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

Leonard Truong
Intel Labs / UC Berkeley, USA
leonard.truong@intel.com

Rajkishore Barik
Intel Labs, USA
rajkishore.barik@intel.com

Ehsan Totoni
Intel Labs, USA
ehsan.totoni@intel.com

Hai Liu
Intel Labs, USA
hai.liu@intel.com

Chick Markley
UC Berkeley, USA
chick@berkeley.edu

Armando Fox
UC Berkeley, USA
fox@cs.berkeley.edu

Tatiana Shpeisman
Intel Labs, USA
tatiana.shpeisman@intel.com

Abstract

Deep neural networks (DNNs) have undergone a surge in popularity with consistent advances in the state of the art for tasks including image recognition, natural language processing, and speech recognition. The computationally expensive nature of these networks has led to the proliferation of implementations that sacrifice abstraction for high performance. In this paper, we present Latte, a domain-specific language for DNNs that provides a natural abstraction for specifying new layers without sacrificing performance. Users of Latte express DNNs as *ensembles* of *neurons* with *connections* between them. The Latte compiler synthesizes a program based on the user specification, applies a suite of domain-specific and general optimizations, and emits efficient machine code for heterogeneous architectures. Latte also includes a communication runtime for distributed memory data-parallelism. Using networks described using Latte, we demonstrate 3-6× speedup over Caffe (C++/MKL) on the three state-of-the-art ImageNet models executing on an Intel Xeon E5-2699 v3 x86 CPU.

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA

http://dx.doi.org/10.1145/2908080.2908105

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization

Keywords Deep Learning, Neural Networks, Domain Specific Language, Compiler, Optimization

1. Introduction

Applications such as image processing, video processing, and speech recognition have become ubiquitous in modern computing. This has driven the development of algorithms that can automatically analyze their content to perform tasks such as online web search and contextual advertising. Recently, neural networks have demonstrated state-of-the-art results for the tasks of understanding images, videos, and speech. Apple Siri, Google Now, Microsoft Cortana, Microsoft Bing, and Amazon Echo all use deep neural networks (DNNs) under-the-hood.

Deep neural networks are a class of models that use a system of interconnected neurons to estimate or approximate functions. In practice these networks are built with groups of neurons called *layers* with an input layer and an output layer corresponding to inputs and outputs of the function being estimated. Between the input and output layers are *hidden layers* that perform intermediate computation. A neural network *architecture* describes a neural network configuration with a specific set of layers.

Advances in deep learning research are largely driven by the introduction of novel hidden layers and architectures. Examples of recent advancements in the state of the art that stem from novel layers and architectures are the Inception Architecture [44], the PReLU layer [26], and Batch Normal-

ization [31]. Developing these new layers and architectures is a time consuming process that requires large amounts of computational power and training data.

The demanding computational characteristics of deep neural networks has lead to the emergence of two dominant approaches for efficient implementations: high-performance layer-specific libraries and computational graph engines. Below we summarize them.

- *Layer-Specific Libraries:* A layer-specific library exposes a set of functions to perform the computation of various layer types. Users can compose these functions to construct various network architectures. By restricting the interface to the computation of specific layers, library writers can provide highly efficient, specialized implementations, particularly when mapping to a specific set of hardware targets. This is the most widely adopted approach for deep learning frameworks and libraries used by Caffe [32] and many others [14, 29, 35, 47].
- *Computational Graph Engines:* Libraries like Theano [10], CNTK [5], Torch [17], and TensorFlow [4] can be used to describe neural networks as computation graphs. These graphs are constructed using nodes that represent generic operations on n-dimensional arrays typically called tensors. The underlying graph engine is responsible for compiling, optimizing, and executing these graphs. This abstraction improves programmability by providing a high-level interface for expressing layers as generic operations.

These approaches have made at least two key sacrifices in order to provide the necessary performance to train complex networks. First, both approaches require the user to represent neural networks using an *array programming model* where the network state is encoded using tensors with layers implemented as functions on these tensors. This allows implementations to leverage the wide body of work on optimizing array programs including techniques such as parallelization and vectorization [33]. However, representing a neural network using an array programming model diverges from the graphical model typically used to describe neural network constructs. It places a burden on neural network researchers to derive array programming formulations of novel layers. Furthermore, a simple array formulation is typically not enough to achieve the performance necessary to train these networks. For example, CNTK [5] uses a matrix-multiplication based formulation for convolution layers. The reasoning behind this approach relies on significant understanding of the target hardware architecture and is described in detail by Chetlur et al [14]. This demonstrates that even a graph engine that strives to improve programmability has failed to prevent architecture specific details from leaking through the abstraction layer.

The other key missing performance-critical feature in existing frameworks is the *cross layer optimizations*. Prior work in the domains of linear algebra [9] and machine learning [7] have revealed opportunities for performance improvements from kernel fusion. Static layer-specific libraries are fundamentally unable to perform kernel fusion for layers because the layer kernels are statically compiled ahead of time. A graph engine could do this at runtime when processing the computation graph for a network but are faced with two challenges. First, in practice these graph engines rely on static kernels to execute computationally expensive nodes, such as a vendor BLAS implementation for matrix multiplication or a static library like cuDNN. When using static kernels, graph engines are subject to the same limitation as layer-specific libraries. Second, when using dynamically compiled kernels, graph engines would require complex dependence analysis to perform fusion.

In this paper, we introduce *Latte*, a domain-specific language (DSL) for deep neural networks. In Latte, a user can express neural networks as a collection of connected neurons — the same way neural networks are introduced and explained in courses and textbooks. The language implementation enforces abstraction by hiding low-level details including parallelization, heterogeneous code generation, and optimizations from the end-user. We provide examples of expressing existing layers in Latte to show how this abstraction facilitates the development of new layers by allowing researchers to use a graphical representation.

Latte extends the computational graph engine approach by replacing generic operations on multi-dimensional arrays with an abstraction specific to neural networks. This allows users to write neural network layers at a high-level without architecture specific optimizations. For example, a convolution layer can be succinctly specified as an Ensemble of neurons with a sparse connection structure. This contrasts with existing approaches including Torch [3], Caffe [32], CNTK [5] and cuDNN [14] that all use a matrix-multiply based implementation of convolution layers.

Under the hood, Latte uses program synthesis and domain-specific optimizations to execute networks efficiently. Guided by semantic information provided by the DSL, the Latte compiler performs cross-layer fusion without complex dependence analysis. The ability to perform cross-layer fusion contrasts with existing graph engine approaches that are limited by nodes that rely on static kernel implementations to do heavy computation. For example, CNTK [37] relies on static kernels to implement many computation nodes, fundamentally limiting their ability to perform cross-layer optimizations.

Our specific contributions are as follows:

- A domain-specific language for expressing neural networks as a graph of connected neurons. The language was designed using a top-down approach to match domain expert concepts rather than platform specific details.

- A DSL compiler that synthesizes a program to execute user-described networks, applies a suite of domain-specific optimizations, and emits efficient parallel and vector machine code for heterogeneous architectures. The DSL was designed to enable the compiler to exploit cross-layer optimizations resulting in a 16× performance improvement on the first three layers of the VGG network [42] over Caffe [32] on an Intel Xeon E5-2699 v3 x86 CPU with 36 cores.
- A runtime that enables data-parallel training of networks by dynamically scheduling execution across available compute units within a node and across nodes in a cluster and managing synchronization of gradients.
- Performance improvements of 3-6× over Caffe (MKL) and 15-40× over Mocha.jl [47] on the state-of-the-art network topologies AlexNet [36], VGG [42], and OverFeat [41] on an Intel Xeon E5-2699 v3 x86 CPU with 36 cores. Furthermore, Latte can achieve an additional 2× speedup from the same code base when the CPU is enhanced with two Intel Xeon Phi 7110P coprocessors. Finally, we show 84% strong scaling efficiency on a 32 node commodity cluster.

The rest of the paper is organized as follows. In Section 2, we describe some background on DNNs including terminology that will be used throughout the paper. Section 3 describes the overall language design. Example programs using Latte are presented in Section 4. Compiler optimizations and synthesis are described in Section 5. Section 6 describes the cluster and heterogeneous scheduling runtime. Experimental evaluations are presented in Section 7. We discuss related work in Section 8 and conclude in Section 9.

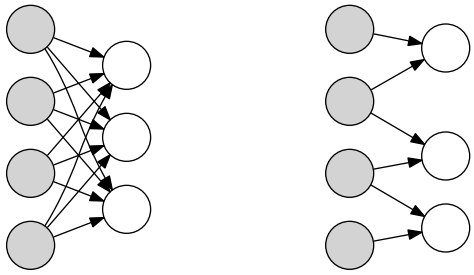
2. Background

To assist readers who may not be familiar with neural networks and deep learning, we provide a summary of key concepts and terminologies that will be used throughout the paper.

2.1 Neural Networks

In machine learning, the term *neural network* refers to a family of machine learning algorithms that are inspired by the biological neural networks found in the brain. These networks can be used to estimate the output(s) of functions that can depend on a large number of inputs. Typically, neural networks are presented as a collection of connected neurons. A connection between neurons indicates that messages are passed between the two neurons. We can define neurons as nodes and connections as directed edges to view neural networks as directed graphs. Networks with cycles in their graph are called *recurrent* neural networks (RNNs), while acyclic networks are called *feedforward*.

Neurons in a neural network are organized into *layers*, as in Figure 1a. The first and last layers are called the input and output layers, respectively. Intermediate layers are called *hidden* layers, and *deep* neural networks are those with many such layers. For example, the AlexNet network [36] has five *convolutional* and three *fully-connected* (described below) layers, with thousands or hundreds of thousands of neurons per layer.



(a) Fully connected layer **(b)** Sparsely connected layer

Figure 1: Simple examples of various layer types. Gray neurons represent input neurons and white neurons represent output neurons.

2.2 Execution

The execution of a neural network begins with initializing the input neurons with the current input data. This seeds the execution of the network by presenting a set of neurons that have available outputs which are propagated to connected neurons. A neuron can compute an output once all of its inputs are available.

2.3 Network Primitives

Neurons are the fundamental unit of neural networks. A neuron, sometimes called simply a *unit*, takes in any number of inputs and computes an output value called an *activation*. A neuron is defined by how the output activation is computed. The most common neuron type found in deep neural networks is the *weighted* neuron. A weighted neuron multiplies each of its inputs by a unique weight, and computes an output activation as a sum of the weighted inputs. Another example of a neuron is a *max* neuron that computes an output activation as the maximum of its inputs.

Fully connected (FC) layers are found in many classes of neural networks. In an FC layer, each input neuron is connected to each output neuron, as shown in Figure 1a. The output neurons are defined as the weighted neurons.

Convolution layers are similar to FC layers except neurons are connected to a spatially-local neighborhood of input neurons. This results in a sparse connection structure shown in Figure 1b. Also different than in fully connected layers, certain neurons in a convolution layer share weights.

Pooling layers use a sparse, spatially-local connection structure, however unlike convolution layers, the neighborhoods typically do not overlap. Pooling layers are commonly

inserted to reduce the spatial size of the network, the number of parameters, and the computational cost.

2.4 Network Architectures

A network architecture describes a specific neural network using a configuration of connected layers. Many modern neural network models use sub-architectures as reusable and composable blocks.

Multi-layer Perceptron (MLP): An example of a simple network architecture that is capable of approximating universal functions [28]. MLPs can be described as a sequence of fully connected layers that perform non-linear transformations on the input data.

Long Short-term Memory (LSTM): An LSTM [22, 27] is a more complex, *recurrent* architecture that has achieved state of the art results on machine learning tasks including handwriting and speech recognition [25]. LSTMs uses a structure called a *memory cell* to control vanishing and exploding gradients when training using back propagation. A *memory cell* can be described as an input gate, a neuron with a self-recurrent connection, a forget gate, and an output gate. The gates modulate the input, output, and internal signals which allow the *memory cell* to perform complex interactions with its environment such as forget its previous state.

2.5 Training

To train neural networks, researchers employ forward and backward propagation. First, under forward propagation, a training input is fed through the network to produce output activations. Next, under backward propagation (or simply *back-propagation*), the output activations are propagated backwards through the network to calculate the difference between the input and output values of each neuron. These differences are called gradients, which are used to update the weights of a neuron. In practice, networks are trained on batches of input items instead of a single input to improve vectorization and parallelization.

A *solver* is responsible for coordinating the forward, backward, and weight update phases of training. It is responsible for solving the minimization problem used to learn the best parameters for a given dataset. Example solving methods include stochastic gradient descent [6], RMSProp [45], and AdaDelta [20].

Large networks can take days or weeks to converge, motivating distributed training algorithms that expose more opportunities for parallelism. Prior work [19, 34] has shown *model* and *data* parallelism as two effective strategies for parallelizing training in a distributed memory environment. Data-parallel training replicates the network across multiple workers, with each worker processing a different batch of inputs. Various schemes have been devised for synchronizing these network instances, including gradient summation [46], parameter averaging [43], and parameter servers [15]. Model parallelism partitions a single network across multiple workers (a requirement when a model is too large to fit in memory on a single node).

3. Latte Language Design

We have developed Latte, a domain-specific language for DNN programming. Latte is implemented as an extension to the Julia programming language [11]. A network in Latte consists of a collection of connected ensembles. An ensemble is a collection of neurons. Ensembles are connected using a mapping function that specifies the connections between the neurons in the ensembles. The names *neuron*, *ensemble*, and *connection* were inspired by the Nengo framework [8]. These language constructs enable the implementation of standard CNN layers such as InnerProduct, Convolution, Pooling, ReLU, Softmax, Batch Normalization, and Local Response Normalization, as well as RNN blocks such as the Gated Recurrrent and Long Short Term Memory units. We present the definition of Latte's core types and functions in Figure 2.

3.1 Neuron

The `Neuron` type is a base type provided by Latte that remains abstract and must be sub-typed by the user. All `Neuron` sub-types will have four default fields: the neuron's output `value`, its gradient to be back-propagated ∇, a vector of vectors of input activations `inputs`, and its gradient ∇`input`. In addition to these default fields, the user can specify additional fields for the neuron to represent internal state. If a user wishes to store information specific to each item in an input batch, they can mark the field as a *Batch* type.

In addition to defining `Neuron` sub-types, the user must also specify `forward` and `backward` functions with the signatures given in Figure 2. These define how forward and backward propagation are computed. They each take a user-defined sub-type of `Neuron` as their argument. For `forward`, the user must write the resulting output value of the neuron to its `value` field. For `backward`, the user must write the resulting gradients to be propagated to ∇`inputs`. Latte offers a macro `@neuron` to help user conveniently define custom neuron types and their forward and backward functions, and an example will be given in Section 4.

Latte provides special handling of gradient computation based on prior work showing neural networks' resilience to lossy gradients. Project Adam [15] showed that in certain cases, lossy weight updates actually improved accuracy by introducing noise into the training process. When enabled, fields specified with ∇ in the name will not be subject to synchronized reductions across threads, instead allowing threads to update their computed values in place. If the user does not enable this mode, Latte will perform a normal synchronized reduction incurring a small performance overhead during back-propagation.

```
type Neuron                  type Connection          type Ensemble{T <: Neuron, N}      type Net
  value    :: Float32          source  :: Ensemble      neurons     :: Array{T, N}          ensembles::Vector{Ensemble}
  ∇        :: Float32          mapping :: Function      connections :: Vector{Connection}   ...
  inputs   :: Vector{Vector}   ...                      ...                                end
  ∇inputs  :: Vector{Vector}  end                     end
end

forward{T <: Neuron}(neuron :: T)     # must be user-defined for each custom neuron type T
backward{T <: Neuron}(neuron :: T)    # must be user-defined for each custom neuron type T
add_connections(net :: Net, source :: Ensemble, sink :: Ensemble, mapping :: Function)
solve(solver :: Solver, net :: Net)
```

Figure 2: Core Latte types and functions using Julia syntax [11], where `type...end` defines a struct (or record) type, `::` denotes the type of a field or a variable, `{...}` encloses type parameters, and `<:` denotes a sub-typing relationship. We also follow Julia's convention of capitalizing the first letter of type names. `Vector{Vector}` indicates a Vector of Vectors.

3.2 Ensembles

Modern neural network topologies leverage the notion of collections of neurons which are composed and connected. Latte provides a built-in type for neuron collections called `Ensemble`. The `Ensemble` type is parameterized by the rank `N` of its neuron array and neuron type `T`, which implies that all neurons in an ensemble must be of the same type and hence use the same function to compute an activation. This uniform activation computation is leveraged by the compiler for optimizations and explained in detail in Section 5. Alongside the fundamental `Ensemble` type, Latte provides two special ensemble types:

ActivationEnsemble: Applying an activation function over each neuron in a collection is a common pattern in neural networks. Latte provides the `ActivationEnsemble` construct as an easy way to apply an activation function over any existing ensemble. Latte will construct a new ensemble with the same dimensions as the input ensemble and connect them using a one-to-one mapping. By using an `ActivationEnsemble`, the user allows Latte to perform the forward and backward propagation computations in-place. This contrasts with the default behavior which would allocate a separate memory region for an ensemble's value.

NormalizationEnsemble: Specifying normalization operations is often better suited for array- or vector-style operations on a set of values in the network. Latte provides the `NormalizationEnsemble` type to allow the user to specify generic operations on an array encoding the output values of an ensemble. The output of a `NormalizationEnsemble` will be identical in shape to the input.

3.3 Connections

Latte provides the built-in function `add_connections` for connecting ensembles and neurons. Connecting an ensemble `source` to an ensemble `sink` specifies that the values of the neurons in `source` will be used by the neurons in `sink`. The `mapping` function specifies the input values for every neuron in `sink` by returning a range of neuron indices in `source`.

```
1  @neuron type WeightedNeuron
2    weights   :: Vector{Float32}
3    ∇weights :: Vector{Float32}
4    bias      :: Vector{Float32}
5    ∇bias     :: Vector{Float32}
6  end
7
8  @neuron forward(neuron::WeightedNeuron) do
9    # perform dot product of weights and inputs
10   for i in 1:length(neuron.inputs[1])
11     neuron.value +=
12         neuron.weights[i] * neuron.inputs[1][i]
13   end
14   # add the bias
15   neuron.value += neuron.bias[1]
16 end
17
18 @neuron backward(neuron::WeightedNeuron) do
19   # Compute back propagated gradient
20   for i in 1:length(neuron.inputs[1])
21     neuron.∇inputs[1][i] +=
22         neuron.weights[i] * neuron.∇
23   end
24   # Compute weight gradient
25   for i in 1:length(neuron.inputs[1])
26     neuron.∇weights[i] +=
27         neuron.inputs[1][i] * neuron.∇
28   end
29   # Compute bias gradient
30   neuron.∇bias[1] += neuron.∇
31 end
```

Figure 3: Latte standard library definition of a WeightedNeuron

3.4 Network

In Latte, layers are represented as an ensemble of neurons with a specific connection structure for an input ensemble. These layers can be composed to create neural networks, defined in Latte with the `Net` type. Examples of constructing common neural network layers in Latte can be found in Section 4 where we describe the implementation of the Latte standard library. To construct a network, the user adds ensembles to an `Net` instance and applies connections between them. Latte provides a routine `init` that initializes a network by compiling the network to an executable and allocating required memory buffers. An initialized network can be passed to solver routines using the `solve` interface for training or used to perform prediction tasks with an existing set of parameters. Alongside standard layers, Latte provides a number of built-in solvers as described in Section 2.5. Solvers

```
1  function FullyConnectedLayer(name::Symbol, net::Net,
       input_ens::AbstractEnsemble, n_outputs::Int)
2    # Create a vector of `n_outputs` WeightedNeurons
3    neurons = Array(WeightedNeuron, n_outputs)
4    # Initialize parameters
5    n_inputs = length(input_ens)
6    weights,∇weights = xavier_init(n_inputs,n_outputs)
7    bias,   ∇bias    = zeros(1,n_outputs)
8    # Instantiate each neuron with unique parameters
9    for i in 1:n_outputs
10     neurons[i] = WeightedNeuron(weights[:, i],
11       ∇weights[:, i], bias[:, i], ∇bias[:, i])
12   end
13   # Construct the ensemble
14   fc = Ensemble(net,name,neurons,
15         [Param(:weights,1.0), Param(:bias,2.0)])
16   # Connect all source neurons to each sink neuron
17   mapping = (i) -> [1:d for d in size(input_ens)]
18   add_connections(net, input_ens, ip, mapping)
19   fc
20 end
```

Figure 4: Latte standard library definition of a FullyConnectedLayer. `xavier_init` refers to the randomized parameter initialization scheme using the Xavier algorithm [23].

define an update method that is responsible for updating the parameters with respect to the gradient.

4. Examples

In this section, we present examples using the Latte language to define `Neuron` sub-types, construct layers using `Neurons` and `Ensembles`, and construct networks using layers. These examples are provided in the Latte standard library.

WeightedNeuron: The `WeightedNeuron` is a fundamental building block of artificial neural networks. A `WeightedNeuron` computes its output as a dot-product of its inputs with a weight vector. This weight vector is a learnable parameter and can be shared amongst neurons. Weighted neurons can be used to construct fully-connected and convolution layers. Figure 3 shows the code used to define the `WeightedNeuron` type in Latte. We begin on line 1 by defining a `Neuron` subtype using the `@neuron` macro and the standard Julia type definition syntax. For `WeightedNeuron`, additional fields are specified to hold the weight and bias parameters for the neuron. Next, on lines 8 and 18 we define `forward` and `backward` functions using the `@neuron` macro and the Julia do-block syntax.

Fully connected layers: Fully connected layers are constructed by connecting an ensemble of `WeightedNeurons` to an input ensemble with a simple mapping function. Figure 4 shows the standard library implementation. First, on line 3 we allocate and instantiate an array of `WeightedNeurons` of length `n_outputs`, each with its own unique set of weights. These neurons are used to construct an ensemble `fc` on line 14. Finally, on line 18 we connect each neuron in `input_ensemble` to each neuron in `fc`, and return `fc`.

Convolution layers: Convolution layers can be described in Latte using the same constructs as fully connected layers.

```
1  add_connections(net,input_ens,conv,function(x,y,_)
2    in_x = (x-1)*stride-pad
3    in_y = (y-1)*stride-pad
4    return (in_x+1:in_x+kernel,  # kernel width
5            in_y+1:in_y+kernel,  # kernel height
6            1:channels)          # all input channels
7  end)
```

Figure 5: Connection structure for a convolution layer encoded as a Latte mapping function. It returns a range of neurons in the input ensemble based on the index of a neuron in `conv`.

```
1  function LSTMLayer(name::Symbol,net::Net,
       input_ensemble::AbstractEnsemble,n_outputs::Int)
2    # Split the input_ensemble into 4 gates
3    for ens in [:ix, C_simx, :fx, :ox]
4      @eval $ens = InnerProductLayer(
5          $net, $input_ensemble, $n_outputs)
6    end
7    # Split the previous output to 4 gates
8    for ens in [:ih, :C_simh, :fh, :oh]
9      @eval $ens = FullyConnectedEnsemble($net, length
           ($input_ensemble), $n_outputs)
10   end
11
12   i = σ(net, +(net, ih, ix))
13   f = σ(net, +(net, fh, fx))
14
15   C_sim = tanh(net, +(net, C_simh, C_simx))
16   f_C = MulEnsemble(net, (n_outputs, ))
17   add_connections(net, f, f_C, (i) -> (i,))
18   C = +(net, *(net, i, C_sim), f_C)
19   add_connections(net, C, f_C, (i) -> (i,);
20                   recurrent=true)
21
22   oC = InnerProductLayer(net, C, n_outputs)
23   o = σ(net, +(net, oC, oh, ox))
24   h = *(net, o, tanh(net, C; copy=true))
25
26   # Connect h back to each gate
27   for ens in [ih, C_simh, fh, oh]
28     add_connections(net, h, ens, (i)->(1:n_outputs,)
         ; recurrent=true)
29   end
30 end
```

Figure 6: Definition of an LSTM unit in Latte. The math functions $\sigma, +, *, tanh$ construct an ensemble of neurons to perform the corresponding operation and connects the inputs. The use of Julia's `@eval` allows the inputs to be split into 4 distinct layers using metaprogramming.

The key differences are in the sharing of weights between certain neurons (based on their index in the ensemble) and the use of the sparse connection structure as shown in Figure 5. This natural, high-level specification contrasts greatly with the matrix-multiplication based convolution specifications found in many deep learning frameworks [3, 5, 14, 32].

Long Short-Term Memory Units (LSTM): The Latte implementation of an LSTM unit is shown in Figure 6. It demonstrates how recurrent connections can be naturally expressed in Latte. Alongside the `WeightedNeuron`, the LSTM unit requires neurons to perform mathematical functions such as σ, $+$, $*$, and $tanh$. We leave out their definitions for space but they can be found online at the open source

```
using Latte

net = Net(8)
data, label = HDF5DataLayer(net, "data/train.txt",
                                 "data/test.txt")
ip1  = FullyConnectedLayer(:ip1,  net, data, 20)
ip2  = FullyConnectedLayer(:ip2,  net, ip1,  10)
loss = SoftmaxLossLayer(  :loss, net, ip2,  label)

params = SolverParameters(
    lr_policy  = LRPolicy.Inv(0.01, 0.0001, 0.75),
    mom_policy = MomPolicy.Fixed(0.9),
    max_epoch  = 50,
    regu_coef  = .0005)
sgd = SGD(params)
solve(sgd, net)
```

Figure 7: A simple multi-layer perceptron in Latte using a solver and layers found in the standard library. `LRPolicy` and `MomPolicy` are parameters to the solver.

link provided in Section 9. LSTMs begin by splitting the input into 4 distinct signals. This is done using the `for` loop on line 4 which instantiates 4 FullyConnectedLayers. On line 9, we do a similar process of splitting the output of the memory cell at the previous time step. These signals are used to compute the values of the LSTM gates and memory cell. `i` encodes the input gate, `f` encodes the forget gate, `C_sim` is the candidate state value, `o` is the output gate, and `h` is the memory cell output.

Multi-layer perceptrons: Figure 7 provides an example of constructing and training an MLP as described in Section 2.4 using two `FullyConnectedLayers`. These layers, as well as an input data layer and a loss layer, are added to a network. The user then instantiates a solver that is used to train the network.

5. Latte Compiler

The Latte compiler consumes the user description of a neural network and produces a compiled binary to perform the execution of the network. The compiler has four phases: analysis, synthesis, optimization, and code generation. During these phases, Latte uses an intermediate representation that is a superset of the internal Julia AST. Machine code generation is handled by the *ParallelAccelerator.jl* package [30]. In this section, we describe the key components of the Latte compiler.

5.1 Internal Representation of Networks

User-described networks in Latte trivially map to a data-flow graph with nodes representing computations and connections indicating data dependencies. Latte internally represents the data-flow graph using implicit adjacency lists. The adjacency lists for neurons in an ensemble can be retrieved using the mapping function(s) used to connect the ensemble to its input(s). The implicit adjacency list representation allows Latte to store complex graphs without heavy memory usage even for networks with billions of neurons and connections.

```
for n = 1:num_neurons
  for i = 1:num_inputs
    fc_value[n] += fc_inputs[i,n] * fc_weights[i,n]
```

⇓

```
for n = 1:num_neurons
  for i = 1:num_inputs
    fc_value[n] += fc_inputs[i] * fc_weights[i,n]
```

Figure 8: FullyConnectedLayer pseudo-code before and after shared variable analysis. The upper code-block shows the naive code where each neuron has a different set of inputs demonstrated by the use of `n` to index `fc_inputs`. In the lower code-block the index `n` has been dropped because the compiler has determined that the inputs are shared.

5.2 Analysis of Shared Variables

The Latte compiler uses shared variable analysis on the data-flow graph to guide code synthesis and optimization. During this phase, the compiler determines compute nodes in the data-flow graph that share data dependencies. Shared variables occur when neurons consume the same input values and neurons sharing local fields such as parameters. The Latte compiler leverages this information to improve data locality and reduce communication overhead in the synthesized code by mapping shared values to the same memory region. Neurons work together to load shared data instead of independently reading separate memory regions which would result in cache capacity conflicts as well as saturation of memory bandwidth. This also results in the runtime allocating a single shared buffer for inputs, reducing overall memory consumption.

Analysis of shared values is performed by partitioning the data-flow graph into ensembles and performing a traversal that compares dependencies amongst compute nodes within a partition. Inside an ensemble partition, the compiler compares the adjacency lists of neurons along a dimension. If this list is uniform across the dimension, the compiler has determined that the neurons along that dimension can share the same buffer.

An example of an optimization guided by shared variables is found during the synthesis of FC layers. In a FC layer, each neuron in the output ensemble consumes the activation of every neuron in the input ensemble. Therefore, every neuron in the output ensemble can consume the same shared input vector of activations, as demonstrated in Figure 8. This optimization improves locality, and enables the computation to be pattern-matched as a matrix multiplication. We will discuss pattern matching in further detail in Section 5.4.

Convolution layers exhibit a more complex connection structure than FC layers but still exhibit the shared variable pattern. In a convolution layer, neurons in each output channel share the same weight values for the filter but are connected to different inputs. Across the output channel dimension, neurons share the same inputs, but use different filter

weights. Latte is able to determine the uniformity of values across an entire dimension. In this case, Latte drops this dimension, allowing the neurons to share the same buffer. As with FC layers, this shared value optimization enables the kernel pattern matcher to formulate convolution layers as a matrix multiplication as well.

5.3 Synthesis

Dataflow: The compiler is responsible for ensuring the input data is available for the computation of a neuron's output. The required data dependencies specified by the user are encoded in the connection structure of neurons in the data-flow graph. Latte performs synthesis by traversing the data-flow graph. Each ensemble has metadata containing information collected from shared variable analysis and the ensemble type. In special cases, such as with `ActivationEnsembles` or when all neurons share the same input values, Latte does not perform data-flow synthesis, instead relies on the runtime mapping of the input pointers to the memory regions containing the appropriate values. In the general case, Latte assumes that the runtime has allocated a buffer for the input values of each neuron in an ensemble. Then, for each connected input (which we will call a source) to a neuron (which we will call a sink), Latte emits a data copy task to move that source's output value into the allocated input buffer for the sink. The synthesis of these data copy tasks is done by generating a loop nest that copies the inputs for each neuron in the current ensemble. This is also guided by shared variable analysis, which allows Latte to drop dimensions in the synthesized loop nest when inputs are shared along a certain dimension. For example, along the output channel dimension of a convolution layer, each filter group shares the same input values. The runtime will map the inputs of these neurons to the same shared buffer, so Latte will only copy data into the shared buffer once instead of for each group along that dimension.

Compute: By definition of ensemble as a homogeneous collection of neurons, the function used to compute the output of each neuron in the ensemble is identical. In Latte, users define neuron functions using references to neuron fields, which naturally maps to an array-of-structs (AoS) representation. However, using an AoS representation would prevent Latte from effectively generating vectorizable code. For example, when performing data-flow copies, Latte iterates over the output values of a neuron in an ensemble, but ignores any other fields of that neuron. To enable efficient vectorization, the Latte compiler performs a transformation pass on the neuron function to convert references to reflect a struct-of-arrays (SoA) layout. During this transformation pass, the compiler also updates indexing expressions on fields that have been determined to be shared along a dimension. Next, the compiler synthesizes a set of nested for-loops by calling the neuron function for each neuron in an ensemble.

```
1  # Convolution Layer
2  for c = 1:n_filters, y = 1:height, x = 1:width
3    for i in 1:n_inputs
4      conv1[x, y, c] += conv1input[i, x, y] *
           conv1weights[i, c]
5  # ReLU Layer performed in place
6  for c = 1:n_filters, y = 1:height, x = 1:width
7    conv1[x, y, c] = max(conv1[x, y, c], 0.0)
8  # Pooling layer
9  for c = 1:n_filters, y = 1:height/2, x = 1:width
10   # Pooling Layer data copy
11   for p = 1:2, q = 1:2
12     poolinput[p*2+q, x, y, c] = conv1[x+q, y+p, c]
13   maxval = -Inf
14   for i = 1:pool_size
15     maxval = max(poolinput[i, x, y, c], maxval)
16   pool1[x, y, c] = maxval
```

Figure 9: Pseudo-code representing the synthesized code for a Convolution, ReLU, and Pooling layer.

Distributed Memory Communication: For synchronizing model replicas when using distributed data parallelism, Latte uses gradient summation for two reasons: First, gradient summation preserves the semantics of optimization algorithms with an increased batch size. Second, it allows us to synthesize communication code that overlaps with the back-propagation computation. After synthesizing a section of the code for the back-propagation of one ensemble, Latte inserts a call to the runtime to perform an asynchronous reduction of the computed gradient between workers. That is, as soon as a gradient is computed, Latte initiates asynchronous communication with other workers, and then continues computing more gradients. This reduces the overall cost of inter-node communication by overlapping it with computation.

5.4 Optimizations

We provide the pseudo-code in Figure 9 as an example to guide the reader through the various compiler optimizations employed by Latte. It represents the synthesized computation of a convolution layer, a ReLU layer, and a pooling layer. For simplicity, we leave out the data copy for the convolution layer, as well as the outermost loop over batch items.

5.4.1 Intra-Layer Optimizations

Library Kernel Pattern Matching: Latte implements a pattern matching pass to transform synthesized code into library calls, when possible. This enables Latte to leverage highly efficient implementations from vendors for specific kernels. Currently Latte only implements replacement of matrix multiplication, a common pattern found in the synthesized code for many neural networks, with a call to MKL's `sgemm` [2]. In our example, Latte transforms the convolution layer computation on lines 2-4 of Figure 9 into the following GEMM call[1] by flattening the x and y loops:

[1] Note the `gemm` call uses a simplified interface `gemm(transA, transB, m, n, k, A, B, C)` without arguments `lda, ldb, ldc, alpha, beta`.

```
1  for y_tile in 1:TILE_SIZE:height
2    gemm('T', 'N', TILE_SIZE*width, n_filters,
           n_inputs, conv1input[n], conv1weights,
3          conv1[n])
4  for y_tile in 1:TILE_SIZE:height
5    for c = 1:n_filters,
6        y = y_tile:y_tile+TILE_SIZE,
7        x = 1:width
8      conv1[x, y, c] = max(conv1[x, y, c], 0.0)
9  for y_tile in 1:TILE_SIZE:height/2
10   for c = 1:n_filters,
11       y = y = y_tile:y_tile+TILE_SIZE,
12       x = 1:width
13     for p = 1:2, q = 1:2
14       poolinput[p*2+q, x, y, c] = conv1[x+q, y+p, c]
15     maxval = -Inf
16     for i = 1:pool_size
17       maxval = max(poolinput[i, x, y, c], maxval)
18     pool1[x, y, c] = maxval
```

Figure 10: Code after tiling has occurred. Notice the difference in loop lengths on lines 4 and 8 as well as the fusion preventing dependency on line 13.

```
1  gemm('T', 'N', height * width, n_filters, n_inputs,
       conv1input, conv1weights, conv1)
```

Latte currently pattern-matches for matrix-multiplication only, however this is not a fundamental limitation. Extending Latte to support other patterns like replacing scatter/gather memory accesses with hand tuned+vectorized implementations is straightforward.

Loop Tiling: The synthesized code for neural networks is loop-heavy, and thus Latte employs loop tiling to improve performance. Tiling loops allows threads to compute tiles of the output in parallel while sharing values in the cache such as parameters. It also reduces memory consumption in a multi-threading environment as the runtime only needs to allocate space for a single tile of the input per thread rather than for the entire input for every batch item. Threads can reuse the allocated tile for each tile processed regardless of the tile index or batch index. The parallelization strategy for tiles is discussed in Section 5.4.3. During the tiling pass, Latte inserts metadata about tiled loops to aid in fusion. This is implemented by introducing a new tiled loop AST node in the compiler that contains the input dependence distance vector along the tiled dimension. Figure 10 shows the code after pattern matching and tiling for the example program in Figure 9.

5.4.2 Cross-Layer Fusion

Cross-layer fusion has the ability to greatly improve performance in certain layer configurations. In general, fusion is difficult because of the presence of fusion-preventing dependencies. Latte is able to leverage semantic information introduced into the AST by the tiling process to perform fusion of tiled loops, allowing threads to reuse data computed for a tile of a layer to immediately compute the output of the next layer. This fusion only occurs when there is no dependence along dimensions greater than the tiled dimension.

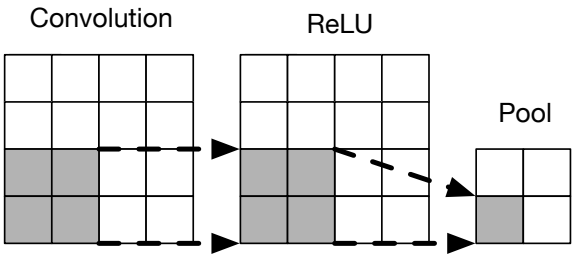


Figure 11: Observe that to compute a 2x2 tile of the Pool layer, a factor 2 larger tile along the vertical dimension of the previous layers is required.

```
1  #pragma omp for collapse(2) schedule(static, 1)
2  for n in 1:length(batch)
3    for y_tile in 1:TILE_SIZE:height/2
4      gemm('T', 'N', TILE_SIZE*2*width, n_filters,
           n_inputs, conv1input[n], conv1weights,
5          conv1[n])
6      for c = 1:n_filters,
7        for y = y_tile:y_tile+TILE_SIZE*2,
8            x = 1:width
9          conv1[x, y, c] = max(conv1[x, y, c], 0.0)
10       for y = y_tile:y_tile+TILE_SIZE,
11           x = 1:width
12         maxval = -Inf
13         for p = 1:2, q = 1:2
14           maxval = max(conv1[x+q, y+p, c], maxval)
15         pool1[x, y, c] = maxval
```

Figure 12: The final code after tiling, fusion, and parallelization. Note the scaled `TILE_SIZE` references in the Convolution and ReLU codes on lines 3 and 5.

In Figure 10 we see that the computation of the ReLU layer is trivially fusible due to an identical tile size and lack of loop-carried dependence along the tiling dimension. However on line 13 we can see a fusion-preventing dependency with an offset access to `conv1` for the pooling layer in the form of a loop-carried dependence. Furthermore, the number of tiles `pool1_y_tiles` is half the amount of `relu1_y_tiles` and `conv1_y_tiles` because pooling layers perform a sub-sampling of the inputs. Latte knows that the pooling layer has a dependence distance of 2 in the y dimension based on the connection structure, and thus attempts to double the tile size of the previous loops for fusion. This is depicted in Figure 11. Our final code after fusion is shown in Figure 12 with a doubled tile size for the convolution and ReLU loops. This results in the tiled loop length being reduced to half, presenting the compiler with identical loop trip counts for the tiled loops, which enables fusion. The compiler can ignore any other data dependencies within the current tile, because they are executed sequentially and by definition will only write to the neuron value array.

5.4.3 Parallelization

Except for *NormalizationEnsemble*s, the computation of an ensemble is implicitly data-parallel across batch items due

to the lack of edges crossing the outermost synthesized loop across items in a batch. Furthermore, inside an iteration of the batch loop, there typically exists a tiled loop. The computation of each loop tile is also data parallel, because the computation of a neuron's output does not depend on any other neurons in the ensemble. Latte parallelizes across the batch loop, and when it exists the tiled loop via collapsing (as shown in Figure 12). Typically values such as parameters or inputs are shared across tiles, thus Latte employs a static schedule that interleaves threads per iteration and assigns threads in a compact fashion to processor units to improve cache performance. This scheduling is performed using the OpenMP `KMP_AFFINITY=compact` and `schedule(static,1)` constructs for the platform used in our evaluation.

5.5 Code Generation

During the synthesis and optimization phases, Latte uses a super-set of the Julia internal AST as an intermediate representation. During parallelization, Latte introduces a node consumable by the ParallelAccelerator.jl package [30] indicating an explicitly parallel for-loop. This node contains information about collapsed loop nests, scheduling, and chunk size. During tiling, Latte introduces an AST node to represent tiled loops that carry metadata used during fusion as well as an AST node to prevent fusion across blocks when synthesizing code for unfuseable ensembles such as NormalizationEnsembles. After completion of optimization, Latte lowers tiled loop nodes into normal for-loops and removes the fusion-preventing AST nodes. When lowering the AST, Julia transforms for-loops into a series of goto statements and conditionals. Intel's C++ compiler is unable to effectively vectorize these loops due to the presence of complex conditionals and goto statements. To ensure that the final generated code is vectorized, Latte transforms the Julia AST before lowering to preserve the loop based structure of the code. Latte also annotates these for-loop nodes with pragmas to guide the C++ compiler to ignore vector dependencies and aliasing. ParallelAccelerator.jl consumes the lowered AST and emits C++ code which is then passed to Intel's C++ compiler. We extended ParallelAccelerator.jl to support code generation for the Xeon Phi coprocessor using Intel's Compiler Assisted Offload [1]. For this, we introduced an AST node to indicate offload regions as well as metadata used as clauses for offload pragmas indicating pointers to be copied to Xeon Phi.

6. Latte Runtime

The Latte runtime employs data parallelism for training neural networks by using a hierarchical design with two levels. The first level is within a server node where accelerators such as GPUs and Xeon Phi coprocessors communicate with a host CPU over a fast interconnection such as PCIe. We will call this *intra-node level data parallelism*. The second level of data parallelism is across the nodes in a cluster communicating over a network using MPI, which we will call *cluster-level data parallelism*. As discussed in Section 5.3, Latte inserts runtime calls to initiate gradient synchronization when the values have been produced. The runtime performs this synchronization using the MPI 3 asynchronous *Iallreduce*. The use of asynchronous MPI allows the reduction and communication of gradients to be overlapped with computation during back propagation and parameter updates.

6.1 Intra-node Data Parallelism

At present, many accelerators, the Intel Xeon Phi and certain discrete GPUs included, require a host processor as an interface to access the system memory. Typically, many deep learning frameworks have used the host CPU to fetch the next set of inputs which is transferred to the accelerator memory before processing. Latte will leverage available accelerators, however it employs two techniques to minimize the overhead of data transfer to and from the accelerator. First, we use input data *double buffering* to hide the latency of moving input data to the accelerator. That is, while the accelerator is processing a set of inputs, the Latte runtime will copy the next set of inputs into a separate buffer. When the iteration is finished, the input buffers are swapped and the runtime begins another copy into the old input buffer. After the first iteration, the time spent copying data to the accelerator is completely hidden by the actual computation. Typically the processing time of a batch of inputs is much larger than the data transfer time of the batch.

The next technique is to overlap host computation with data transfer and accelerator computation. Instead of having the host sit idle while the accelerator processes a batch, Latte divides the current batch into chunks, assigning chunks to each available compute units on the accelerator including the host. We begin by assigning accelerators a chunk size of 16 and the rest to the host. We use a linear search to increase chunk size until the time to process a chunk on the accelerator matches the time to process on the host. This search occurs once at the beginning of training.

7. Evaluation

In this section, we evaluate the performance of Latte on both shared memory and distributed memory systems and compare its performance with existing systems. In particular, we compare Latte with the popular and high-performance Caffe [32] deep learning framework, which is implemented in C++, as well as Mocha.jl [47], a Julia deep learning framework inspired by Caffe. We used the 2012 ImageNet Large Scale Visual Recognition Challenge [39] dataset with each image resized to 224×224 for our evaluation.

The Latte system infrastructure is built on top of Julia's ParallelAccelerator.jl package [30]. We extend ParallelAccelerator to generate both coprocessor Intel offload and distributed memory MPI codes. We use the Intel® C++ Com-

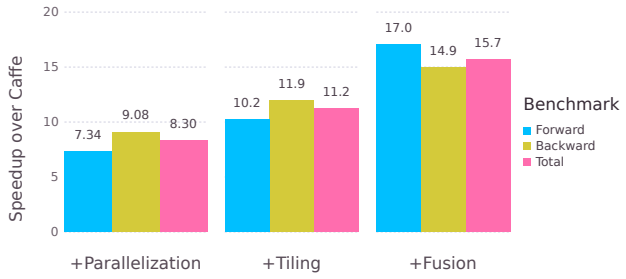


Figure 13: Effect of individual optimizations on speedup in the microbenechmark.

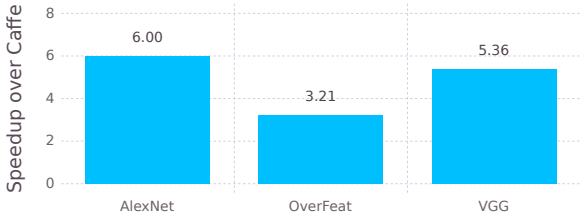


Figure 14: Speedup of Latte over Caffe on the ImageNet models.

piler (ICC) v15.0.2 with "-O3" flag for compilation of the generated C++ code. Our evaluation uses Julia v0.4.

Figures 13-17 show speedups, so higher is better.

7.1 Single Node Evaluation

For single node and accelerator performance, we use a machine with a dual-socket Intel Xeon E5-2699 v3 x86 CPU with 36 cores having 132GB system memory and two Intel Xeon Phi 7110P coprocessors.

7.1.1 Benefits of Cross-Layer Fusion

To understand the performance benefits of cross layer optimizations in Latte, we used a microbenchmark consisting of only the first three layers of the VGG network. Latte is able to perform tiling and fusion of the convolution, pooling, and ReLU layers simultaneously, resulting in data locality and synchronization benefits. Figure 13 shows the speedup of Latte over Caffe's standard library approach. With the parallelization strategy described in Section 5.4.3, the Latte compiler outperforms Caffe by more than 7×. Furthermore, the Latte compiler equipped with advanced optimizations such as cross-layer fusion, tiling, and vectorization is able to achieve 17.0×, 15.0×, and 15.7× speedup for forward, backward, and forward+backward runs, respectively. Our results show that a neural network implemented using a high-level specification can enjoy impressive training-time speedups as a result of cross-layer optimizations that are impossible when using highly-tuned layer-specific libraries.

7.1.2 ImageNet Models

The ImageNet Large Scale Visual Recognition Challenge [40] is an annual benchmark challenge in object category classification and detection on hundreds of object categories and millions of images. To evaluate Latte's overall performance, we implemented three well-known ImageNet models, AlexNet [36], OverFeat [41], and VGG [42], using the standard, publicly available configurations [16]. The results are shown in Figure 14. We observe between 5-6× performance improvement for the AlexNet and VGG models, and a 3.2× improvement for the OverFeat model. These results are less dramatic than the cross-layer optimization microbenchmark due to the cost of the overall computation of the network, including layers that cannot be optimized with cross-layer optimizations, as well as reduced benefit from tiling as the spatial dimension of the data gets smaller after each pooling layer. Figure 15 provides a breakdown of speedup over Caffe for the first four groups of Convolution+ReLU+Pooling layers in the VGG network. As the problem size reduced after a pooling layer, the effect of our optimizations were less pronounced. For example, tiling is less effective because more of the data fits into the cache. In `Group 4` especially, we see less speedup as this group contains two convolution layers followed by a pooling layer; in this case, we cannot fuse the convolution layers due to dependencies along the third dimension. The OverFeat network uses 2-4× the number of filters in the later convolution layers than AlexNet, increasing the size of the final GEMM calls for the fully-connected layers. This results in more of the compute time to be spent in MKL calls executing the fully-connected layers in both implementations. Because both Latte and Caffe use MKL, we see less speedup as they have the same performance for computing these fully-connected layers.

7.1.3 Comparison with Mocha

Figure 16 shows Latte's speedup over Julia's Mocha.jl library [47] as a reference baseline for a high-level implementation of a deep learning framework. We observe orders-of-magnitude performance improvements: 37.9× for AlexNet, 16.2× for OverFeat, and 41× for VGG. We see these improvements for two reasons. First, like Caffe, Mocha does not use parallelization or tiling when computing layers. Second, Mocha's code that does not call into MKL executes in Julia, which is unable to match the performance of C++ code for these compute intensive operations. Because of this, Mocha provides some kernel implementations in C++ to improve performance, which were used when collecting the numbers in Figure 16. However, these native kernel extensions were still unable to match Latte's synthesized code.

7.1.4 Xeon+Xeon Phi Accelerator Performance

To evaluate Latte's ability to leverage a Xeon Phi coprocessor attached to a Xeon host node, we measured the throughput in terms of number of images processed per second. Be-

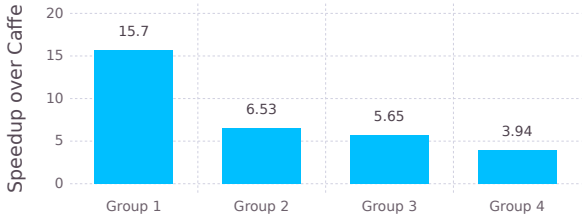


Figure 15: Breakdown of speedup for the first four groups of Convolution+ReLU+Pooling layers of VGG.

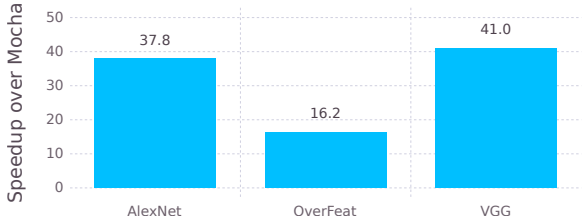


Figure 16: Speedup of Latte over Mocha on the ImageNet models.

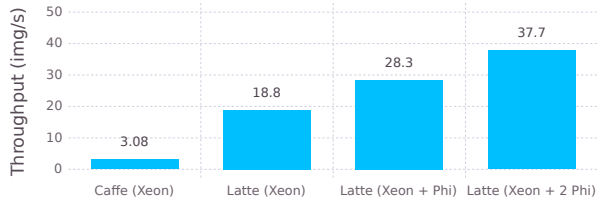


Figure 17: Throughput (number of images processed per second) results when adding Xeon Phi coprocessor with Xeon host.

cause Latte is able to hide the latency of communication by overlapping it with host computation (described in Section 6.1), we see an improvement in images processed per second as the number of Xeon Phi cards increase (as shown in Figure 17). That is, each Xeon Phi card adds an additional 50% throughput. In our current implementation, Xeon Phi throughput is limited by the time spent communicating computed gradients back from the Xeon Phi at the end of each chunk computation.

7.2 Cluster Evaluation

We evaluate Latte's scalability on the Cori Supercomputer and a commodity cluster of Intel Xeon processors. This evaluation was done to demonstrate that Latte is able to support distributed memory execution without any fundamental limitations. The Cori Supercomputer (Phase 1) has 1,630 compute nodes interconnected with a Cray Aries high speed "dragonfly" topology. Each compute node has a dual socket Intel Xeon E5-2698 v3 processor with 16 cores per socket and 128 GB of memory. The commodity cluster has 128

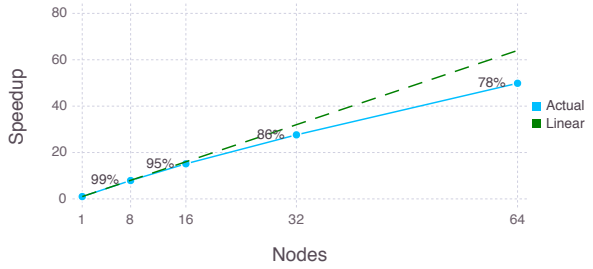


Figure 18: Scaling results using a fixed batch size of 512. For a set of nodes N, each node gets a $512/N$ chunk of the batch to process. The dashed-line shows near-linear scaling.

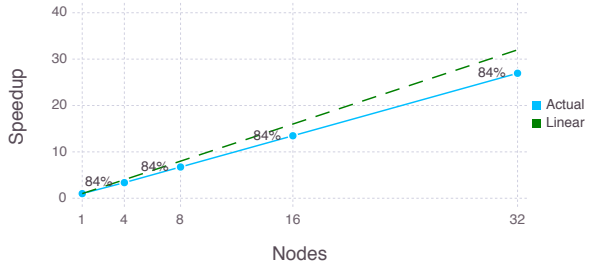


Figure 19: Scaling results using a fixed batch size of 64 per worker using AlexNet training on the ImageNet dataset.

nodes with each node configured with an Intel Xeon E5-2697 v3 processor and 64GB of system memory. It is also equipped with Lustre file system and Infiniband interconnects.

7.2.1 Cori

To evaluate Latte's performance on Cori, we measure Latte's scaling efficiency with respect to the throughput of images trained. For this evaluation, we fix the batch size to 512 and evenly partition the batch amongst compute nodes. We use the VGG network model for the evaluation. Figure 18 shows scaling curve compared to the ideal, linear speedup for 1 to 64 nodes. The drop in efficiency as the number of nodes increases can be attributed to Latte being less efficient on smaller batch sizes due to the reduction in the amount of available parallelism.

7.2.2 Commodity Cluster

For the Xeon commodity cluster evaluation, we fix a batch size of 64 per node. Figure 19 shows near-linear scaling that is constant with respect to the size of the model. That is, as the number of workers/nodes increase, the cost of communication required remains constant. Our results are consistent with those reported in Baidu's Deep Image system [46] which used the same strategy for asynchronous gradient communication.

Goodfellow et al. [24]	99.55%
Adam [15]	99.63%
Latte	99.20%
Latte (sequential)	99.20%

Figure 20: MNIST Top-1 Accuracy

7.3 Accuracy with Gradient Approximation

To evaluate the effect of lossy gradient calculations as described in Section 6, we trained a simple network configuration on the MNIST dataset based on the one described in Microsoft's Project Adam work [15]. Our results in Figure 20 showed identical accuracy for both versions, indicating that the noise introduced via parallelization in our system did not contribute to a degradation in accuracy.

8. Related Work

Low-Level Deep Learning Libraries: cuDNN [14] is the de facto standard for accelerating deep learning on NVIDIA GPUs. cuDNN provides highly efficient implementations of various layers found in deep neural networks. NNPACK [21] and PCL-DNN [18] similarly provide efficient layer implementations focused on targeting x86 processors. Unlike Latte, these static libraries are unable to dynamically fuse across layers.

Graph Engines/Toolkits: There exits a variety of modern computation graph engines and toolkits including CNTK [5], Theano [10], TensorFlow [4], and Torch [17]. Graph engines improve composability and expressibility by supporting layers expressed at a higher-level. Unlike Latte, these libraries do not provide a high-level abstraction using neuron, ensemble, and connection for describing networks. Furthermore, expressing layers at a high-level comes at a large performance cost. Thus, many of these graph engines expose bindings to low-level libraries like cuDNN, meaning they suffer from the same lack of opportunity for cross-layer optimizations. To our knowledge no graph engine supports fusion of non-elementwise layers like convolution and pooling (Section 5.4.2).

Deep Learning Frameworks: Caffe [32] is one of the most popular open-source deep learning frameworks. It has C++ and CUDA backends, as well as bindings to NVIDIA's cuDNN. Mocha.jl [47] is a deep learning framework written in Julia. Its design and implementation was heavily modeled after Caffe. These frameworks are built on top of graph-based engines and low-level libraries, and thus suffer from the same limitations.

Distributed Deep Learning: Google's DistBelief [19] pioneered large-scale distributed deep learning by presenting new optimization methods that mapped efficiently to distributed training. Microsoft's Project Adam [15] and Baidu's Deep Image system [46] are modern examples of highly optimized implementations of distributed deep learning. Latte's cluster code generation replicates the synchronization strategy used in Deep Image [46] and our evaluation results were consistent with their reported scaling efficiency.

Domain-Specific Languages: Halide [38] demonstrated that high-performance image processing code could be written productively using a DSL. SEJITS [12] and Delite [13] provide infrastructure for the creation of DSLs. Latte is a DSL built with ParallelAcclerator.jl [30], a framework for accelerating Julia that was inspired by both the SEJITS and Delite projects.

9. Conclusion

Latte provides a high-level abstraction in the form of a concise and simple DSL to improve programmer productivity for implementing neural networks. In addition to improved programmability, Latte's abstraction enables optimizations that are impossible to do in static libraries and difficult to do on general code. We demonstrated Latte's ability to outperform standard approaches in micro-benchmarks as well as on popular neural network models. Latte's runtime supports training networks on heterogeneous and distributed memory systems, an important feature for accelerating the training of real-world models. Our hope is that Latte will be useful to DNN researchers by bridging the time gap between algorithm development to prototyping to high-performance implementation. Latte is released in the open source and the latest version of the code including a standard library with common DNN layers can be found at `https://github.com/IntelLabs/Latte.jl`. In future, we would like to extend Latte to support complex network models used in neural simulators like Nengo [8]. While it is important to optimize DNNs for CPUs given their widespread use in large-scale computing clusters, several optimizations that we perform including cross-layer optimizations and overlapping communication-computation will benefit GPUs and we would like to extend Latte to generate efficient GPU code as a part of future work.

Acknowledgments

We would like to thank Lindsey Kuper, Brian T. Lewis, and Justin Gottschlich for their valuable feedback that helped to improve the presentation. We are grateful to Paul Hargrove for getting us access to the Cori Supercomputer. We are also thankful to the anonymous reviewers for their comments and suggestions. 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 ASPIRE Lab industrial sponsors and affiliates Intel, Google, Hewlett-Packard, Huawei, LGE, NVIDIA, Oracle, and Samsung.

References

[1] Effective Use of the Intel Compiler's Offload Features. URL https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features.

[2] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, Santa Clara, USA, 2009. ISBN 630813-054US.

[3] Torch NN. https://github.com/torch/nn, 2015.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/.

[5] A. Agarwal, E. Akchurin, C. Basoglu, G. Chen, S. Cyphers, J. Droppo, A. Eversole, B. Guenter, M. Hillebrand, R. Hoens, X. Huang, Z. Huang, V. Ivanov, A. Kamenev, P. Kranen, O. Kuchaiev, W. Manousek, A. May, B. Mitra, O. Nano, G. Navarro, A. Orlov, M. Padmilac, H. Parthasarathi, B. Peng, A. Reznichenko, F. Seide, M. L. Seltzer, M. Slaney, A. Stolcke, Y. Wang, H. Wang, K. Yao, D. Yu, Y. Zhang, and G. Zweig. An introduction to computational networks and the computational network toolkit. Technical Report MSR-TR-2014-112, August 2014. URL http://research.microsoft.com/apps/pubs/default.aspx?id=226641.

[6] S. Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4):185 – 196, 1993. ISSN 0925-2312. doi: http://dx.doi.org/10.1016/0925-2312(93)90006-O. URL http://www.sciencedirect.com/science/article/pii/092523129390006O.

[7] A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan. On optimizing machine learning workloads via kernel fusion. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 173–182, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. URL http://doi.acm.org/10.1145/2688500.2688521.

[8] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith. Nengo: a Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7, 2013.

[9] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 59. ACM, 2009.

[10] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[11] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A Fast Dynamic Language for Technical Computing. *CoRR*, abs/1209.5145, 2012. URL http://arxiv.org/abs/1209.5145.

[12] B. Catanzaro, S. Kamil, Y. Lee, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization.

[13] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. *ACM SIGPLAN Notices*, 46(8):35–46, 2011.

[14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014. URL http://arxiv.org/abs/1410.0759.

[15] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi.

[16] S. Chintala. Convnet Benchmarks. https://github.com/soumith/convnet-benchmarks, 2015.

[17] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A MATLAB-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[18] D. Das, S. Avancha, D. Mudigere, K. Vaidyanathan, S. Sridharan, D. D. Kalamkar, B. Kaul, and P. Dubey. Distributed deep learning using synchronous stochastic gradient descent. *CoRR*, abs/1602.06709, 2016. URL http://arxiv.org/abs/1602.06709.

[19] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[20] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[21] M. Dukhan. NNPACK. https://github.com/Maratyszcza/NNPACK, 2016.

[22] F. Gers. Long short-term memory in recurrent neural networks.

[23] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[24] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[25] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[26] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet

classification. *CoRR*, abs/1502.01852, 2015. URL http://arxiv.org/abs/1502.01852.

[27] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[28] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[29] Intel. Intel Data Analytics Acceleration Library (DAAL). https://software.intel.com/en-us/intel-daal, 2015.

[30] Intel Labs. ParallelAccelerator.jl. https://github.com/IntelLabs/ParallelAccelerator.jl, 2015.

[31] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

[32] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[33] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[34] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997.

[35] A. Krizhevsky. cuda-convnet2. https://github.com/akrizhevsky/cuda-convnet2, 2015.

[36] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[37] Microsoft. CNTK. https://github.com/Microsoft/CNTK/tree/7d3e84e7733c1c965d995e28ff4bac60f166a03b, 2015.

[38] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[39] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[40] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[41] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. *CoRR*, abs/1312.6229, 2013. URL http://arxiv.org/abs/1312.6229.

[42] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[43] Skymind. Deep Learning for Java (DL4J). http://deeplearning4j.org/architecture.html, 2015.

[44] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

[45] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop. *COURSERA: Neural Networks for Machine Learning*, 2012.

[46] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep Image: Scaling up Image Recognition. *CoRR*, abs/1501.02876, 2015. URL http://arxiv.org/abs/1501.02876.

[47] C. Zhang. Mocha.jl. https://github.com/pluskid/Mocha.jl, 2015.