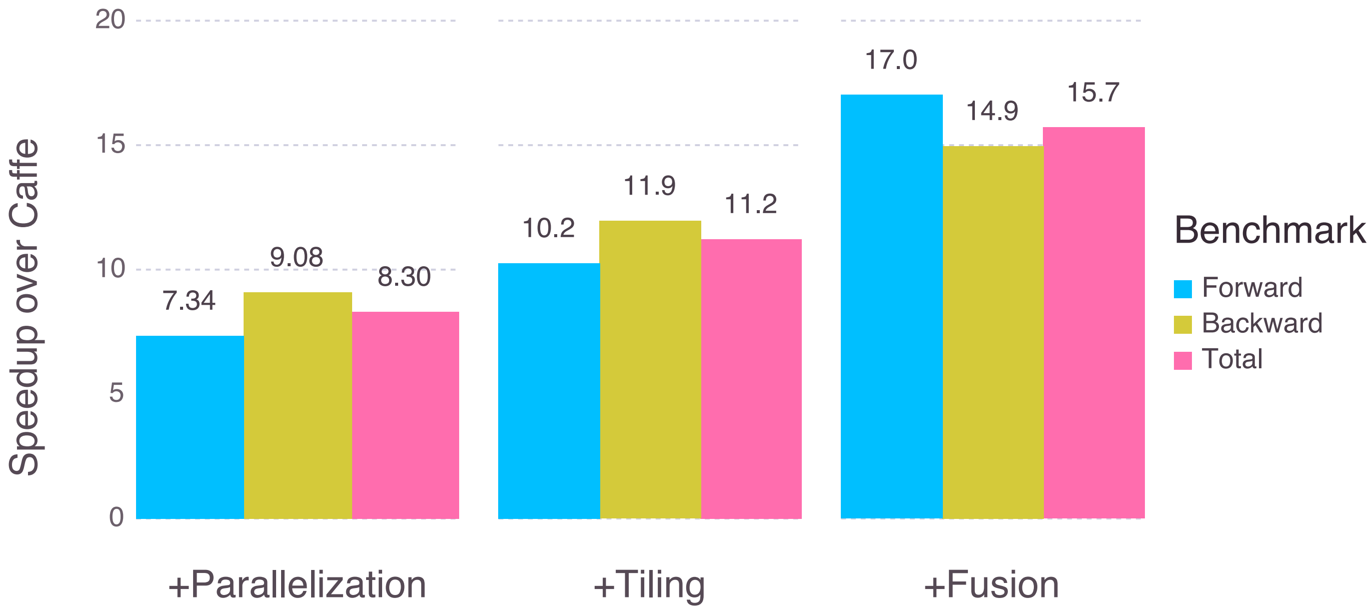
Latte Single Node Evaluation

3-6x speedup over Caffe for Training (Forward + Backward)



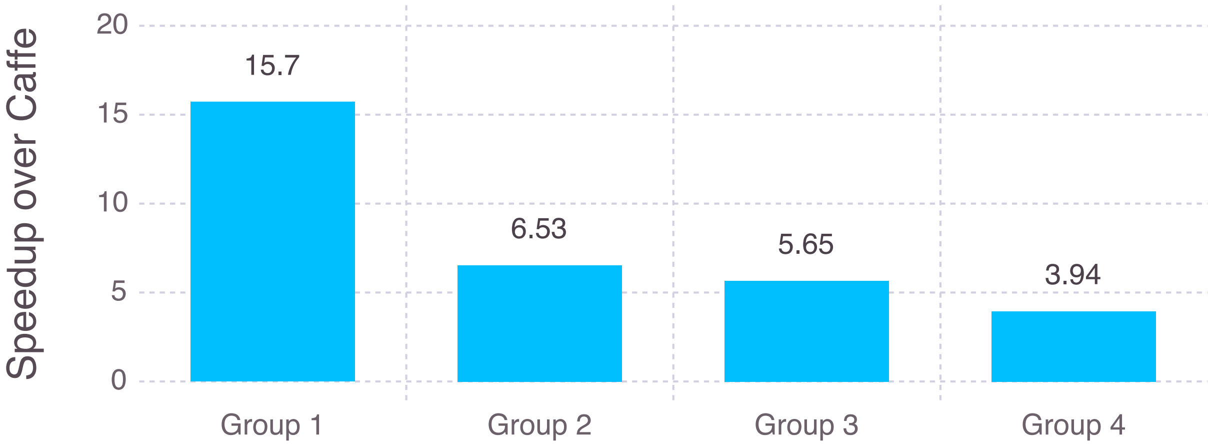
Latte Single Node Evaluation

Breakdown of optimization effectiveness for first 3 layers of VGG



Latte Single Node Evaluation

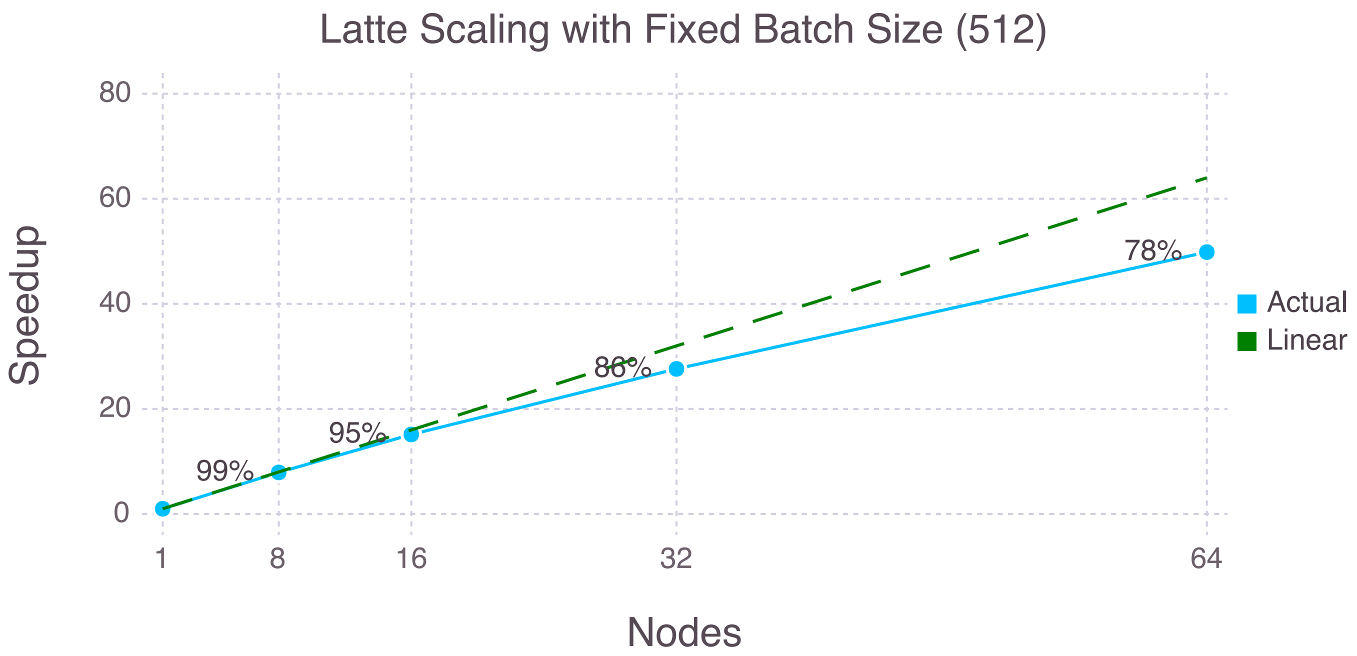
Breakdown of speedup for VGG (Forward)



First 16 layers of VGG

VGG	
Input (128, 3, 224, 224)	
Conv filts=64, kernel=3, pad=1	Group 1
ReLU	Group 1
Pool kernel=2, stride=2	Group 1
Conv filts=128, kernel=3, pad=1	Group 2
ReLU	Group 2
Pool kernel=2, stride=2	Group 2
Conv filts=256, kernel=3, pad=1	Group 3
ReLU	Group 3
Conv filts=256, kernel=3, pad=1	Group 3
ReLU	Group 3
Pool kernel=2, stride=2	Group 3
Conv filts=512, kernel=3, pad=1	Group 4
ReLU	Group 4
Conv filts=512, kernel=3, pad=1	Group 4
ReLU	Group 4
Pool kernel=2, stride=2	Group 4

Latte on Cori



Each worker is given a 512 / N sized sub-batch (N = number of Nodes)

Conclusion

Developed a novel abstraction for describing neural networks

Demonstrated performance competitive with existing frameworks

Code: Latte.jl is available at https://github.com/IntelLabs/Latte.jl

Ongoing Work: Forthcoming release of Python implementation with new features and better performance

Acknowledgements

Thanks to **Brian Lewis, Lindsey Kuper,** and **Justin Gottschlich** for their valuable feedback on the paper and presentations.

Thanks to **Paul Hargrove** at NERSC for access to Cori

For the UC Berkeley contributors, research was partially funded by DARPA Award Number HR0011- 12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and AS- PIRE Lab industrial sponsors and affiliates Intel, Google, Hewlett-Packard, Huawei, LGE, NVIDIA, Oracle, and Samsung.