Caffe* Optimized for Intel® Architecture: Applying Modern Code Techniques

Abstract
This paper demonstrates a special version of Caffe*—a deep learning framework originally developed by the Berkeley Vision and Learning Center (BVLC)—that is optimized for Intel® architecture. This version of Caffe, known as Caffe optimized for Intel architecture, is currently integrated with the latest release of Intel® Math Kernel Library 2017 and is optimized for Intel® Advanced Vector Extensions 2 and will include Intel Advanced Vector Extensions 512 instructions. This solution is supported by Intel® Xeon® processors and Intel® Xeon Phi™ processors, among others. This paper includes performance results for a CIFAR-10* image-classification dataset, and it describes the tools and code modifications that can be used to improve computational performance for the BVLC Caffe code and other deep learning frameworks.

Introduction
Deep learning is a subset of general machine learning that in recent years has produced groundbreaking results in image and video recognition, speech recognition, natural language processing (NLP), and other big-data and data-analytics domains. Recent advances in computation, large datasets, and algorithms have been key ingredients behind the success of deep learning, which works by passing data through a series of layers, with each layer extracting features of increasing complexity.

Table of Contents
Abstract .................................. 1
Image Classification .................. 2
Initial Performance Profiling ...... 2
Code Optimizations ................. 5
Scalar and Serial Optimizations .. 5
Code Vectorization ................. 5
Generic Code Optimization ...... 5
CPU-Specific, System-Specific ..... 6
Code Parallelization with OpenMP .. 6
Neural-Network-Layers Optimization 6
Convolution Layer ............... 6
Pooling or Subsampling .......... 7
Softmax and Loss Layer .......... 7
Rectified Linear Unit (ReLU) .... 7
Conclusion .............................. 8

Figure 1. Each layer in a deep network is trained to identify features of higher complexity—this figure shows a small subset of the features of a deep network projected down to the pixels space (the gray images on the left) and the corresponding images (colored images on the right) that activate those features.

Supervised deep learning requires a labeled dataset. Three popular types of supervised deep networks are multilayer perceptrons (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). In these networks, the input is passed through a series of linear and non-linear transformations as it progresses.
through each layer, and an output is produced. An error and the respective cost of the error are then computed before a gradient of the costs of the weights and activations in the network is computed and iteratively backward propagated to lower layers. Finally, the weights or models are updated based on the computed gradient.

In MLPs, the input data at each layer (represented by a vector) is first multiplied by a dense matrix unique to that layer. In RNNs, the dense matrix (or matrices) is the same for every layer (the layer is recurrent), and the length of the network is determined by the length of the input signal. CNNs are similar to MLPs, but they use a sparse matrix for the convolutional layers. This matrix multiplication is represented by convolving a 2-D representation of the weights with a 2-D representation of the layer’s input. CNNs are popular in image recognition, but they are also used for speech recognition and NLP. For a detailed explanation of CNNs, see “CS231n Convolutional Neural Networks for Visual Recognition” at http://cs231n.github.io/convolutional-networks/.

**Image Classification**

The CIFAR-10 dataset consists of 60,000 color images, each with dimensions of 32 × 32, equally divided and labeled into the following 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The classes are mutually exclusive; there is no overlap between different types of automobiles (such as sedans or sports utility vehicles [SUVs]) or trucks (which includes only big trucks)—neither group includes pickup trucks (see Figure 2).

When Intel tested the Caffe frameworks, we used the CIFAR-10 full-sigmoid model, a CNN model with multiple layers including convolution, max pooling, batch normalization, fully connected, and softmax layers. For layer descriptions, refer to the Code Parallelization with OpenMP* section.

**Caffe**

Caffe* is a deep learning framework developed by the Berkeley Vision and Learning Center (BVLC) and community contributors. This paper refers to that original version of Caffe as “BVLC Caffe.”

In contrast, Caffe optimized for Intel® architecture is a specific, optimized fork of the BVLC Caffe framework.² Caffe optimized for Intel architecture is currently integrated with the latest release of Intel® Math Kernel Library (Intel® MKL) 2017, and it is optimized for Intel® Advanced Vector Extensions 2 (Intel® AVX2) and will include Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions, which are supported by Intel® Xeon® processors and Intel® Xeon Phi™ processors, among others. For a detailed description of compiling, training, fine-tuning, testing, and using the various tools available, read “Training and Deploying Deep Learning Networks with Caffe* Optimized for Intel® Architecture” at https://software.intel.com/en-us/articles/training-and-deploying-deep-learning-networks-with-cafe-optimized-for-intel-architecture.

This paper describes the performance of Caffe optimized for Intel architecture compared to BVLC Caffe running on Intel architecture, and it discusses the tools and code modifications used to improve computational performance for the Caffe framework. It also shows performance results from using the CIFAR-10* image-classification dataset (https://www.cs.toronto.edu/~kriz/cifar.html) and the CIFAR-10 full-sigmoid model that composes layers of convolution, max and average pooling, and batch normalization: (https://github.com/BVLC/caffe/blob/master/examples/cifar10/cifar10_full_sigmoid_train_test_bn.prototxt).

To download the source code for the tested Caffe frameworks, visit the following:

- BVLC Caffe: https://github.com/BVLC/caffe
- Caffe optimized for Intel architecture: https://github.com/intelcaffe/caffe

**Initial Performance Profiling**

One method for benchmarking Caffe optimized for Intel architecture and BVLC Caffe is using the time command, which computes the layer-by-layer forward and backward propagation time. The time command is useful for measuring the time spent in each layer and for providing the relative execution times for different models:

```
$ ./build/tools/caffe time \\
--model=examples/cifar10/cifar10__full__sigmoid__train__test__bn.prototxt \\
-iterations 1000
```

In this context, an iteration is defined as one forward and backward pass over a batch of images. The previous command returns the average execution time per iteration for 1,000 iterations per layer and for the entire network. Figure 3 shows the full output.
In our testing, we used a dual-socket system with one Intel Xeon processor E5-2699 v3 at 2.30 GHz per socket, 18 physical cores per CPU, and Intel® Hyper-Threading Technology (Intel® HT Technology) disabled. This dual-socket system had 36 cores in total, so the default number of OpenMP* threads, specified by the OMP_NUM_THREADS environment variable, was 36 for our tests, unless otherwise specified (note that we recommend letting Caffe optimize for Intel architecture automatically specify the OpenMP environment rather than setting it manually). The system also had 64 GB of DDR4 memory installed, operating at a frequency of 2,133 MHz.

Using those numbers, this paper demonstrates the performance results of code optimizations made by Intel engineers. We used the following tools for performance monitoring:

- Callgrind* from Valgrind* toolchain
- Intel® VTune™ Amplifier XE 2017 beta

Intel VTune Amplifier XE tools provide the following information:

- Functions with the highest total execution time (hotspots)
- System calls (including task switching)
- CPU and cache usage
- OpenMP multithreading load balance
- Thread locks
- Memory usage

We can use the performance analysis to find good candidates for optimization, such as code hotspots and long function calls. Figure 4 shows important data points from the Intel VTune Amplifier XE 2017 beta summary analysis running 100 iterations. The Elapsed Time, Figure 4a, is 37 seconds. The CPU Time, shown below Elapsed Time, is 1,306 seconds—this is slightly less than 37 seconds multiplied by 36 cores (1,332 seconds). CPU Time is the combined duration sum of all threads (or cores, because hyper-threading was disabled in our test) contributing to the execution.

The source code of the BVLC Caffe package contains no #pragma omp parallel code line. In the BVLC Caffe code, there is no explicit use of the OpenMP library for multithreading. However, OpenMP threads are used inside of the Intel MKL to parallelize some of the math-routine calls. To confirm this parallelization, we can look at a bottom-up tab view (see Figure 5 and review the function calls with Effective Time by Utilization [at the top] and the individual thread timelines [at the bottom]).

Figure 5 shows the function-call hotspots for BVLC Caffe on the CIFAR-10 dataset.
The `gemm_omp_driver_v2` function—part of `libmkl_intel_thread.so`—is a general matrix-matrix (GEMM) multiplication implementation of Intel MKL. This function uses OpenMP multithreading behind the scenes. Optimized Intel MKL matrix-matrix multiplication is the main function used for forward and backward propagation—that is, for weight calculation, prediction, and adjustment. Intel MKL initializes OpenMP multithreading, which usually reduces the computation time of GEMM operations. However, in this particular case—convolution for 32 × 32 images—the workload is not big enough to efficiently utilize all 36 OpenMP threads on 36 cores in a single GEMM operation. Because of this, a different multithreading-parallelization scheme is needed, as will be shown later in this paper.

To demonstrate the overhead of OpenMP thread utilization, we run code with the `OMP_NUM_THREADS=1` environment variable, and then compare the execution times for the same workload: 31.1 seconds instead of 37 seconds (see the Elapsed Time section in Figure 4a and Figure 6a). By using this environment variable, we force OpenMP to create only a single thread and to use it for code execution. The resulting almost six seconds of runtime difference in the BVLC Caffe code implementation provides an indication of the OpenMP thread initialization and synchronization overhead.

With this analysis setup, we identified three main candidates for performance optimization in the BVLC Caffe implementation: the `im2col_cpu`, `col2im_cpu`, and `PoolingLayer::Forward_cpu` function calls (see Figure 6b).
Code Optimizations

The Caffe optimized for Intel architecture implementation for the CIFAR-10 dataset is about 13.5 times faster than BVLC Caffe code (20 milliseconds [ms] versus 270 ms for forward-backward propagation). Figure 7 shows the results of our forward-backward propagation averaged across 1,000 iterations. The left column shows the BVLC Caffe results, and the right column shows the results for Caffe optimized for Intel architecture.

For an in-depth description of these individual layers, refer to the Neural-Network-Layers Optimization Results section below.

For more information about defining calculation parameters for layers, visit http://caffe.berkeleyvision.org/tutorial/layers.html.

The following sections describe the optimizations used to improve the performance of various layers. Our techniques followed the methodology guidelines of Intel® Modern Code Developer Code, and some of these optimizations rely on Intel MKL 2017 math primitives. The optimization and parallelization techniques used in Caffe optimized for Intel architecture are presented here to help you better understand how the code is implemented and to empower code developers to apply these techniques for other machine learning and deep learning applications and frameworks.

Scalar and Serial Optimizations

Code Vectorization

After profiling the BVLC Caffe code and identifying hotspots—function calls that consumed most of the CPU time—we applied optimizations for vectorization. These optimizations included the following:

- Basic Linear Algebra Subprograms (BLAS) libraries (switch from Automatically Tuned Linear Algebra System [ATLAS*] to Intel MKL)
- Optimizations in assembly (Xbyak just-in-time [JIT] assembler)
- GNU Compiler Collection* (GCC*) and OpenMP code vectorization

BVLC Caffe has the option to use Intel MKL BLAS function calls or other implementations. For example, the GEMM function is optimized for vectorization, multithreading, and better cache traffic. For better vectorization, we also used Xbyak—a JIT assembler for x86 (IA-32) and x64 (AMD64* or x86-64). Xbyak currently supports the following list of vector-instruction sets: MMX™ technology, Intel® Streaming SIMD Extensions (Intel® SSE), Intel SSE2, Intel SSE3, Intel SSE4, floating-point unit, Intel AVX, Intel AVX2, and Intel AVX-512.

The Xbyak assembler is an x86/x64 JIT assembler for C++, a library specifically created for developing code efficiently. The Xbyak assembler is provided as header-only code. It can also dynamically assemble x86 and x64 mnemonics.

JIT binary-code generation while code is running allows for several optimizations, such as quantization, an operation that divides the elements of a given array by the elements of a second array, and polynomial calculation, an operation that creates actions according to constant, variable x, add, sub, mul, div, and so on. With the support of Intel AVX and Intel AVX2 vector-instruction sets, Xbyak can get a better vectorization ratio in the code implementation of Caffe optimized for Intel architecture. The latest version of Xbyak has Intel AVX-512 vector-instruction-set support, which can improve computational performance on the Intel Xeon Phi processor x200 product family. This improved vectorization ratio allows Xbyak to process more data simultaneously with single instruction, multiple data (SIMD) instructions, which more efficiently utilize data parallelism. We used Xbyak to vectorize this operation, which improved the performance of the process pooling layer significantly. If we know the pooling parameters, we can generate assembly code to handle a particular pooling model for a specific pooling window or pooling algorithm. The result is a plain assembly that is proven to be more efficient than C++ code.

Generic Code Optimizations

Other serial optimizations included:

- Reducing algorithm complexity
- Reducing the amount of calculations
- Unwinding loops

Common-code elimination is one of the scalar optimization techniques that we applied during the code optimization. This was done in order to predetermine what can be calculated outside of the innermost for-loop.

For example, consider the following code snippet:

```c
for (int h_col = 0; h_col < height_col; ++h_col)
for (int w_col = 0; w_col < width_col; ++w_col)
int h_im = h_col + stride_h - pad_h + h_offset;
int w_im = w_col + stride_w - pad_w + w_offset;
```
In the third line of this code snippet, for the `h_im` calculation, we are not using a `w_col` index of the innermost loop. But this calculation will still be performed for every iteration of the innermost loop. Alternatively, we can move this line outside of the innermost loop with the following code:

```c
for (int h_col = 0; h_col < height_col; ++h_col) {
    int b_im = h_im = h_col; 
    int w_col = 0; w_col < width_col; ++w_col) {
        int w_im = w_col * stride_w + pad_w + w_offset;
```

CPU-Specific, System-Specific, and Other Generic Code-Optimization Techniques

The following additional generic optimizations were applied:

- Improved `im2col_cpu/col2im_cpu` implementation
- Complexity reduction for batch normalization
- CPU/system-specific optimizations
- Use one core per computing thread
- Avoid thread movement

Intel VTune Amplifier XE 2017 beta identified the `im2col_cpu` function as one of the hotspot functions—making it a good candidate for performance optimization. The `im2col_cpu` function is a common step in performing direct convolution as a GEMM operation for using the highly optimized BLAS libraries. Each local patch is expanded to a separate vector, and the whole image is converted to a larger (more memory-intensive) matrix whose rows correspond to the multiple locations where filters will be applied.

One of the optimization techniques for the `im2col_cpu` function is index-calculation reduction. The BVLC Caffe code had three nested loops for going through image pixels:

```c
for (int c_col = 0; c_col < channela_col; ++c_col) {
    for (int h_col = 0; h_col < height_col; ++h_col, 
```

In this code snippet, BVLC Caffe was originally calculating the corresponding index of the `data_col` array element, although the indexes of this array are simply processed sequentially. Therefore, four arithmetic operations (two additions and two multiplications) can be substituted by a single index-incrementation operation. In addition, the complexity of the conditional check can be reduced due to the following:

```c
if (((unsigned) x) < ((unsigned) N))
```

To avoid thread movement by the operating system, we used the OpenMP affinity environment variable, `KMP_AFFINITY=compact,granularity=fine`. Compact placement of neighboring threads can improve performance of GEMM operations because all threads that share the same last-level cache (LLC) can reuse previously prefetched cache lines with data.


Code Parallelization with OpenMP*

Neural-Network-Layers Optimization Results

The following neural network layers were optimized by applying OpenMP multithreading parallelization to them:

- Convolution
- Deconvolution
- Local response normalization (LRN)
- Softmax
- Concatenation
- Utilities for OpenBLAS* optimization—such as the `vPowx` operation, `caffe_set`, `caffe_copy`, and `caffe_rng_bernoulli`
- Pooling
- Batch normalization
- Data
- Eltwise

Convolution Layer

The convolution layer, as the name suggests, convolves the input with a set of learned weights or filters, each producing one feature map in the output image. This optimization prevents under-utilization of hardware for a single set of input feature maps.

```c
void ConvolutionLayer<% Dtype%>(Forward_cpu(constat vector<%Blob%<%Dtype%>>*>& bottom, constat vector<%Blob%<%Dtype%>>*>& top) {
```

In BVLC Caffe, the original code had the conditional check if `x >= 0 && x < N`, where `x` and `N` are both signed integers, and `N` is always positive. By converting the type of those integer numbers into unsigned integers, the interval for the comparison can be changed. Instead of running two compares with logical `AND`, a single comparison is sufficient after type casting:

```c
if (((unsigned) x) < ((unsigned) N))
```
We process \( k = \min(\text{num\_threads}, \text{batch\_size}) \) sets of input\_feature maps; for example, \( k \text{ im2col} \) operations happen in parallel, and \( k \) calls to Intel MKL. Intel MKL is switched to a single-threaded execution flow automatically, and performance overall is better than it was when Intel MKL was processing one batch. This behavior is defined in the source code file, src/caffe/layers/base_conv_layer.cpp. The implementation optimized OpenMP multithreading from src/caffe/layers/conv_layer.cpp—the file location with the corresponding code.

### Pooling or Subsampling

Max-pooling, average-pooling, and stochastic-pooling (not implemented yet) are different methods for downsampling, with max-pooling being the most popular method. The pooling layer partitions the results of the previous layer into a set of usually non-overlapping rectangular tiles. For each such sub-region, the layer then outputs the maximum, the arithmetic mean, or (in the future) a stochastic value sampled from a multinomial distribution formed from the activations of each tile.

Pooling is useful in CNNs for three main reasons:

- Pooling reduces the dimensionality of the problem and the computational load for upper layers.
- Pooling lower layers allows the convolutional kernels in higher layers to cover larger areas of the input data and therefore learn more complex features; for example, a lower-layer kernel usually learns to recognize small edges, whereas a higher-layer kernel might learn to recognize sceneries like forests or beaches.
- **Max-pooling** provides a form of translation invariance. Out of eight possible directions in which a 2 × 2 tile (the typical tile for pooling) can be translated by a single pixel, three will return the same max value; for a 3 × 3 window, five will return the same max value.

Pooling works on a single feature map, so we used Xbyak to make an efficient assembly procedure that can create average-to-max pooling for one or more input feature maps. This pooling procedure can be implemented for a batch of input feature maps when you run the procedure parallel to OpenMP.

The pooling layer is parallelized with OpenMP multithreading; because images are independent, they can be processed in parallel by different threads:

```
#pragma omp parallel for
for (int i = 0; i < num_blobs; i++) {
    caffe_div(inner_num, top_data + i*inner_num_, scale_data,
              top_data + i*inner_num_);
}
```

With the collapse(2) clause, OpenMP #pragma omp parallel spreads on to both nested for-loops, iterates though images in the batch and image channels, combines the loops into one, and parallelizes the loop.

### Softmax and Loss Layer

The **loss (cost)** function is the key component in machine learning that guides the network training process by comparing a prediction output to a target or label and then readjusting weights to minimize the cost by calculating gradients—partial derivatives of the weights with respect to the loss function.

The **softmax (normalized exponential)** function is the gradient-log normalizer of the categorical probability distribution. In general, this is used to calculate the possible results of a random event that can take on one of \( K \) possible outcomes, with the probability of each outcome separately specified. Specifically, in multinomial logistic regression (a multi-class classification problem), the input to this function is the result of \( K \) distinct linear functions, and the predicted probability for the \( j \)th class for sample vector \( x \) is:

\[
P(y = j | x) = \frac{e^{x^T w_j}}{\sum_{k=1}^{K} e^{x^T w_k}}
\]

OpenMP multithreading, when applied for these calculations, is a method of parallelizing by using a master thread to fork a specified number of subordinate threads as a way of dividing a task among them. The threads then run concurrently as they are allocated to different processors. For example, in the following code, parallelized individual arithmetic operations with independent data access are implemented through division by the calculated norm in different channels:

```
1 // division
2 #ifdef OPENMP
3 #pragma omp parallel for
4 #endif
5 for (int j = 0; j < channels; j++) {
6    caffe_div(inner_num, top_data + j*inner_num_, scale_data,
7              top_data + j*inner_num_);
8 }
```

### Rectified Linear Unit (ReLU) and Sigmoid—Activation/Neuron Layers

ReLU is currently the most popular non-linear functions used in deep learning algorithms. Activation/neuron layers are element-wise operators that take one bottom blob and produce one top blob of the same size. (A blob is the standard array and unified memory interface for the framework. As data and derivatives flow through the network, Caffe stores, communicates, and manipulates the information as blobs.)

The ReLU layer takes input value \( x \) and computes the output as \( x \) for positive values and scales them by negative_slope for negative values:

\[
f(x) = \begin{cases} 
  x, & \text{if } x > 0 \\
  \text{negative\_slope} \times x, & \text{otherwise}
\end{cases}
\]

The default parameter value for negative_slope is zero, which is equivalent to the standard ReLU function of taking max(\( x, 0 \)). Due to the data-independent nature of the activation process, each blob can be processed in parallel as shown on the next page:
Similar parallel calculations can be used for backward propagation, as shown below:

```c
template <typename Dtype>
void ReLU(const vector< Blob<Dtype> > & top, const vector< Blob<Dtype> > & bottom)
{
  if (propagate_data(0)) {
    for (int i = 0; i < count; ++i) {
      top_data[i] = std::max(bottom_data[i], Dtype(0));
    }
  } else {
    Dtype negative_slope = this->layer_param_.relu_param().negative_slope();
    for (int i = 0; i < count; ++i) {
      Dtype bottom_data[i] = bottom[i] <= 0 ? 0 : bottom[i] * negative_slope;
    }
  }
}
```

In the same fashion, the sigmoid function $S(x) = \frac{1}{1+e^{-x}}$ can be parallelized in the following way:

```c
template <typename Dtype>
void ReLU(const vector< Blob<Dtype> > & top, const vector< Blob<Dtype> > & bottom)
{
  for (int i = 0; i < count; ++i) {
    Dtype bottom_diff = bottom[i] > 0 ? 1 : 0;
    Dtype top_diff = top_data[i] > 0 ? bottom[i] : 0;
    Dtype negative_slope = this->layer_param_.relu_param().negative_slope();
    Dtype bottom_data[i] = bottom[i] * negative_slope;
  }
}
```

Because Intel MKL does not provide math primitives to implement ReLUs to add this functionality, we tried to implement a performance-optimized version of the ReLU layer with assembly code (via Xbyak). However, we found no visible gain for Intel Xeon processors—perhaps due to limited memory bandwidth. Parallelization of the existing C++ code was good enough to improve the overall performance.

**Conclusion**

The previous section discussed various components and layers of neural networks and how blobs of processed data in these layers were distributed among available OpenMP threads and Intel MKL threads. The CPU Usage Histogram in Figure 8 shows how often a given number of threads ran concurrently after our optimizations and parallelizations were applied.

With Caffe optimized for Intel architecture, the number of simultaneously operating threads is significantly increased. The execution time on our test system dropped from 37 seconds in the original, unmodified run to only 3.6 seconds with Caffe optimized for Intel architecture—improving the overall execution performance by more than 10 times.

As shown in the Elapsed Time section, Figure 8a, there is still some Spin Time present during the execution of this run. As a result, the execution's performance does not scale linearly with the increased thread count (in accordance with Amdahl's law). In addition, there are still serial execution regions in the code that are not parallelized with OpenMP multithreading. Re-initialization of OpenMP parallel regions was significantly optimized for the latest OpenMP library implementations, but it still introduces non-negligible performance overhead. Moving OpenMP parallel regions into the main function of the code could potentially improve the performance even more, but it would require significant code refactoring.

Figure 9 summarizes the described optimization techniques and code rewriting principals that we followed with Caffe optimized for Intel architecture.

In our testing, we used Intel VTune Amplifier XE 2017 beta to find hotspots—good code candidates for optimization and parallelization. We implemented scalar and serial optimizations, including common-code elimination and reduction/simplification of arithmetic operations for loop index and conditional calculations. Next, we optimized the code for vectorization following the general principles described in "Auto-vectorization in GCC" (https://gcc.gnu.org/projects/tree-ssa/vectorization.html). The JIT assembler Xbyak allowed us to use SIMD operations more efficiently.
We implemented multithreading with an OpenMP library inside the neural-network layers, where data operations on images or channels were data-independent. The last step in implementing the Intel Modern Code Developer Code approach involved scaling the single-node application for many-core architectures and a multi-node cluster environment. This is the main focus of our research and implementation at this moment. We also applied optimizations for memory (cache) reuse for better computational performance. For more information see: http://arxiv.org/pdf/1602.06709v1.pdf. Our optimizations for the Intel Xeon Phi processor x200 product family included the use of high-bandwidth MCDRAM memory and utilization of the quadrant NUMA mode.

Caffe optimized for Intel architecture not only improves computational performance, but it enables you to extract increasingly complex features from data. The optimizations, tools, and modifications included in this paper will help you achieve top computational performance from Caffe optimized for Intel architecture.

For more information about the Intel Modern Code Developer Code program, refer to the following publications:

https://software.intel.com/en-us/articles/what-is-code-modernization
https://software.intel.com/modern-code

For more information on machine learning, see:

http://software.intel.com/machinelearning

Figure 9. Step-by-step approach of Intel® Modern Code Developer Code

We implemented multithreading with an OpenMP library inside the neural-network layers, where data operations on images or channels were data-independent. The last step in implementing the Intel Modern Code Developer Code approach involved scaling the single-node application for many-core architectures and a multi-node cluster environment. This is the main focus of our research and implementation at this moment. We also applied optimizations for memory (cache) reuse for better computational performance. For more information see: http://arxiv.org/pdf/1602.06709v1.pdf. Our optimizations for the Intel Xeon Phi processor x200 product family included the use of high-bandwidth MCDRAM memory and utilization of the quadrant NUMA mode.

Caffe optimized for Intel architecture not only improves computational performance, but it enables you to extract increasingly complex features from data. The optimizations, tools, and modifications included in this paper will help you achieve top computational performance from Caffe optimized for Intel architecture.

For more information about the Intel Modern Code Developer Code program, refer to the following publications:

https://software.intel.com/en-us/articles/what-is-code-modernization
https://software.intel.com/modern-code

For more information on machine learning, see:

http://software.intel.com/machinelearning