# Optimizing CNN Model Inference on CPUs

Yizhi Liu*, Yao Wang*, Ruofei Yu, Mu Li, Vin Sharma, Yida Wang
Amazon Web Services
East Palo Alto, CA, USA
{yizhiliu,wayao,yuruofei,mli,vinarm,wangyida}@amazon.com



## Abstract

The popularity of Convolutional Neural Network (CNN) models and the ubiquity of CPUs imply that better performance of CNN model inference on CPUs can deliver significant gain to a large number of users. The current approach to improving the performance of CNN model inference on CPUs relies on the use of a hardware-specific library of low-level operations such as Intel MKL-DNN and some basic model-level optimizations, which is restrictive and misses the opportunity to optimize the end-to-end inference pipeline as a whole. This paper proposes a more comprehensive approach of CNN model inference on CPUs that employs a full-stack and systematic scheme of operation-level and model-level optimizations coupled with efficient data layout transformations. Experiments show that our solution achieves up to 2.81× better latency for CNN model inference on a 18-core Intel Platinum 8000-series CPU compared to the state-of-the-art implementations using Intel MKL-DNN.

***Keywords*** high-performance computing, CNN model inference

## 1 Introduction

The growing use of Convolutional Neural Network (CNN) models in computer vision applications makes this model architecture a natural focus for performance optimization efforts. Similarly, the widespread deployment of CPUs in servers, clients, and edge devices makes this hardware platform an attractive target. Therefore, performing CNN model inference efficiently on CPUs is of critical interest to many users.

The performance of CNN model inference on CPUs leaves significant room for improvement. To optimize CNN model inference, deep learning frameworks rely on libraries provided by processor vendors (e.g. Intel MKL-DNN [23]). This raises several issues. First, incorporating a vendor-supplied library into a deep learning framework requires error-prone and time-consuming engineering effort. Second, although the kernel libraries are highly optimized, they present as third-party plug-ins, which may introduce compatibility issues with other libraries. For example, TensorFlow [4] originally used the Eigen library [3] to handle computation on CPUs. In contrast, Intel MKL-DNN uses OpenMP to implement multi-threading. OpenMP threads compete with Eigen threads, resulting in performance loss. Third, vendor-supplied libraries may not deliver the best performance in many cases. Ordinarily, these libraries tune very carefully for common operations with normal input data shapes but they often handle new operations poorly. Most importantly, the optimizations in the framework do not work in concert with the optimizations in the kernel library, which leaves significant performance gains unrealized in practice. In summary, this kind of *framework-specific* approach for CNN model inference on CPUs is cumbersome, inflexible, and sub-optimal.

On the other hand, optimizing the performance of CNN model inference in a *framework-agnostic* method is of obvious interest to many deep learning practitioners since it releases the constraint imposed by the framework. Recently, Intel launched a universal CNN model inference engine called OpenVINO Toolkit [14]. This toolkit optimizes CNN models in the computer vision domain on Intel processors (mostly CPUs) and achieves better performance than the deep learning frameworks alone. Yet, OpenVINO provides limited model-level optimization (e.g. operator fusion as implemented in ngraph [11]). Also, OpenVINO relies upon MKL-DNN to deliver performance gains for the carefully-tuned operations but is lack of coverage to all (e.g. some fused operations). Therefore, we argue that the optimization done by OpenVINO is still not sufficient for most of the CNN models.

In this paper, we propose a more comprehensive approach to optimize CNN models for efficient inference on CPUs. Our approach is full-stack and systematic, which includes operation-level as well as model-level optimizations. At the operator level, we focus on optimizing the most computationally-intensive operations like convolution (*CONV*) by arranging the data layout in accordance with memory access patterns on CPUs. We also implement multi-thread parallelization with minimal overhead. At the model level, we coordinate the individual operation optimizations by manipulating the data layout flowing through the entire model for the best end-to-end performance, in addition to the common techniques such as operator fusion and inference simplification. Ultimately, our approach produces a standalone executable binary with minimal size that delivers high performance for CNN inference on CPUs with no dependence on either the deep learning framework or the vendor-provided kernel library.

Our approach includes a number of enhancements that build upon TVM [7]. Specifically, this paper makes the following contributions to accelerate CNN model inference on CPUs:

- Manages the data layout of the computationally-intensive operations (e.g. *CONV*) to be memory-access friendly and makes good use of vectorized instructions (e.g., AVX-512);
- Designs schemes to search for the best data layout combination in different operations to maximize single-operation performance and minimize data layout transformation overhead;
- Implements a scalable thread pool to schedule the multi-thread parallelization with minimal overhead.

Although these individual techniques are not novel by themselves, it is worth noting that our approach is the first to combine them effectively and achieve unprecedented performance for deep learning model inference. In this paper we focus on optimizing CNN models on Intel CPUs but the techniques are easily generalizable to other model architectures and hardware targets.

On a 18-core Intel Platinum 8000-series (formerly code named Skylake) processor, the executable binaries generated by our approach outperform the best framework-specific solution (MXNet [6]) by $1.15 - 2.81\times$. Compared to the framework-agnostic inference engine (Intel OpenVINO toolkit), our approach delivers $0.98 - 1.75\times$ performance for all but four models. OpenVINO does not optimize those four models. In addition, our approach produces a standalone module that does not depend on either the framework or the vendor-provided library, which enables easy deployment to multiple targets. Our approach is in production use in several applications that employ deep learning for inference on several types of platforms. All source code has been released to the open source TVM project[1].

The rest of this paper is organized as follows: Section 2 reviews the background of modern CPUs as well as the typical CNN models; Section 3 elaborates the optimization ideas that we propose and how we implement them, followed by evaluations in Section 4. We list the related works in Section 5 and summarize the paper in Section 6.

## 2 Background

### 2.1 Modern CPUs

Although accelerators like GPUs and TPUs demonstrate their outstanding performance on the deep learning workloads, in practice, there is still a lot of deep learning computation, especially model inference, taking place on the general-purpose CPU due to its high availability. Currently, most of the CPUs equipped on PCs and servers are manufactured by Intel with x86 architecture [1].

[1]https://github.com/dmlc/tvm

Modern CPUs use thread-level parallelism via multi-core [16] to improve the overall processor performance given the diminishing increasing of transistor budgets to build larger and more complex uniprocessor. For example, the current mainstream generation of Intel CPUs (*Skylake*) typically incorporates from 2 (PCs and workstations) to 28 (high-end servers) physical cores. Different physical cores of the same processor share the LLC (last level cache) and access the main memory symmetrically. It is critical to avoid the interference among threads running on the same processor and minimize their synchronization cost in order to have a decent scalable performance on multi-core processors.

A single physical core achieves the peak performance via the SIMD (single-instruction-multiple-data) technique. SIMD loads multiple values into wide vector registers to process together. Each core of *Skylake* has two vector processing units (VPUs) which support the latest 512-bit Advanced Vector Extension instruction set (AVX-512). AVX-512 handles up to 16 32-bit single precision floating point numbers (totally 512 bits) per CPU cycle. In addition, it can utilize the Fused-Multiply-Add (FMA) instruction which executes one vectorized multiplication and then accumulates the results to another vector register in the same CPU cycle. Putting together, for the Skylake CPU we used in the experiments featured with 18 physical cores and two AVX-512 enabled VPUs running in 2.7 GHz frequency [2], its peak performance is $2.7 \times 18 \times 2 \times 2 \times 16 \simeq 3.1$ TFLOPS for single precision floating point numbers.

Our solution has taken all features above into consideration while doing the optimization. It is worth noting that *Skylake* normally supports hyper-threading [30] via the simultaneous multithreading (SMT) technique, in which the system could assign two virtual cores (i.e. two threads) to one physical core, aiming at improving the system throughput. However, the performance improvement of hyper-threading is application-dependent [29]. In our case, we do not use hyper-threading since one thread has fully utilized its physical core resource, adding one more thread to the same physical core will normally decrease the performance due to the additional context switch. We also restrict our optimization within a single processor using the shared-memory programming model. The Non-Uniformed Memory Access (NUMA) pattern occurred in the context of multiple processors on the same motherboard is beyond the scope of this paper. Although we ran the experiments on Intel Skylake processors, our solution is easy to be extended to other CPUs with both x86 and ARM architectures.

### 2.2 Convolutional neural networks

Convolutional neural networks (CNNs) are commonly used in computer vision workloads [19, 21, 22, 28, 34, 38, 39]. A CNN model is normally abstracted as a computation graph, essentially, Directed Acyclic Graph (DAG), in which a node represents an operator and a directed edge pointing from

node X to Y represents that the output of operator X serves as (part of) the input of operator Y (i.e. Y cannot be executed before X). Executing a model inference is actually to flow the input data through the graph to get the output. Doing the optimization on the graph (e.g. prune unnecessary nodes and edges, pre-compute values independent to input data) could potentially boost the model inference performance.

Most of the computation in the CNN model inference attributes to *convolutions (CONVs)* and *generalized matrix multiplications (GEMMs)*. These operations are essentially a series of multiplication and accumulation, which by design can fully utilize the parallelization, vectorization and FMA features of modern CPUs. The challenge is how to manage the input data of these operations to achieve it.

The rest of the CNN workloads are mostly memory-bound operations (e.g. batch normalization, pooling, activation, element-wise addition, etc.). According to the roofline model [44], they may become the performance bottleneck. Fortunately, it is possible to fuse them to the compute-bound operations like *CONVs* to eliminate a lot of unnecessary memory access [7]. This would largely increases the overall arithmetic intensity of the workload so as to boost the performance. The challenge is how to fuse the operations in a nice way so that the data layout optimized for *CONVs* can be kept throughput the entire inference.

# 3 Optimizations

This section describes our optimization ideas and implementations in details. The solution presented in this paper is end-to-end for doing the CNN model inference. Our solution is generic enough to work for a lot of popular models as we will show in the evaluation. Although we are targeting the Intel CPU specifically in this paper, our optimization is applicable to other hardware targets with minimal efforts. We summarize the optimization ideas we used as below:

1. Arranging the data layout for computation to reduce the cache misses, guarantee continuous memory access and facilitate vectorization (Section 3.1.1);
2. Making good use of vectorized registers (e.g. AVX-512 and AVX2 registers) for vectorization (Section 3.1.1);
3. Eliminating the contentions between different threads on computing resource, thread-shared data, etc. (Section 3.1.2);
4. Minimizing the overhead brought by other proposed optimization ideas (Section 3.2);
5. Leveraging auto search to look for best optimization schemes (Section 3.3).

We implemented the optimization based on the TVM stack [7] by adding a number of new features to the compiling pass, operator scheduling and runtime components. The original TVM stack has done a couple of general graph-level optimizations including operator fusion, pre-computing, simplifying inference for batch-norm and dropout [7], which are also inherited to our solution but will not be covered in this paper.

## 3.1 Operation optimization

Optimizing convolution operations is critical to the overall performance of a CNN workload. Previous work normally goes deep to the assembly code level to get decent performance. In this subsection, we show how to take advantage of the latest CPU features (AVX-512 instructions, vectorization, parallelization, etc.) to optimize a single *CONV* without going into the tedious assembly intrinsics.

### 3.1.1 Single thread optimization

We started from optimizing *CONV* within a thread. *CONV* is a computationally-intensive operation which traverses its operands multiple times for computation. Therefore, it is critical to manage the layout of the data fed to the *CONV* to reduce the memory access overhead for getting good performance. We first revisit the computation of *CONV* to facilitate the illustration of our memory management scheme. A typical *CONV* takes a 3D tensor and a number of 3D kernels to convolve to output another 3D tensor. Essentially, *CONV* consists of a sequence of multiplication and accumulation operations. The calculation is illustrated in Figure 1, which implies loops of 6 dimensions: *in_channel*, *kernel_height*, *kernel_width*, *out_channel*, *out_height* and *out_width*. Each kernel slides over the input tensor, does elementwise product and accumulates the values to produce the corresponding element in the output tensor. The number of kernels forms the *out_channel*. Note that three of the dimensions (*in_channel*, *kernel_height* and *kernel_width*) are reduction axes that cannot be embarrassingly parallelized.

We use the conventional notation *NCHW* to describe the default data layout, which means the input and output are 4-D tensors of (*batch size* N, *number of channels* C, *feature map height* H, *feature map width* W). The related layout of kernel is *KCRS*, in which *K*, *C*, *R*, *S* stand for the output channel, input channel, kernel height and kernel width, respectively.

*CONV* requires accumulation across the input channel, a natural way of doing so is to have this loop at the innermost so that no extra memory access takes place. It implies a more efficient input layout *NHWC*, which can leverage the L1/L2 cache locality of the CPU. This is a common practice that many implementations would follow, including frameworks [41] and chip-vendor provided libraries [23].

However, simple *NHWC* is not the most efficient data layout for CPUs. The number of input channels (can be as large as 512) is usually too large to fit all input channels of a single pixel into a cache line (typically only 32 or 64 bytes). Therefore, we define a layout *NCHW[x]c*, in which $c$ is a split sub-dimension of channel $C$ in super-dimension, and the number $x$ indicates the factor size of the sub-dimension. Empirically, $x$ is chosen from 32, 16 and 8, which can be integrally divided by the channel sizes (e.g. 64, 128, 256, 512)

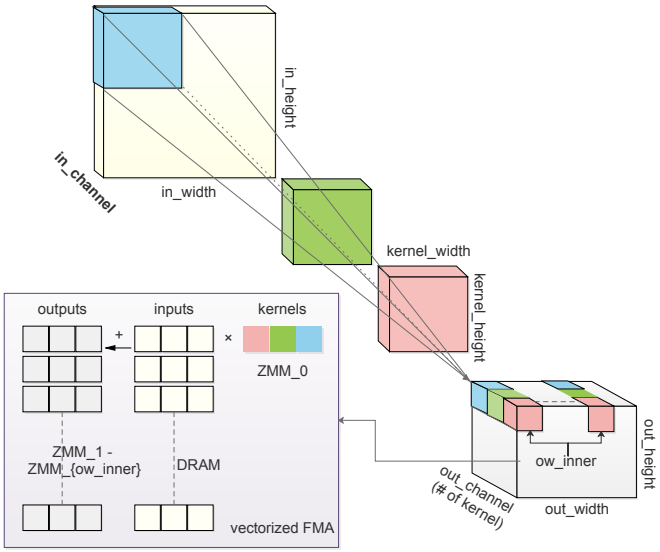



**Figure 1.** The illustration of *CONV* and its efficient implementation in AVX-512 instructions. In the figure there are three kernels depicted in blue, green and pink. To do efficient FMA, multiple kernel values are packed into one *ZMM* register and reused to multiply with different input values and accumulate to output values in different *ZMM* registers.

that most convolution neural network workloads use. Other sizes can be used for uncommon workloads.

The output has the same layout *NCHW[y]c* as the input, while the split factor can be different. Unlike the input channel, the output channel is no longer a reduction axis so can be vectorized. Therefore we split *out_channel* according to the width of the SIMD register, which is 512 bits (a.k.a 16 float32) for AVX-512 and 256 bits (a.k.a 8 float32) for AVX2, and reorder this loop to the innermost for vectorization of the FMA operations.

We also split *out_width* to *ow_outer* and *ow_inner* using a factor *ow_bn* and move the loop of *ow_inner* inside for register blocking. Intel AVX-512 introduces 32 512-bit width SIMD registers $ZMM_0 - ZMM_{31}$ [24]. In practice, we maintain the loop hierarchy to use one ZMM register to store the kernel data while others storing the feature map. The kernel values stored in one *ZMM* register are used to multiply with a number of input feature map values continuously stored in the DRAM via AVX-512F instructions [24], whose results are then accumulated to other *ZMM* registers storing the output values. The idea is illustrated in Figure 1.

Finally, in order to achieve continuous memory access of the convolution kernel, we re-arrange its layout to a 6-D tensor *KCRS[x]c[y]k*, in which $c$ with split size $x$ is the sub-dimension of input channel $C$, and $k$ with split size $y$ is the sub-dimension of output channel $K$. For example, with input data layout *NCHW32c* and output data layout *NCHW16c*, the related kernel layout will be *KCRS32c16k*.

We summarize the optimization of a single CONV algorithm in Algorithm 1. To sum up, we optimized the computation of *CONV* in single thread by 1) splitting the *in_channel* (one of the reduction axes) which accelerates the memory access (optimization idea §1); and 2) applying SIMD vectorization over the sub-dimension of *out_channel* and deliberately manipulating the loops to make better use of the SIMD registers, which accelerates the multiply-accumulate operation (optimization idea §2). All the optimizations can be implemented via the TVM stack scheduling schemes without writing a single line of assembly code (see supplementary material).

**Algorithm 1** Single CONV algorithm

```
1:  INPUT: IFMAP in NCHW[x]c
2:  INPUT: KERNEL in KCRS[x]c[y]k
3:  OUTPUT: OFMAP in NCHW[y]c
4:  for each disjoint chunk of OFMAP do
5:      for ow.outer:= 0 → out_width/ow_bn do
6:          Initialize ZMM_1 to ZMM_{ow_bn} by 0⃗
7:          for ic.outer:= 0 → in_channel/x do
8:              for each entry of KERNEL do
9:                  for ic.inner:= 0 → x do
10:                     vload(KERNEL, ZMM_0)
11:                     for i:= 1 → ow_bn + 1 do
12:                         vfmadd(IFMAP, ZMM_0, ZMM_i)
13:                     end for
14:                 end for
15:             end for
16:         end for
17:         for i:= 1 → ow_bn + 1 do
18:             vstore(ZMM_i, OFMAP)
19:         end for
20:     end for
21: end for
```

### 3.1.2 Thread-level parallelization

So far we have talked about optimizing the operations within a single thread. In a modern multi-core CPU, being able to utilize the thread-level parallelization well is also critical to achieve good performance. An ideal $N$-way parallelization should partition the work into independent $N$ pieces and have each of them finished at the same time, making the total execution time $1/N$ of the sequential case.

To achieve this goal, our approach parallelizes the work within an operation. Basically, in a system of $N$ physical cores, we evenly divide the outermost loop of the operation into $N$ pieces to assign to $N$ threads, then use a simple fork-join mechanism to coordinate the threads. Although conceptually straightforward, it is non-trivial to implement it efficiently. For example, introducing OpenMP pragmas before the for loop could easily parallelize it, but the resulting

scalability is not desirable according to our experiments (see Section 4) probably due to the parallelization overhead of multiple threads brought by the OpenMP implementation.

In order to better realize our optimization ideas §3, we implemented the multi-threading parallelization via a customized thread pool. The thread pool uses C++11 atomics to coordinate threads during fork/join and a lock-free queue to manage assigning tasks to the working threads. Active threads are guaranteed to run on disjoint physical cores via thread binding to minimize hardware contention, and no hyper-threading is used as discussed in Section 2. Different working threads thereby compute on different portions of the data using different physical cores. For the global data structure accessed by multiple threads such as the local-free queue, we inserted cache line padding as needed to avoid false sharing between threads.

### 3.2 Layout transform elimination

In this subsection, we extend the optimization scope from a single operation to the entire computation graph of the CNN model. The main idea here is to come up with a generic solution at the graph level to minimize the data layout transformation described in Section 3.1.

Since *NCHW[x]c* is efficient for *CONV*s which takes the majority of the CNN model computation, we should make sure that every *CONV* is executed in this layout. However, other operations between *CONV*s may not be compatible with *NCHW[x]c*, which makes each *CONV* transform the input data layout from default (*NCHW* or *NHWC*) to *NCHW[x]c* before the computation and transform it back at the end. This transformation introduces significant overhead.

Fortunately, from the perspective of the graph level, we can take the layout transformation out of *CONV* to be an independent node, and insert it only when necessary. That is, we eliminate the transformation taking place in the *CONV* operation and maintain the transformed layout flow through the graph as far as possible (optimization idea §4).

In order to determine if a data transformation is necessary, we first classify operations into three categories according to how they interact with the data layout as follows:

1. *Layout-mutable* operations. These operations process the data without the knowledge of its layout, i.e. it can handle data in any layout. Unary operations like *ReLU*, *Softmax*, etc., fall in this category.
2. *Layout-semimutable* operations. These operations need to know the data layout for processing, but can handle a number of layout options. For example, *CONV*, in our case, can deal with *NCHW*, *NHWC* and *NCHW[x]c* layouts. Other operations like *Batch_Norm*, *Pooling*, etc., fall in this category as well.
3. *Layout-immutable* operations. These operations process the data only in one specific layout, that is, they are not tolerate to any data transformation. Therefore, the layout has to be transformed back to the original one before passing to a layout-immutable operation. Transformation operations like *Flatten*, *Reshape*, etc, fall in this category.

Operations between *CONV*s in typical CNN models are either layout-mutable (e.g. *ReLU*, *SoftMax*, *Concat*, and *ElemwiseAdd*) or layout-semimutable (e.g. *Batch_Norm*, *Pooling*), making it possible to keep the data layout being *NCHW[x]c* across convolution layers. Layout transforming from *NCHW* to *NCHW[x]c* happens before the first CONV. Only if getting to a layout-immutable operation, e.g. *Flatten*, the data layout is transformed back from *NCHW[x]c* to *NCHW*.

In practice, we first traverse the computation graph to infer the data layout of each node as illustrated in the left side of Figure 2, then we alter the layout of *CONV*s from default to *NCHW[x]c* for better performance. Note that the detailed optimal layout (i.e. the value of $x$) may differ by different data shapes and types, we will explain more in Section 3.3. Finally, the layout-transform nodes are inserted to the graph accordingly. Thus, we still have *NCHW* input and output for the network, but the internal layouts between *CONV* layers are in optimized *NCHW[x]c*, as shown in the right part of Figure 2. It is worth noting that, the layout of the model parameters such as convolution kernel weights and the mean and variance of *Batch_Norm* are invariant so can be pre-transformed during the compilation. We also illustrate this in the right part of Figure 2.

We implemented the ideas by introducing multiple graph-level optimization *passes* to the TVM stack. By keeping transformed data layout invariant between *CONV* layers as much as possible and pre-transforming the layout of convolutional kernel weights at compilation time, we further improve the end-to-end performance of CNN model inference.

### 3.3 Optimization scheme search

We came up with the aforementioned optimization schemes, especially, how to layout the data, based on our understanding of the hardware, e.g. cache size, vectorization unit width, memory access pattern, etc. However, human knowledge is limited. Although we carefully applied the first 4 optimization ideas summarized in the beginning of this section, it is impossible to exhaust all possible optimal cases by hand. Also, Section 3.2 assumes that the factor of the channel, i.e. $x$ in *NCHW[x]c*, stays the same values during the entire network. Having various $x$ values in different *CONV*s may lead to a better performance.

Therefore, an automatic search for the best scheme (optimization idea §5) is needed to further improve the performance. Basically, we should build a system to allow the domain experts to construct the searching space for the machine to explore and come up with the best scheme resulting in the shortest execution time. The search is two-stage, first

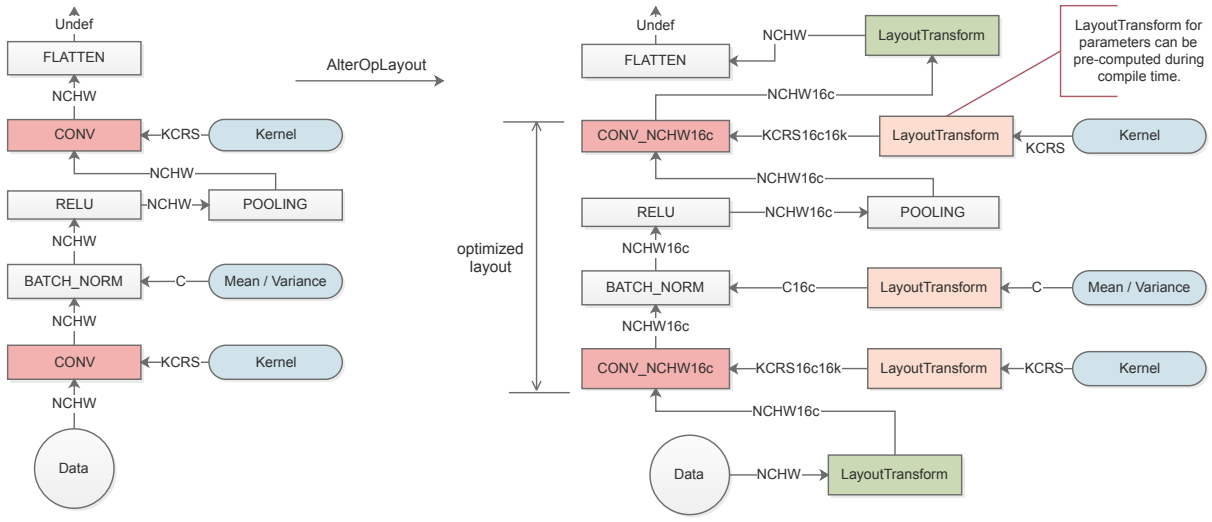



**Figure 2.** Layout optimization of a simple CNN model. The left side depicts the network with data layout inferred; the network is optimized by transforming and preserving the proper data layout on the right side.

local to find optimization scheme candidates for the individual computationally-intensive operations, then global to select and combine the individual schemes for the optimal end-to-end results.

### 3.3.1 Local search

The first searching step is to find the optimal schedules for each computationally-intensive operations, i.e. *CONV*s in a CNN model. We use a list *[ic_bn, oc_bn, reg_n, unroll_ker]* to represent a convolution schedule, where *ic_bn* and *oc_bn* stand for the splitting factor for input and output channels (i.e. $x$ in the *NCHW[x]c* notation), *reg_n* is the number of SIMD registers to be used at the inner loop, *unroll_ker* is a boolean deciding whether to unroll the convolution kernel for loop computation (line 8 of Algorithm 1). The local search uses a template to find the best combination of these values to minimize the *CONV* execution time, similar to the kernel optimization step in [25].

Specifically, the local search works as follows:

1. Define the candidate lists of *ic_bn* and *oc_bn*. To exhaust the searching space, we include all factors of the number of channels. For example, if the number of channels is 64, [32, 16, 8, 4, 2, 1] are listed as the candidates.
2. Define the candidate list of *reg_n*. In practice, we observed that utilizing all SIMD registers in a single thread does not always return the best performance. As a result, we choose *reg_n* from [32, 16, 8, 4, 2].
3. Define the candidate list of *unroll_ker* to be [*True*, *False*].
4. Walk through the defined schedule space to find the optimal combination leading to the shortest execution time. Each searching scheme will be run multiple times for averaging to cancel out the possible variance rooted from the unexpected interference from the operating system and/or other processes.

Local search works perfectly for each individual operation and indeed finds better optimization scheme than our manual work (see Table 1 in Section 4). However, greedily adopting the local optimal of every operation may not lead to the global optimal. Consider two consecutive *CONV* operations *conv_0* and *conv_1*, if the output splitting factor (*oc_bn*) of *conv_0* is different from the input splitting factor (*ic_bn*) of *conv_1*, an *LayoutTransform* node needs to be inserted to the graph as discussed in Section 3.2. This transformation overhead can be too expensive to take advantage of the benefit brought by the local optimal, especially when the data size of the network is large. On the other hand, if we maintain the same splitting factor throughout the entire network (as we did in Section 3.2, we may miss the opportunity to optimize some *CONV*s. Therefore, a trade-off should be made using a global search.

### 3.3.2 Global search

In this subsection, we extend the optimization search to the entire computation graph. The idea is to allow each *CONV* freely choosing the splitting factor $x$ (i.e. *ic_bn* and *oc_bn*), and take the corresponding data layout transform time into consideration. According to Section 3.2, the operations between *CONV*s are either *layout-mutable* or *layout-semimutable*, so they can use whatever $x$ decided by the *CONV* operation.

We extract a snippet of a typical CNN model in Figure 3 to illustrate the idea. From the figure we see that each *CONV* has a number of candidate schemes specified by different (*ic_bn* and *oc_bn*) pairs. The shortest execution time achieved

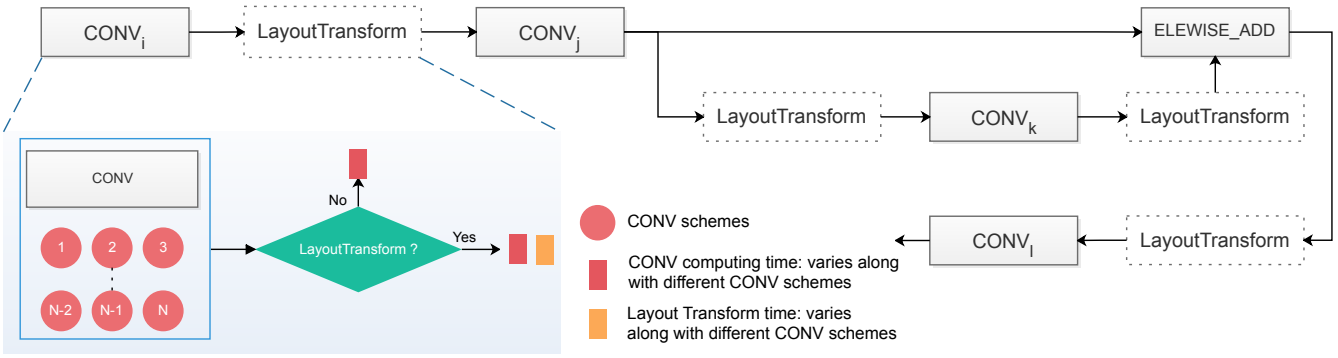



**Figure 3.** Global search for CNN model inference.

by each pair can be obtained in the local search step. The number of pairs depends on the numbers of input and output channels, which in practice is bound to 100. Choosing different schemes will introduce different data transformation overheads (denoted in dashed boxes between *CONV*s) or no transformation (if the *oc_bn* of the *CONV* equals the *ic_bn* of its successor). For simplicity, in the figure we omit the operations which do not impact the global search decision such as *ReLU*, *Batch_Norm* between two *CONV*s. However, operations like *Elementwise_Add* could not be omitted since it requires the layout of its two input operands (outputs of *CONV_j* and *CONV_k* in the figure) to be the same.

Naively speaking, if a CNN model consists of $n$ *CONV*s, each of which has $k_i$ candidate schemes, the total number of options of the global scheme will be $\prod_{i=1}^{n} k_i$, very easy to become intractable as the number of layers $n$ grows. Fortunately, in practice, we can use a dynamic programming algorithm to efficiently solve this problem. Note that when choosing the scheme for a *CONV*, we only need to consider the data layout of it and its direct predecessor(s) but not any other ancestor *CONV*s provided the so-far globally optimal schemes up to the predecessor(s) are memorized.

Therefore, a straightforward algorithm is constructed in Algorithm 2. This algorithm works well for CNN models with the simple structure like a list, in which each *CONV* only has one predecessor [28, 34, 37, 38]. In this case, after a *CONV* is done, the intermediate states stored for its predecessor can be safely removed. For networks with more complex structure like using *Elementwise_Add* to add two *CONV* outputs to feed to the next *CONV* [19], it is a little trickier since the schemes of a *CONV* may need to be saved for a future use (e.g. in Figure 3 $CONV_l$ needs the schemes of $CONV_j$ via *Elementwise_Add*).

In order to generalize to more complicated networks where a *CONV* may use arbitrary number of predecessors, we formalize the problem into *Markov Decision Process (MDP)* [5, 13] ans solve it using the Bellman optimality equation [27, 36]. It is doable because our problem is memoryless (making decision of a *CONV* only depends on its predecessors but not other ancestors). We define a candidate scheme of a *CONV* to be a *state*, the data layout transformation as the *action*, the inverse of the summation of current *CONV* execution time and the transformation time as the reward (since in MDP we are maximizing the reward). Note that in our problem setting if a *state* takes an *action*, it will transfer to another determined *state* with probability 1.

**Algorithm 2** Global search algorithm

```
Sort the nodes of the graph in topological order
Initialize the optimal schemes of the CONVs without dependency using the execution time of their candidate schemes
for CONV_i whose predecessors are solved do
    for each candidate scheme CSI_j of CONV_i do
        t = execution_time(CSI_j)
        for each predecessor X's so-far globally optimal scheme GSX_k do
            GSI_{j,k} = t + transform_time(j, k)
        end for
        Update the shortest GSI_j
    end for
    Mark CONV_i as solved
end for
return last node's shortest scheme
```

Let $s_{i,j}$ denote the jth candidate scheme in $CONV_i$, $v(s)$ denote the value function for the *state*, $transform\_time(s_{i_1,j_1}, s_{i_2,j_2})$ denote the layout transformation time between two states, and $execution\_time(s_{i,j})$ denote the execution time for the state, we can write the Bellman optimality equation for $v$:

$$v(s_{i,j_i}) = \max\{\sum_{i-n \le x \le i-1}(v(s_{x,j_x}) + transform\_time(s_{x,j_x}, s_{i,j_i}))\}$$

$$for\ j_x \in [0,\ number\ of\ states\ for\ CONV_x)$$

Next, we briefly introduce the algorithm to solve the Bellman optimality equation. Starting from all input nodes, we do a BFS traverse to the network. For the no-dependency

*CONV*s, we can directly get optimal value functions, since the optimal value is equal to the execution time given a candidate scheme. For other *CONV*s, we follow the Bellman optimality equation to get optimal value functions. Then we add the current *CONV* as a dependency to all of its children (i.e. the nodes it points to). If a *CONV* node has $m$ dependent nodes with $k_1, k_2, ..., k_m$ states, respectively, it will generate $\prod_{1 \le i \le m} k_i$ states. Finally, we release a dependency once its all predecessor is visited to reduce the total number of states. It is worth noting that if the network structure is too complicated, there are also algorithms to solve the Bellman optimality equation approximately to reduce the time and space [18, 35, 36], which we have not applied in this paper.

The time and memory complexities of this algorithm are equal to the number of states. In theory, all nodes in network can have a large number of input nodes, leading to the worst case where the total number of states being $O(n * k^n)$, in which $k$ is the number of candidate schemes for each stage and $n$ is the total number of stages. However, this situation rarely happens. In practice, nodes in a neural network only have a small number of dependencies. As a result, the number of states would be $O(n * k^m)$, where m is a small number.

# 4 Evaluation

This section evaluates the performance of our proposed solution by answering the following questions:

1. What is the overall performance of our end-to-end solution comparing with the start-of-the-art alternatives on CPUs?
2. What is the individual contribution of each optimization idea we proposed?
3. Is the classification accuracy of our solution comparable to the state-of-the-art?

All experiments were done on Amazon EC2 C5.9xlarge instances. Each instance is equipped with an entire Intel Skylake processor with 18 physical cores (36 virtual cores due to hyper-threading).

Our solution was built on top of the code base of the TVM stack 0.4.0. We chose one framework-specific solution and one framework-agnostic solution as baselines for comparison. For the framework-specific solution, we investigated a wide range of options and figured out that MXNet 1.2.1 with Intel MKL-DNN v0.15 enabled has the widest model coverage with best inference performance compared to others. The latest Intel OpenVINO Toolkit 2018 R2.0.2 served as the framework-agnostic solution. In addition, the LLVM OpenMP runtime library 4.0 was used in the comparison with our own thread pool for multi-thread scalability.

We ran the model inference on a number of popularly used CNN models, including ResNet [19], VGG [34], DenseNet [22], and Inception-v3 [39]. All models were pre-trained using ImageNet [12] dataset and available from the Gluon Model Zoo [2]. Our solution can take in models in other formats as well. The input data of the model inference are $224 \times 224$ ImageNet images, except for the Inception Net whose input size is $299 \times 299$ (This is to follow one of the original experimented sizes of receptive field for inception architecture and the default input size of inception-v3 implementation from popular deep learning frameworks). Since the most important performance criterion of model inference is the *latency*, We did all experiments with batch size 1, i.e. each time only one image was fed to the model, to measure the inference time.

## 4.1 Overall Performance

We first report the overall performance we got for 14 popular CNN models comparing with both MXNet and OpenVINO in Figure 4. From the figure we can tell our solution is substantially faster than MXNet on all models ($1.15 - 2.81\times$) as well as outperforms OpenVINO in most cases ($1.04 - 11.48\times$), except for ResNet-34 and Inception-v3 (2% slower).

As a framework-specific solution, MXNet was suboptimal for CNN inference on CPUs because it is lacking of flexibility to perform sufficient model level optimization (e.g. thorough operator fusion). Comparatively, the framework-agnostic solution provided by the OpenVINO gets rid of the framework limitation to get better performance than MXNet. While OpenVINO gets good performance on some networks, so far it does not handle all common CNN models well. For example, it performs terribly on the VGG model family probably due to the *batch_norm* operator in these models is not optimized. As OpenVINO is not open-sourced, we could not exactly understand its details. Our solution further outperforms OpenVINO mostly because of the advanced optimization techniques we presented in Section 3. In addition, both MXNet and OpenVINO largely rely on the third-party library Intel MKL-DNN to achieve good performance. Our solution, on the other hand, is independent from those chip vendor provided accelerating libraries, which gives us more room to optimize the model inference as a whole.

## 4.2 Performance Implications of Optimizations

This subsection breaks up the end-to-end performance gain of our solution by investigating the performance boost of each individual optimization technique we described in Section 3. For the sake of space, in each comparison we only pick one network from a network family, respectively. Other networks in the same family share the similar benefits.

### 4.2.1 Layout optimization of *CONV*

Firstly, we compare the performance with and without organizing the data in a memory access friendly format in the *CONV* operations at the second row of Table 1. We argued in Section 3.1.1 that using right data layout in *CONV*

[2] https://mxnet.incubator.apache.org/api/python/gluon/model_zoo.html

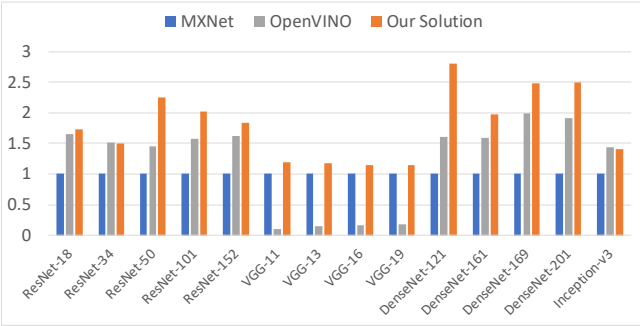



**Figure 4.** End-to-end performance comparison on 14 CNN models (higher is better). The performance of MXNet is normalized to 1.

| Speedup | ResNet-50 | VGG-19 | DenseNet-201 | Inception-v3 |
|---|---|---|---|---|
| Baseline | 1 | 1 | 1 | 1 |
| Layout Optimization | 5.34 | 8.33 | 4.08 | 7.41 |
| Transform Elimination | 8.22 | 9.33 | 5.51 | 9.11 |
| Global Search | 12.25 | 10.54 | 6.89 | 11.85 |

**Table 1.** The incremental individual speedup brought by our optimization compared to the NCHW baseline. The speedup of row *n* was achieved by applying the optimization techniques mentioned till this row.

is critical to get good performance due to better memory access and cache locality. From row 2 of Table 1 we see significant improvement compared to the default data layout, whose performance is normalized to baseline 1. Both implementations are with proper vectorization and thread-level parallelization. This part of optimization was done by most of the existing high-performance deep learning implementations, we replicate it using the TVM scheduling schemes without touching the assembly code or intrinsics.

#### 4.2.2 Layout transform elimination

Secondly, we evaluate the performance boost brought by eliminating the data layout transformation overhead as discussed in Section 3.2. The results were summarized at the third row of Table 1. Compared to the layout optimization of *CONV*, layout transform elimination further accelerates the execution time by $1.1 - 1.5\times$. Our solution uses a systematic way to eliminate the unnecessary data layout transform by inferring the data layout throughout the computation graph and inserting the layout transform nodes only if needed, which is not seen in other work.

#### 4.2.3 Optimization scheme search

Next, we compare the performance between the optimization schemes produced by our global searching algorithm and the ones carefully picked by us manually. As discussed in Section 3.3.2, our MDP algorithm is able to find the best combination of data layouts which outperforms the manually picked results by $1.1 - 1.5\times$, comparing the third and fourth row of Table 1. ResNet-50 (and its variants) gains more speedup from global search because the network structure is more complicated, hence leaving more optimization room. We do not observe much performance gain for VGG-19 (and its variants) by global search since the structure of this model is relatively simple. The results also verify that, with automatic global search, we can get rid of the tedious manual picking of data layout parameters by producing even better results.

| Top-1/Top-5 accuracy | ResNet-50 | VGG-19 | DenseNet-201 | Inception-v3 |
|---|---|---|---|---|
| Our solution | 0.765/0.931 | 0.743/0.919 | 0.773/0.936 | 0.777/0.937 |
| MXNet | 0.765/0.931 | 0.743/0.919 | 0.773/0.936 | 0.777/0.937 |
| OpenVINO | 0.765/0.931 | 0.743/0.919 | 0.773/0.936 | 0.777/0.937 |

**Table 2.** Top-1/top-5 accuracy comparison.

#### 4.2.4 Multi-thread parallelization

Lastly, we ran a strong scalability experiment using the multi-threading implementations backed by our own thread pool described at Section 3.1.2 and the commonly available/used OpenMP. We also included th result of MXNet and OpenVINO using Intel MKL-DNN (realizing multi-threading via OpenMP) for comparison. We configured OpenMP via environmental variables [3] to make sure that each thread runs on a disjoint core, which resembles the behavior of our thread pool. Figure 5 summarizes the number of images a model can inference one by one (i.e. $batchsize = 1$) in a second as a function of the number of threads the model uses. From the figure we see that using our thread pool achieves much better scalability than using OpenMP in our solution as well as in MXNet and OpenVINO. Furthermore, sometimes we observed that multi-threading enabled by OpenMP obtained worse performance while adding threads for unknown reasons. This makes us feel more comfortable to use a thread pool on which we have full control. Out of the four networks, VGG-19 achieves the best scalability (13.2× on 18 cores) because its *CONV* operations are larger. On the other hand, OpenVINO does not optimize the VGG family well and gets almost no speedup by adding threads.

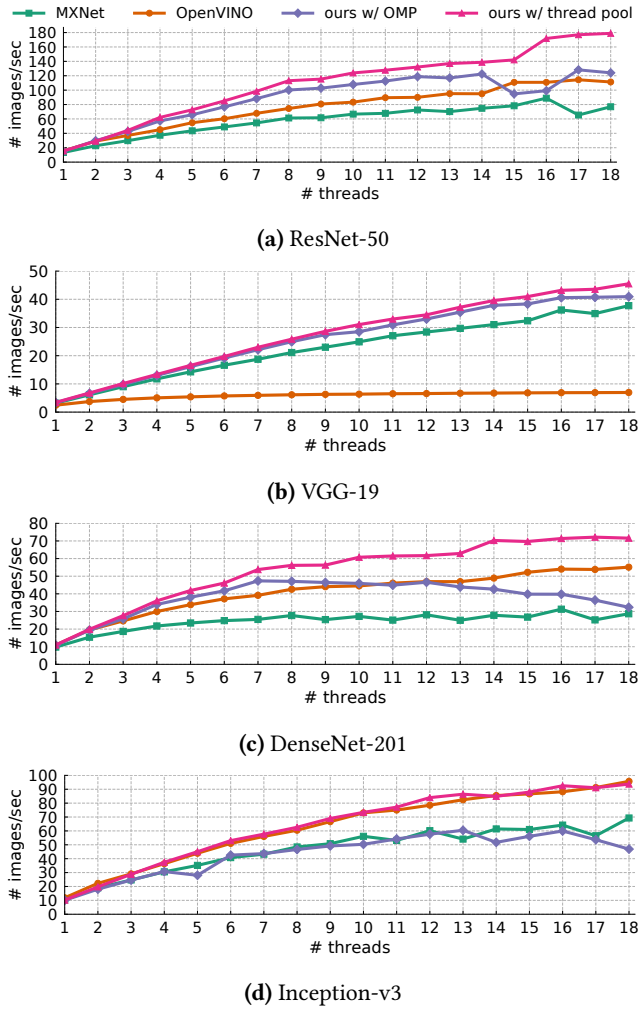
### 4.3 Accuracy

In this subsection, as a sanity check, we validate the correctness of the models compiled by our solution. For each model, we randomly sampled 1000 images from ImageNet and used our solution, MXNet and OpenVINO to run the model inference for prediction. Table 2 reports the top-1 and top-5 accuracy numbers we got out of three methods on different CNN models, from which we can see that our solution does not lose any accuracy while speeding up the process.

## 5 Related Works

As deep learning demonstrates more and more power in the real-world applications, there is a significant amount of effort being made to accelerate the deep learning workloads

[3] KMP_AFFINITY=granularity=fine,compact,1,0




**(a)** ResNet-50

**(b)** VGG-19

**(c)** DenseNet-201

**(d)** Inception-v3

**Figure 5.** Scalability comparison between different multi-threading implementations on various neural networks.

on all kinds of hardware ranging from CPUs [20, 23, 40, 46], GPUs [8, 10], FPGAs [15, 17, 45], to special-purpose accelerators [9, 26]. Modern deep learning frameworks normally leverage these optimized implementations to run deep learning training and inference on the corresponding hardware targets. There are also works tailored for inference to address the inference-specific requirement such as low latency and small binary size on different hardware targets (e.g. GPUs [31], ASICs [17]). OpenVINO Toolkit [14] developed by Intel employs similar optimization pipeline as we did to optimize the CNN models for Intel products. Our solution outperforms OpenVINO on Intel CPUs due to more advanced optimization on both the model-level and operation-level.

Our solution is based on the TVM stack [7], an end-to-end framework inspired by Halide [32], which expresses a deep learning model into intermediate representations (IRs) and lowers the IRs eventually to the machine code. We adopted the pipeline of TVM, and extended it to Intel CPUs via a series of optimizations reported in the paper. There are several other similar IR-style frameworks such as Tensor Comprehensions [42], Glow [33] and DLVM [43]. However, so far none of them has reported decent Intel CPU inference results as we did. We believe our solution is capable to be adapted by these frameworks.

The idea of fully utilizing registers for vectorized FMA in Section 3.1.1 is mainly inspired by Intel MKL-DNN [23]. While MKL-DNN uses x86 assembly to arrange the memory access and computation, our solution stays at the Domain Specific Language (DSL) level. Moreover, with DSL-level templates, we have the chance to surpass human knowledge via local and global searching described in Section 3.3.

The optimized layout is used by several deep learning frameworks and libraries to accelerate the computation. For example, TensorFlow [4] uses *NHWC*, Intel MKL-DNN [23] uses *NCHW16c* and *NCHW8c* for AVX-512 and AVX2 respectively. Unlike the solutions above, our solution supports arbitrary dimension tiling size and inserts layout transform nodes into the computation graph. This general layout support makes it possible to optimize uncommon deep learning workloads and extremely reduces the layout transforming overhead during runtime.

The concept of Markov Decision Process (MDP) dates back to 1950s [5, 13]. Much of the later work focused on modeling and solving MDP problem with direct or approximated algorithms. Modeling global searching for neural network performance as MDP and formatting it with the Bellman optimality equation [27, 36] are inspired by the fact that neural network holds the Markov Property. The work in this paper leverages the previous idea and applies it to a new application.

## 6 Conclusion

In this paper, we proposed an end-to-end solution to compile and optimize convolutional neural networks to do the model inference efficiently on modern CPUs. The experiments show that we are able to achieve up to 2.81× speedup on 14 popular CNN models on the up-to-date Intel processors (Intel Xeon Platinum 8000-series, formerly code named Skylake) compared to the performance of the state-of-the-art framework-specific (MXNet) and framework-agnostic (OpenVINO toolkit) solutions. Our solution is applicable to get decent performance on other CPUs (e.g. ARM CPUs) with minimal effort. The future work includes executing the optimized computation graph in parallel [40], doing optimization for model inference with quantized values (e.g. INT8) and extending our optimization ideas (e.g. data layout manipulation and global optimal data layout search) to the optimization work on other hardware targets. Supporting the model inference of recurrent neural networks is another interesting direction to explore.

## References


[1] [n. d.]. AMD vs Intel Market Share. https://www.cpubenchmark.net/market$_s$hare.html. ([n. d.]). [Online; accessed 13-Jul-2018].

[2] [n. d.]. Intel Xeon Processor Scalable Family Specification Update. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf. ([n. d.]). [Online; accessed 18-Jul-2018].

[3] 2017. Eigen: a C++ Linear Algebra Library. http://eigen.tuxfamily.org/. (2017). [Online; accessed 03-May-2018].

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[5] R. Bellman. 1957. A Markovian Decision Process. *Journal of Mathematics and Mechanics*.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *arXiv preprint arXiv:1802.04799* (2018).

[8] Xie Chen, Yongqiang Wang, Xunying Liu, Mark JF Gales, and Philip C Woodland. 2014. Efficient GPU-based training of recurrent neural network language models using spliced sentence bunch. In *Fifteenth Annual Conference of the International Speech Communication Association*.

[9] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM* 59, 11 (2016), 105–112.

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759* (2014).

[11] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Krovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *arXiv preprint arXiv:1801.08058* (2018).

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 248–255.

[13] C. Derman. 1957. In *Finite state Markovian decision processes*. Academic Press.

[14] Deanne Deuermeyer and Andrey Z. [n. d.]. OpenVINO Toolkit Release Notes. https://software.intel.com/en-us/articles/OpenVINO-RelNotes. ([n. d.]). [Online; accessed 13-Jul-2018].

[15] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. 2009. CNP: An FPGA-based Processor for Convolutional Networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 32–37.

[16] Lance Hammond, Benedict A Hubbert, Michael Siu, Manohar K Prabhu, Michael Chen, and K Olukolun. 2000. The stanford hydra cmp. *IEEE micro* 20, 2 (2000), 71–84.

[17] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA.. In *FPGA*. 75–84.

[18] W. K. Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika. 57 (1): 97âĂŞ109.* (1970).

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[20] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.

[21] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[22] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks.. In *CVPR*, Vol. 1. 3.

[23] Intel. 2018. Intel Math Kernel Library for Deep Neural Networks (Intel MKL-DNN). https://github.com/intel/mkl-dnn. (2018). [Online; accessed 16-Apr-2018].

[24] James R. (Intel). 2013. Intel AVX-512 Instructions. https://software.intel.com/en-us/blogs/2013/avx-512-instructions. (2013). [Online; accessed 25-Apr-2018].

[25] Ziheng Jiang, Tianqi Chen, and Mu Li. 2018. Efficient Deep Learning Inference on Edge Devices. (2018).

[26] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, 1–12.

[27] Donald E. Kirk. 1970. In *Optimal Control Theory: An Introduction*.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[29] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. 2002. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution* 45 (2002).

[30] Debbie Marr, Frank Binns, D Hill, Glenn Hinton, D Koufaty, et al. 2002. Hyper-threading technology in the netburst® microarchitecture. *14th Hot Chips* (2002).

[31] NVIDIA. 2018. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. (2018). [Online; accessed 16-Apr-2018].

[32] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM*

*SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 519–530.

[33] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *arXiv preprint arXiv:1805.00907* (2018).

[34] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[35] Richard Sutton. 1970. Learning to predict by the methods of temporal differences. *Machine Learning. 3 (1): 9-44.* (1970).

[36] R. S. Sutton and A. G. Barto. 2017. In *Reinforcement Learning: An Introduction*. The MIT Press, 37–57.

[37] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning.. In *AAAI*, Vol. 4. 12.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[39] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2818–2826.

[40] Linpeng Tang, Yida Wang, Theodore Willke, and Kai Li. 2018. Scheduling Computation Graphs of Deep Learning Models on Manycore CPUs. *ArXiv e-prints* (July 2018). arXiv:cs.DC/1807.09667

[41] Tensorflow. 2018. Tensorflow Performance Guide. https://www.tensorflow.org/performance/performance$_{g}$uide#data$_{f}$ormats. (2018). [Online; accessed 3-Aug-2018].

[42] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[43] Richard Wei, Lane Schwartz, and Vikram Adve. 2017. DLVM: A modern compiler framework for neural network DSLs. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*.

[44] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[45] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Metworks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[46] Aleksandar Zlateski, Kisuk Lee, and H Sebastian Seung. 2016. ZNN–A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 801–811.

# Optimizing CNN Model Inference on CPUs

## Supplementary material

Here is how to define and schedule a convolution (CONV) operation in the way we described in Section 3.1.1. `declaration_conv_NCHWc` defines the computation of CONV in NCHW[x]c layout with necessary padding and stride setting. `schedule_conv_NCHWc` uses TVM scheduling primitives to schedule the computation as the procedure described in Algorithm 1. We used a high-level language (Python) instead of intrinsics and/or assembly codes that a high-performance implementation typically does. This allows us to try out different ideas easily, and facilitates the layout transform elimination and best optimization scheme search we introduced in Section 3.2 and 3.3.

```
In [ ]: def declaration_conv_NCHWc(wkl, sch, data, kernel):
            """Define convolution computation in NCHW[x]c layout

            Parameters
            ----------
            wkl : namedtuple
                The CONV workload configuration

            sch : namedtuple(ic_bn, oc_bn, reg_n, unroll_ker)
                The schedule configuration

            data : tvm.Tensor
                Input feature map in NCHW[x]c layout

            kernel : tvm.Tensor
                Kernel in KCRS[x]c[y]k layout
            """
            batch_size = data.shape[0]
            # Calculate the output height and width
            out_height = (wkl.height + 2 * wkl.hpad - wkl.hkernel) // wkl.hstride + 1
            out_width = (wkl.width + 2 * wkl.wpad - wkl.wkernel) // wkl.wstride + 1

            # Pad NCHW[x]c data
            data_pad = pad(data, (0, 0, wkl.hpad, wkl.wpad, 0), name="data_pad")

            # Define the output shape
            oshape = (batch_size, wkl.out_filter//sch.oc_bn,
                      out_height, out_width, sch.oc_bn)

            # Define the reduce axes
            ic = tvm.reduce_axis((0, wkl.in_filter), name="ic")
            kh = tvm.reduce_axis((0, wkl.hkernel), name="kh")
            kw = tvm.reduce_axis((0, wkl.wkernel), name="kw")

            # Define the computation in NCHW[x]c layout
            conv = tvm.compute(oshape, lambda n, oc_chunk, oh, ow, oc_block:
                               tvm.sum(data_pad[n, ic//wkl.ic_bn,
```

```
                                    oh*wkl.hstride+kh,
                                    ow*wkl.wstride+kw,
                                    ic%wkl.ic_bn].astype(wkl.out_dtype) *
                                kernel[oc_chunk, ic//wkl.ic_bn,
                                       kh, kw, ic%wkl.ic_bn, oc_block],
                                axis=[ic, kh, kw]), name="conv2d_NCHWc")
    return conv


def schedule_conv_NCHWc(sch, conv):
    """Schedule the convolution in NCHW[x]c layout

    Parameters
    ----------
    sch : namedtuple(ic_bn, oc_bn, reg_n, unroll_ker)
        The schedule configuration

    conv : tvm.tensor.ComputeOp
        Compute node generated by declaration_conv_NCHWc
    """
    s = tvm.create_schedule(conv.op)
    # Create a cache stage to help codegen further promote to register.
    CC = s.cache_write(conv, 'global')

    _, oc_chunk, oh, ow, oc_block = s[conv].op.axis
    ow_chunk, ow_block = s[conv].split(ow, factor=sch.reg_n)
    s[C].reorder(oc_chunk, oh, ow_chunk, ow_block, oc_block)
    # parallel over oc_chunk and oh
    parallel_axis = s[C].fuse(oc_chunk, oh)
    s[C].parallel(parallel_axis)

    # the inner loop where CC computes,
    # i.e., ow_block was split according to the number of registers
    s[CC].compute_at(s[C], ow_chunk)
    _, oc_chunk, oh, ow, oc_block = s[CC].op.axis
    ic, kh, kw = s[CC].op.reduce_axis

    # split to keep same as that in s[conv]
    ow_chunk, ow_block = s[CC].split(ow, factor=sch.reg_n)
    ic_chunk, ic_block = s[CC].split(ic, factor=sch.ic_bn)

    s[CC].reorder(oc_chunk, oh, ow_chunk, ic_chunk, kh, kw,
                  ic_block, ow_block, oc_block)
    if sch.unroll_ker:
        s[CC].unroll(kw)
        s[CC].unroll(kh)

    # Explictily use vectorization instruction
    s[CC].vectorize(oc_block)
    # Unroll ow_block < # of registers,
    # so that it can be optimized to use registers.
    s[CC].unroll(ow_block)

    return s
```