WHITE PAPER

Artificial Intelligence
Deep Neural Network Training
Intel AI® Builders

Myth Busted: General Purpose CPUs Can't Tackle Deep Neural Network Training

Author

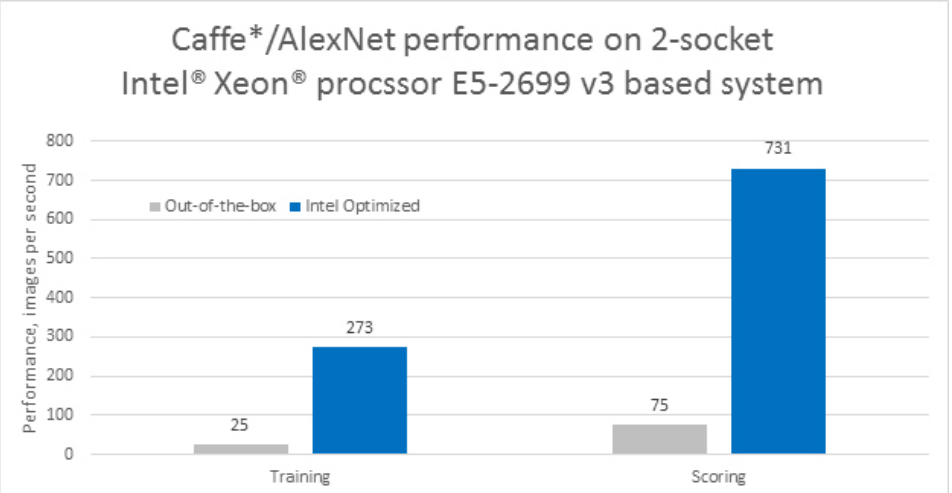
Pradeep Dubey
Intel Fellow at Intel Labs

Table of Contents

Introduction 1
Optimizing Computation 2
Cache Blocking 2
Data Layout 3
Vectorization Register Blocking . . 3
Threading and Work Partitioning . 5

Introduction

Following up on my previous post with respect to "Pushing Machine Learning to a New Level with Intel Xeon and Intel Xeon Phi Processors", I would like to put things into the terms of one of the most popular deep learning frameworks being used today, Caffe*. Caffe is an open source project out of the Berkeley Vision and Learning Center (BVLC) at the University of California at Berkeley and a project that Intel sponsors and actively participates in. Intel is currently investing significant resources in optimizing performance for Deep Neural Network (DNN) workloads and development frameworks on both Intel® Xeon® and Intel® Xeon PhiTM processors. To date projects like Caffe haven't been optimized for CPUs and thus a myth has formed that CPUs are somehow not applicable or cannot support these types of workloads. Applying our initial findings and optimizations to the AlexNet topology in the Caffe framework, we were able to achieve significant speedup of up to 11x and 10x in neural network training and scoring performance respectively (see the chart below). This effectively busts the myth that CPUs are anemic for neural networks allowing developers an alternative platform with many advantages including a simplified programming model and easy access through their pervasive availability in today's cloud infrastructure.



Configuration Info - Hardware: Intel® Xeon® Processor E5-2699 v3, 2 eighteen-core CPUs (45MB LLC, 2.3GHz), Intel® Turbo Boost Technology off, Intel® Hyperthreading Technology off, 64GB of RAM; Operating System: RHEL 6.5 GA x86_64; testing source: internal Intel measurements

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visithttp://www.intel.com/performance. Benchmark Source: Intel Corporation

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

At a high level, the incremental performance seen was achieved through the use of the Intel® Math Kernel Library (Intel® MKL), parallelization using OpenMP*, aligned data allocation, and the use of new performance primitives such as direct batched convolution, pooling, and normalization. These new primitives are coming to future versions of Intel® MKL and Intel® Data Analytics Acceleration Library (Intel® DAAL). Technical preview package demonstrating achievable performance with Caffe on the AlexNet topology is already available for download.

Since this is a technical blog, I'd like to drill down into the details of how we achieved the above results and illustrate some of the amazing optimizations being delivered in our upcoming libraries. I also provide a link at the end of the post where anyone can download the application and perform the same evaluation.

Optimizing Computation

Intel MKL technology preview package contains several architecture-specific algorithmic optimizations which yield highly efficient neural network code. These optimizations are generic and agnostic of a specific architecture, and hence are applicable to a broad spectrum of multi-core and many-core chips. Moreover, while designing the techniques presented in this work, we consciously selected designs that remain efficient for multi-node implementations, where the mini-batch size per node is small. The focus of this section shall be on convolutions, as we believe that fully connected layers which trivially map to GEMM operations and Intel MKL GEMM (Intel MKL 11.2.1 onwards) implementations, yield high efficiency for them.

Cache Blocking

The convolutional layer operation consists of a 6-nested loop as shown in Figure 1. When written in the naïve fashion as in Figure 1, the convolutional operation is bandwidth bound for many instances. It is simple to see that unless the activations and weights completely fit in cache (which is often not the case), the third loop of neural network convolution operation (line 3) pulls in OFH*OFW output activations, (OFH*STRIDE+KH-1)*(OFW*STRIDE+KW-1) input activations (denoted as IFH * IFW), and KH*KW weights, while it performs KH*KW*OFW*OFH multiply-and-accumulate operations. The bytes to flops ratio can be easily computed to be:

B/F = data_size(OFW*OFH + IFW*IFH + KW*KH)/ (2*KS*KH*OFH*OFW).*

```
1.  for ofm = 0 … OFM-1
2.      for ifm= 0… IFM-1
3.          for ofh = 0 … OFH-1
4.              for ofw = 0 … OFW-1
5.                  for kh = 0 … KH-1
6.                      for kw = 0 … KW-1
7.                          output[ofm][ofh][ofw]
    += input[ifm][STRIDE*ofh + kh][stride*ofw +
    kw]*weight[ofm][ifm][kh][kw];
8.            output[ofm][ofh][ofw] += bias[ofm];
```

Figure 1: A naïve 6-nested loop for convolutional layers.

For a typical convolutional neural network layer with OFH=OFW=12, KH=KW=3, STRIDE=1 (as in OverFeat-FAST layer C3,4,5), we obtain a Bytes-Flops (B/F) ratio of 0.54, which is heavily bandwidth bound compared to a machine B/F of 0.07 for a dual socket Intel® Xeon® processor E5-2699 v3 (codename Haswell). The naïve loop therefore is theoretically limited to 12 percent efficiency. Even if we assume that loop 2 onwards the content can be stored in on-chip cache, the B/F ratio improves only to 0.24 and the efficiency is limited to 30 percent.

Clearly there is a need and opportunity to block loops 1 and 2 (over input and output feature maps). We first consider blocking on loop 1 over output feature maps (block-size=OB), and produce the loop structure in Figure 2. If OB output features, each of size OFH*OFW can be stored in the cache then the B/F ratio becomes:

```
B/F    =    data_size*(OB*OFH*OFW   +   OB*IFM*KH*KW
+    IFM*IFH*IFW)/(OB*2*OFW*OFH*KH*KW*IFM)
```

```
1.  for ofm = 0 … OFM/OB-1
2.      for ifm= 0… IFM-1
3.          for ofh = 0 … OFH-1
4.              for ofw = 0 … OFW-1
5.                  for kh = 0 … KH-1
6.                      for kw = 0 … KW-1
7.                          for ob = 0 … OB-1
8.                              output[ofm*OB+ob]
    [ofh][ofw]+= input[ifm][STRIDE*ofh + kh]
    [stride*ofw + kw] *weight[ofm][ifm][kh][kw];
9.              for ob = 0 … OB-1
10.                 output[ofm*OB + ob][ofh][ofw]
    += bias[ofm*OB + ob];
```

Figure 2: A blocked 7-nested loop for convolutional layers, with the loop over output feature maps blocked (loop-1 and loop-7).

Here, we stream through the input features and we reuse each input feature to compute OB output features. For the earlier example for the C5 layer of OverFeat-FAST, we compute the B/F ratio for OB=8 to be: 0.052. This is below the 0.07 machine B/F ratio, and makes the loop compute bound. Indeed many convolutional layers become compute bound when OB is set to the SIMD-Width of the processor. We can further improve the B/F ratio by additionally blocking on loop 1 (Figure 2), wherein the B/F ratio further improves to:

```
B/F    =    data_size*(OB*OFH*OFW    +
OB*IB*KH*KW    +    IB*IFH*IFW)/(IB*OB*2*OFW*OFH*KH*KW)
```

For the example C5 layer (example earlier) we can achieve a B/F ratio of 0.02. Note however, that a B/F ratio below the machine B/F ratio is sufficient, and blocking the loop over input feature maps with a simple SIMD_WIDTH sized block keeps the implementation simple, and B/F ratio within requisite limits.

The backward pass also has a similar loop blocking, wherein

loops 1 and 2 are swapped, and the blocking is performed over the input-feature maps dimension.

The blocking strategy for weight updates is the same as that for forward propagation, that is blocking over the output feature maps.

Data Layout

Following up on the blocked loop structure in Figure 2, we immediately see that data is accessed in a strided manner in the innermost loop 7 (in Figure 2). Clearly there is benefit in laying out data so that the access in the innermost loop is continuous. This aids both a better utilization of cache lines (and hence bandwidth), while improving prefetcher performance.

In this work we lay out all data, including activations and weights with the innermost dimension over groups of SIMD_WIDTH output feature maps. That is we lay out the different data structures as:

Activations and gradient of activations: feature_maps × feature-height × feature-width -> **feature_maps/SIMD_WIDTH** × feature-height × feature-width × SIMD_WIDTH

Weights and gradients of weights: input_feature_maps × output_feature_maps × kernel_height × kernel_width -> input_feature_maps × **output_feature_maps/SIMD_WIDTH** × kernel_height × kernel_width × SIMD_WIDTH

Transpose-weights: input_feature_maps × output_feature_maps × kernel_height × kernel_width -> output_feature_maps × **input_feature_maps/SIMD_WIDTH** × kernel_height × kernel_width × **SIMD_WIDTH**

Vectorization and Register Blocking

The three main loops for forward-propagation, backpropagation, and weight gradient update are written as follows (in Figure 3, 4, and 5). It is notable that we further block the loop over output feature map width (line 6 in Figure 3) into blocks of size RB_SIZE, and push the loop over RB_SIZE to the innermost loop (aside from the vector operation). This loop is to provide register blocking, wherein registers vout[...] store intermediate results for MACs involving all the weights. This way we can generate an instruction sequence in the innermost loop which consists of RB_SIZE vector FMA instructions, per vector load. While register blocking is performed within one row-vector of the output feature map, we can also perform two dimensional blocking, especially in cases where the size of the row is less than that of the number of registers which can be used.

```
1.  vector vwt, vout[RB_SIZE];
2.  for ofm = 0 … OFM/SIMD-1
3.      for ifm= 0… IFM/SIMD_WIDTH-1
4.          for ib = 0…SIMD_WIDTH-1
5.              for ofh = 0 … OFH-1
6.                  for ofw = 0 … OFW/RB_SIZE-1
7.                      for rb=0 … RB_SIZE-1
8.                          vout[rb] = SETZERO()
9.                      for kh = 0 … KH-1
10.                         for kw = 0 … KW-1
11.                           vwt = LOAD(weights[ifm][ofm][kh][kw][0])
12.                             for rb = 0 … RB_SIZE-1
13.                                 VFMADD(EXTLOAD(input[ib][STRIDE*ofh + kh][stride*ofw+kw][0]), vwt, vout[rb])
14.                     for rb=0 … RB-1
15.                         STORE(vout, output[ofm][ofmh][ofmw*RB_SIZE + rb][0])
```

Figure 3: Vectorized pseudo-code for forward propagation (bias additions are not included).

Similar to forward propagation, the backward propagation code is presented in Figure 4. This is similar to forward propagation in terms of vectorization and cache blocking, but differs in terms of how the register blocking is used. Here, the register block slides across the input row, while getting updated by corresponding transpose of weights and "gradient of outputs". In order to reduce overheads, we push in the loop over the "gradient of output" feature block as an inner loop (line 8, Figure 4). Moreover, the rim of the "gradient of input" feature map does not require the computation involving all the weights. For example row 0 of the input gradients will only be affected by weights[0][...]. We exploit this property by maintaining an array of kernel-start indices for each

row of the input gradients. We do not use a similar technique for columns as this leads to additional scalar operations in the innermost loop thereby hurting performance.

```
1.  vector vtranswt, vdelinp[RB_SIZE];
2.  for ifm = 0 … IFM/SIMD-1
3.      for ofm= 0… OFM/SIMD_WIDTH-1
4.          for ofh = 0 … OFH-1
5.              for ofw = 0 … OFW/RB_SIZE-1
6.                  for kw = 0 …  KW
7.                      for ob=0…SIMD_WIDTH-1
8.                          for rb=0 … RB_SIZE-1
9.                              vout[rb] = LOAD(delinput[ifm][ofh*][][0])
10.                         for kh = kh_start … kh_end
11.                             vtranswt = LOAD(trans_weights[ifm][ofm][kh][kw][0])
12.                             for rb = 0 … RB_SIZE-1
13.                                 VFMADD(EXTLOAD(deloutput[ofm*SIMD_WIDTH + ob][ofh][ofw*RB_SIZE+ rb][0]),
    vtranswt, vdelinp[rb])
14.                         for rb=0 … RB-1
15.                             STORE(delinp, delinput[ifm][ofmh][ofmw*RB_SIZE + rb][0])
```

Figure 4: Vectorized pseudo-code for backward propagation.

For weight gradient computation, we store the "weight gradients" in the vector register file. We have an alternate strategy where we store rows of the "weight gradients" in case the register file is not sufficiently large. For example, a 3x3 convolution will be stored in a register file, while a 5x5 convolution will be split into five updates over five weights. A third strategy stores weights for different output-feature maps. Here we only describe the first strategy in details.

The weight gradient computation, unlike forward or backward pass, requires a reduce operation, wherein the weight gradient contributions of each image are accumulated. For this we structure the loop to create an inner loop over SIMD_WIDTH input features, and SIMD_WIDTH output features (lines 5 and 6) over the minibatch size (line 4).

```
1.  vector vdelout, vdelwt[RB_SIZE];
2.      for ofm = 0 … OFM/SIMD-1
3.          for ifm= 0… IFM/SIMD_WIDTH-1
4.              for mb=0… MINIBATCH-1
5.                  for ib = 0…SIMD_WIDTH-1
6.                      for ob = 0 … SIMD_WIDTH-1
7.                          for kh=0 … KH
8.                              for kw=0…KW
9.                                  STORE(vdelwt[kh*KW+kw], delweights[ifm][ofm][kh][kw][0])
10.                                 for ofh = 0 … OFH-1
11.                                     for ofw = 0 … OFW/RB_SIZE-1
12.                                         for rb=0 … RB_SIZE-1
13.                                             vdelout = LOAD(deloutput[mb][ofm][ofh] [ofw*RB_SIZE+rb][0])
14.                                                 for kh = 0 … KH-1
15.                                                     for kw = 0 … KW-1
16.                                                         vdelwt[kh*KW+kw] = VFMADD(EXTLOAD(input[mb][ib]
    [STRIDE*ofh + kh][stride*(ofw*RB_SIZE+rb)+kw][0]), vdelout, vdelwt[kh*KH+kw])
17.                         for kh=0 … KH
18.                             for kw=0…KW
19.                                 STORE(vdelwt, delweights[ifm][ofm][kh][kw][0])
```

Figure 5: Weight gradient update operation for convolutional layers.

Threading and Work Partitioning

We perform fine grained partitioning of work across threads. For the forward and backpropagate operations, we partition the work across multiple minibatches into jobs, each for one row of the output/input across SIMD_WIDTH output/input features. These jobs are then equally distributed across the different threads.

For weight update, we treat weight kernels for SIMD_WIDTH input and output lane pairs to be the basic unit of work, and these jobs are subsequently distributed across multiple threads.

As you've seen from the analysis above, initial optimizations to Caffe achieve up to 10-11x improvement in performance, which is very exciting. This however is merely scratching the surface of what we believe we can achieve for this workload and we are excited to continually share our results as we progress. As mentioned, this result is largely achieved through the use Intel MKL, OpenMP, aligning data allocation as well as through the use of key new performance primitives. I encourage you to stay tuned as these new primitives are coming to future versions of Intel MKL and Intel Intel DAAL. I also invite you to try the technical preview package demonstrating achievable performance with Caffe on the AlexNet topology which is already available for download.

In the future, I'll share more details on scaling up multi-node training to achieve unprecedented time to model speeds for training neural networks.

A special thank you to Vadim Pirogov from Intel's Software and Services Group and Dipankar Das from Intel Lab's Parallel Computing Lab. They were the driving force behind the performance optimizations shown and the Caffe work illustrated in this blog post.