Optimizing CNN Model Inference on CPUs

Yizhi Liu[*]*, Yao Wang*[*]*, Ruofei Yu, Mu Li, Vin Sharma, Yida Wang*
Amazon Web Services
{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. To improve the performance of CNN inference on CPUs, current approaches like MXNet and Intel OpenVINO usually treat the model as a graph and use the high-performance libraries such as Intel MKL-DNN to implement the operations of the graph. While achieving reasonable performance on individual operations from the off-the-shelf libraries, this solution makes it inflexible to conduct optimizations at the graph level, as the local operation-level optimizations are predefined. Therefore, it is restrictive and misses the opportunity to optimize the end-to-end inference pipeline as a whole. This paper presents *NeoCPU*, a comprehensive approach of CNN model inference on CPUs that employs a full-stack and systematic scheme of optimizations. *NeoCPU* optimizes the operations as templates without relying on third-parties libraries, which enables further improvement of the performance via operation- and graph-level joint optimization. Experiments show that *NeoCPU* achieves up to 3.45× lower latency for CNN model inference than the current state-of-the-art implementations on various kinds of popular CPUs.

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. Performing a CNN model inference is essentially executing a computation graph consisting of operations. In practice, people normally use high-performance kernel libraries (e.g. Intel MKL-DNN [27] and OpenBlas [51]) to obtain high performance for CNN operations. While these libraries tune very carefully for common operations with normal input data shapes (e.g. 2D convolutions), they only focus on the (mostly, convolution) operations but miss the opportunities to further optimize the end-to-end model inference at the graph level. The graph-level optimization is often handled by the deep learning frameworks, e.g. TensorFlow [5] and MXNet [8].

However, the graph-level optimization such as operation fusion and data layout planing that a framework can do is limited because the operation implementation is already predefined in the third-party libraries. Therefore, the optimizations in the frameworks do not work in concert with the optimizations in the kernel library, which leaves significant performance gains unrealized in practice. Furthermore, different CPU architectures rely on different high-performance libraries and integrating a library into a deep learning framework requires error-prone and time-consuming engineering effort. Lastly, although those libraries are highly optimized, they present as third-party plug-ins, which may introduce contention issues with other libraries in the framework. As an example, TensorFlow originally used the Eigen library [4] to handle computation on CPUs. Later on, MKL-DNN was also introduced. As a consequence, at runtime MKL-DNN threads coexist with Eigen threads, resulting in resource contention. In summary, this kind of *framework-specific* approach for CNN model inference on CPUs is inflexible, cumbersome, and sub-optimal.

Because of the constraint imposed by the framework, optimizing the performance of CNN model inference end-to-end without involving a framework (i.e. a *framework-agnostic* method) is of obvious interest to many deep learning practitioners. Recently, Intel launched a universal CNN model inference engine called OpenVINO Toolkit [16]. This toolkit optimizes CNN models in the computer vision domain on Intel processors (mostly x86 CPUs) and claims to achieve better

[*]Equal contribution

performance than the deep learning frameworks alone. Yet, OpenVINO could only provide limited graph-level optimization (e.g. operation fusion as implemented in ngraph [15]) as it still relies upon MKL-DNN to deliver performance gains for the carefully-tuned operations. Therefore, the optimization done by OpenVINO is still not sufficient for most of the CNN models.

Based on the previous observation, we argue that in order to further improve the CNN model inference performance on CPUs, being able to do the *flexible end-to-end optimization* is the key. In this paper, we propose *NeoCPU*, a comprehensive approach to optimize CNN models for efficient inference on CPUs. *NeoCPU* is full-stack and systematic, which includes operation- and graph-level joint optimizations and does not rely on any third-party high-performance libraries. At the operation level, we follow the well-studied techniques to optimize the most computationally-intensive operations like convolution (*CONV*) in a *template*, which is applicable to different workloads on multiple CPU architectures and enables us for flexible graph-level optimization. At the graph level, in addition to the common techniques such as operation fusion and inference simplification, we coordinate the individual operation optimizations by manipulating the data layout flowing through the entire model for the best end-to-end performance. In summary, *NeoCPU* does the end-to-end optimization in a flexible and automatic fashion, while the existing works rely on third-party libraries and lack comprehensive performance tuning.

NeoCPU is built upon a deep learning compiler stack named TVM [9] with a number of enhancements. TVM enables the possibility of using own operation-level optimizations instead of third-party high-performance libraries, which make it flexible to apply our operation- and graph-level joint optimization. However, there was only one customized operation-level optimization on ARM CPUs for convolutions with specific data shapes and no operation- and graph-level joint optimization in the original TVM stack before our work. In addition, there exist other deep learning compilers such as Tensor Comprehensions [46] and Glow [40]. Unfortunately, they either do not target on CPUs or not optimize the CPU performance well, e.g. based on the paper description and our own experiments, Glow only optimizes the single-core performance for CPUs. Therefore we do not incorporate those works as the baseline. Table 1 summarizes the features of *NeoCPU* compared to others. To the best of our knowledge, *NeoCPU* achieves competitive performance for CNN model inference on various kinds of popular CPUs.

Specifically, this paper makes the following contributions:

- Provides an operation- and graph-level joint optimization scheme to obtain high CNN model inference performance on different popular CPUs including Intel, AMD and ARM, which outperforms the current state-of-the-art implementations;
- Constructs a template to achieve good performance of convolutions, which is flexible to apply to various convolution workloads on multiple CPU architectures (x86 and ARM) without relying on high-performance kernel libraries;
- Designs a global scheme to look for the best layout combination in different operations of a CNN model, which minimizes the data layout transformation overhead between operations while maintaining the high performance of individual operations.

	Op-level opt	Graph-level opt	Joint opt	Open-source
NeoCPU	✓	✓	✓	✓
MXNet [8]/TensorFlow [5]	3rd party	limited	✗	✓
OpenVINO [16]	3rd party	limited	?	✗
Original TVM [9]	incomplete	✓	✗	✓
Glow [40]	single core	✓	✗	✓

Table 1: Side-by-side comparison between *NeoCPU* and existing works on CNN model inference

It is worth noting that, this paper primarily deals with direct convolution computation, while *NeoCPU* is compatible to other optimziation works on the computationally-intensive kernels, e.g. *CONV*s via Winograd [7, 29] or FFT [52].

We evaluated *NeoCPU* on CPUs with both x86 and ARM architectures. In general, *NeoCPU* delivers the best performance for 13 out of 15 popular networks on Intel Skylake CPUs, 14 out of 15 on AMD EYPC CPUs, and all 15 models on ARM Cortex A72 CPUs. It is worthwhile noting that the baselines on x86 CPUs were more carefully tuned by the chip vendor (Intel MKL-DNN) but the ARM CPUs were less optimized. While the selected framework-specific (MXNet and TensorFlow) and framework-agnostic (OpenVINO) solutions may perform well on one case and less favorably on the other case, *NeoCPU* runs efficiently across models on different architectures.

In addition, *NeoCPU* produces a standalone module with minimal size that does not depend on either the frameworks or the high-performance kernel libraries, which enables easy deployment to multiple platforms. *NeoCPU* is used in Amazon SageMaker Neo Service [1], enabling model developers to optimize for inference on CPU-based servers in the cloud and devices at the edge. Using this service, a number of application developers have deployed CNN models optimized for inference in production on several types of platforms. All source code has been released to the open source TVM project[2].

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.

[1] https://aws.amazon.com/sagemaker/neo/
[2] https://github.com/dmlc/tvm

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 significant number of deep learning computation, especially model inference, taking place on the general-purpose CPUs due to the high availability. Currently, most of the CPUs equipped on PCs and servers are manufactured by Intel or AMD with x86 architecture [1], while ARM CPUs with ARM architecture occupy the majority of embedded/mobile device market [2].

Modern CPUs use thread-level parallelism via multi-core [21] to improve the overall processor performance given the diminishing increasing of transistor budgets to build larger and more complex uniprocessor. It is critical to avoid the interference among threads running on the same processor and minimize their synchronization cost in order to have good scalability on multi-core processors. Within the processor, 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. For example, Intel introduced the 512-bit Advanced Vector Extension instruction set (AVX-512), which handles up to 16 32-bit single precision floating point numbers (totally 512 bits) per CPU cycle. And the less advanced AVX2 processes data in 256-bit registers. In addition, these instruction sets utilize the Fused-Multiply-Add (FMA) technique which executes one vectorized multiplication and then accumulates the results to another vector register in the same CPU cycle. The similar SIMD technique is embodied in ARM CPUs as NEON [3]. As shown in the experiments, our proposed solution works on both x86 and ARM architectures.

In addition, it is worth noting that modern server-side CPUs normally supports hyper-threading [37] 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 [35]. In our case, we do not use hyper-threading since one thread has fully utilized its physical core resource and 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 processors using the shared-memory programming model as this is the typical system setting for CNN model inference. 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.

2.2 Convolutional neural networks

Convolutional neural networks (CNNs) are commonly used in computer vision workloads [23, 26, 33, 36, 41–43]. A CNN model is normally abstracted as a computation graph, essentially, Directed Acyclic Graph (DAG), in which a node represents an operation and a directed edge pointing from node X to Y represents that the output of operation X serves as (a part of) the inputs of operation 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)*. 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. Existing works [19, 24, 27] have demonstrated that it is possible to achieve high performance of convolution operations on CPUs by arranging the data layout and consequently, the computation, in an architecture-friendly way. The remaining challenge is how to manage the data layout flowing through these operations efficiently to get the high performance out of the end-to-end CNN model inference.

The rest of the CNN workloads are mostly memory-bound operations associated to *CONV*s (e.g. batch normalization, pooling, activation, element-wise addition, etc.). The common practice [9] is fusing them to *CONV*s so as to increase the overall arithmetic intensity of the workload and consequently boost the performance.

The computation graph of CNN model training has no essential difference with inference, just being larger (adding in backwards operations) and with some more computationally-trivial operations (e.g. loss function). Therefore, the optimization work done for CNN model inference is applicable to training as well.

3 Optimizations

This section describes our optimization ideas and implementations in detail. The solution presented in this paper is end-to-end for doing the CNN model inference. Our proposed solution is generic enough to work for a wide range of common CNN models as we will show in the evaluation. The basic idea of our approach is to view the optimization as an end-to-end problem and search for a globally best optimization. That is, we are not biased towards a local performance optimal of a single operation as many previous works. In order to achieve this, we first present how we optimized the computationally intensive convolution operations at low-level using a configurable template (Section 3.1). This makes it flexible to search for the best implementation of a specific convolution workload on a particular CPU architecture, and to optimize the entire computation graph by choosing proper data layouts between operations to eliminate unnecessary data layout transformation overhead (presented in Section 3.2

and 3.3).

We implemented the optimization based on the TVM stack [9] by adding a number of new features to the compiling pass, operation scheduling and runtime components. The original TVM stack has done a couple of generic graph-level optimizations including operation fusion, pre-computing, simplifying inference for batch-norm and dropout [9], which are also inherited to this work 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 as it takes the majority of computation. This is a well-studied problem but the previous works normally go deep to the assembly code level for high performance [24, 27]. In this subsection, we show how to take advantage of the latest CPU features (SIMD, FMA, parallelization, etc.) to optimize a single *CONV* without going into the tedious assembly code or C++ intrinsics. By managing the implementation in high-level instead, it is then easy to extend our optimization from a single operation to the entire computation graph.

3.1.1 Single thread optimization

We started from optimizing *CONV* within one thread. *CONV* is computationally-intensive 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. We first revisit the computation of *CONV* to illustrate our memory management scheme. A 2D *CONV* in CNN takes a 3D feature map (height $\times$ width $\times$ channels) and a number of 3D convolution kernels (normally smaller height and width but the same number of channels) to convolve to output another 3D tensor. 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 feature map along the height and width dimensions, does element-wise product and accumulates the values to produce the corresponding element in the output feature map, which can naturally leverage FMA. The number of kernels forms *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 with *batch size* N, *number of channels* C, *feature map height* H, *feature map width* W, where *N* is the outermost and *W* is the innermost dimension of the data. 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.

Following the common practice [27, 45], we organized the

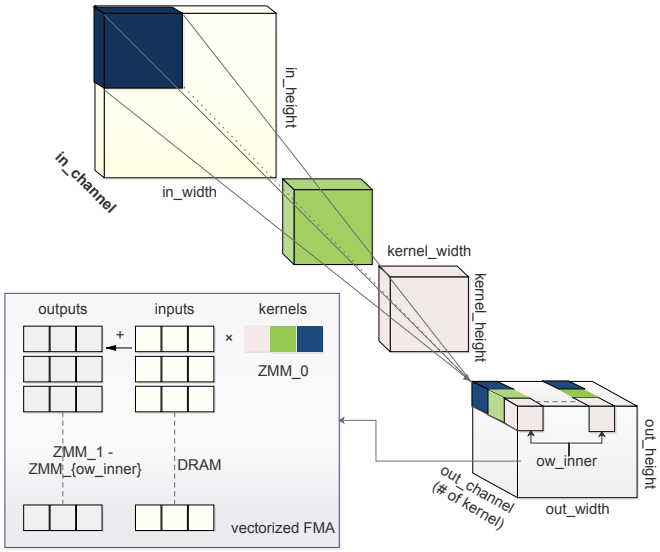


Figure 1: The illustration of *CONV* and the efficient implementation in AVX-512 instructions as an example. There are three kernels depicted in dark blue, green and light pink. To do efficient FMA, multiple kernel values are packed into one *ZMM* register and reused to multiply with different input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

feature map layout as *NCHW[x]c* for better memory access patterns i.e. better cache locality, in which c is a split sub-dimension of channel C in super-dimension, and the number x indicates the split size of the sub-dimension (i.e. $\#channels = sizeof(C) \times sizeof(c)$, where $sizeof(c) = x$). The output has the same layout *NCHW[y]c* as the input, while the split factor can be different. Correspondingly, the convolution kernel is organized in *KCRS[x]c[y]k*, in which c with split size x and k with split size y are the sub-dimensions of input channel C and output channel K, respectively. It is worth noting that a significant amount of data transformation overhead needs to be paid to get the desired layout.

In addition to the dimension reordering, for better utilizing the latest vectorization instructions (e.g. AVX-512, AVX2, NEON, etc.), we split *out_width* to *ow_outer* and *ow_inner* using a factor *reg_n* and move the loop of *ow_inner* inside for register blocking. For example, on a CPU featured AVX-512, we can utilize its 32 512-bit width registers $ZMM_0 - ZMM_{31}$ [28] as follows. 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 (up to 512 bits, a.k.a, 16 output channels in float32) are used to multiply with a number of input feature map values continuously stored in the DRAM via AVX-512F instructions [28], whose results are then accumulated to other *ZMM* registers storing the output values. Figure 1 illustrates this idea. For other vectorized instructions, the same idea applies but the split factor of *out_width* (i.e. *reg_n*) may change.

~~Algorithm 1 summarizes our optimization of *CONV* in single thread, which essentially is about 1) dimension ordering for friendly memory locality and 2) register blocking for good vectorization instruction utilization, as in previous works. However, unlike others, we made it a *template* in high-level language , in which the block size (x, y), the number of utilized registers (reg_n), and the loop-unroll strategy ($unroll\_ker$) are easily configurable. Consequently, the computing logic can be adjusted according to different CPU architectures (cache size, registered vector width, etc.) as well as different workloads (feature map size, convolution kernel size, etc.). This is flexible and enables graph-level optimization we will discuss later.~~

Algorithm 1 CONV operation algorithm via FMA

PARAM: $x > 0$ s.t. $in\_channel \bmod x = 0$
PARAM: $y > 0$ s.t. $out\_channel \bmod y = 0$
PARAM: $reg\_n > 0$ s.t. $out\_width \bmod reg\_n = 0$
PARAM: $unroll\_ker \in \{True, False\}$
INPUT: $IFMAP$ in NCHW[x]c
INPUT: $KERNEL$ in KCRS[x]c[y]k
OUTPUT: $OFMAP$ in NCHW[y]c
for each disjoint chunk of $OFMAP$ **do** ▷ parallel
  for ow.outer:= $0 \rightarrow out\_width/reg\_n$ **do**
    Initialize V_REG_1 to $V\_REG_{reg\_n}$ by $\vec{0}$
    for ic.outer:= $0 \rightarrow in\_channel/x$ **do**
      for each entry of $KERNEL$ **do** ▷ (opt) unroll
        for ic.inner:= $0 \rightarrow x$ **do**
          $vload(KERNEL, V\_REG_0)$ ▷ y floats
          for i:= $1 \rightarrow reg\_n + 1$ **do** ▷ unroll
            $vfmadd(IFMAP, V\_REG_0, V\_REG_i)$
          end for
        end for
      end for
    end for
    for i:= $1 \rightarrow reg\_n + 1$ **do**
      $vstore(V\_REG_i, OFMAP)$
    end for
  end for
end for

3.1.2 Thread-level parallelization

It is a common practice to partition *CONV* into disjoint pieces to parallelize among multiple cores of a modern CPU. Kernel libraries like Intel MKL-DNN usually uses off-the-shelf multi-threading solution such as OpenMP. However, we observe that the resulting scalability of the off-the-shelf parallelization solution is not desirable (Section 4.2.4).

Therefore, we implemented a customized thread pool to efficiently process this kind of embarrassing parallelization. Basically, in a system of N physical cores, we evenly divided the outermost loop of the operation into N pieces to assign to N threads. Then we used C++11 atomics to coordinate threads during fork-join and an single-producer-single-consumer lock-free queue between the scheduler and every working thread to assign tasks. Active threads are guaranteed to run on disjoint physical cores via thread binding to minimize the hardware contention, and no hyper-threading is used as discussed in Section 2.1. For the global data structure accessed by multiple threads such as the lock-free queues, we inserted cache line padding as needed to avoid false sharing between threads. In summary, this customized thread pool employs deliberate mechanism to prevent resource contention and reduce the thread launching overhead, which makes it outperform OpenMP according to our evaluation.

3.2 Layout transformation 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 introduced by the optimization in Section 3.1. Previous works [19, 24, 27] which focus on individual operation optimization normally do not consider about the data layout transformation overhead between highly optimized operations.

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 only be compatible with the default layout, 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.

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-oblivious* 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-tolerant* 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-dependent* operations. These operations process the data only in one specific layout, that is, they do not

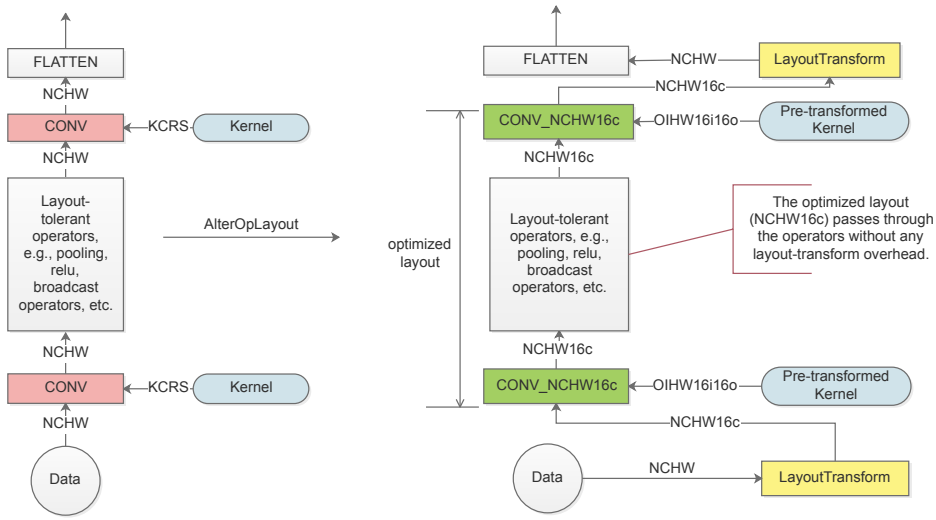


Figure 2: Layout optimization of a simple CNN model. The notation on an edge represents the layout of the data passing through this edge. The left side depicts the network with default data layout. Each *CONV* node in pink needs to pay additional overhead to transform the data into a favorable layout to achieve good performance and then transform back to default. The network in the right side is optimized at the graph level to minimize the data layout transformation during the runtime. The *CONV* nodes in green do not need to transform any data before and after computation.

tolerate any data transformation. Therefore, the layout has to be transformed to a certain format before passing to a layout-dependent operation. Transformation operations like *Flatten*, *Reshape*, etc, fall in this category.

Operations between *CONV*s in typical CNN models are either layout-oblivious (e.g. *ReLU*, *SoftMax*, *Concat*, and *ElemwiseAdd*) or layout-tolerant (e.g. *Batch_Norm*, *Pooling*), making it possible to keep the data layout being *NCHW[x]c* across convolution layers. Layout transformation from *NCHW* to *NCHW[x]c* happens before the first *CONV*. Data layout between *CONV*s can be maintained the same (i.e. *NCHW[x]c* sharing the same x value) without transformation. Only if getting to a layout-dependent 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 in order to prevent further transformation, we make x a constant number (e.g. 16) across all *CONV*s. However, this value may vary across different *CONV*s in order to get the optimal performance, which requires layout transformation. We will explain more about this in Section 3.3. Finally, the *LayoutTransform* 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 convolution 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, it is tedious and impractical to exhaust all possible optimal cases by hand. As a trade-off, Section 3.2 assumes that the split factor of the channel, i.e. x in *NCHW[x]c*, stays the same during the entire network, while having various x values in different *CONV*s may lead to a better performance. In addition, the split factor of the output width, i.e. *reg_n*, also needs to adjust for different vectorization instruction sets.

Therefore, an automatic search for the best scheme is in demand to further improve the performance. Basically, we should build a system to allow the domain experts to construct the search space for the machine to explore for the best scheme resulting in the shortest execution time. The search is two-stage, first 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. It is feasible to conduct this

kind of search given the optimization template described in Section 3.1.

3.3.1 Local search

The first step is to find the optimal schedules for each computationally-intensive operations, i.e. *CONV*s in a CNN model. We used a tuple *(ic_bn, oc_bn, reg_n, unroll_ker)* to represent a convolution schedule, whose items are chosen to cover different CPU architectures and generations for different convolution workloads. The first two terms *ic_bn* and *oc_bn* stand for the split factors of input and output channels (i.e. x in the *NCHW[x]c* notation), which are relevant to the cache sizes of a specific CPU. The third term *reg_n* is the number of SIMD registers to be used at the inner loop, which varies among different CPU architectures and generations. Also, we observed that utilizing all SIMD registers in a single thread does not always return the best performance. The last term *unroll_ker* is a boolean deciding whether to unroll the *for* loop involving convolution kernel computation (line 12 of Algorithm 1), as in some scenarios unrolling this loop may increase the performance by reducing branch penalties and such. The local search uses the template discussed in 3.1.1 to find the best combination of these values to minimize the *CONV* execution time, similar to the kernel optimization step in [31].

Specifically, the local search works as follows:

1. Define the candidate lists of *ic_bn* and *oc_bn*. To exhaust the possible cases, 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 choose the *reg_n* value from [32, 16, 8, 4, 2].
3. Define the candidate list of *unroll_ker* to be [*True*, *False*].
4. Walk through the defined space to measure the execution time of all combinations, each of which 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. This eventually generates a list of combinations ascendingly ordered by their execution time.

It is worth noting that we designed the above tuple in a configurable way, which means that we can always revise the tuple (e.g. adding or removing items, modifying the candidate values of an item) as needed.

Empirically, the local search of a CNN model takes a few hours using one machine, which is acceptable as it is one-time work. For example, it took about 6 hours to search for the 20 different *CONV* workloads of ResNet-50 on an 18-core Intel Skylake processor. In addition, we can maintain a database to store the results for every convolution workload (defined by the feature map and convolution kernel sizes) on every CPU type to prevent repeating search for the same convolution in different models.

Local search works well for each individual operation and indeed finds better optimization scheme than our manual work. 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 split factor (*oc_bn*) of *conv_0* is different from the input split factor (*ic_bn*) of *conv_1*, a *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 split 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 split factor x (i.e. *ic_bn* and *oc_bn*), and take the corresponding data layout transformation time into consideration. According to Section 3.2, the operations between *CONV*s are either *layout-oblivious* or *layout-tolerant*, 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 by each pair can be obtained in the local search step. The number of pairs is bound to 100 since both *ic_bn* and *oc_bn* usually have choices less than 10. 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 *Element-wise_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 (DP) 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 as long as the so-far globally optimal schemes up to the predecessor(s) are memorized.

Therefore, a straightforward algorithm is constructed in

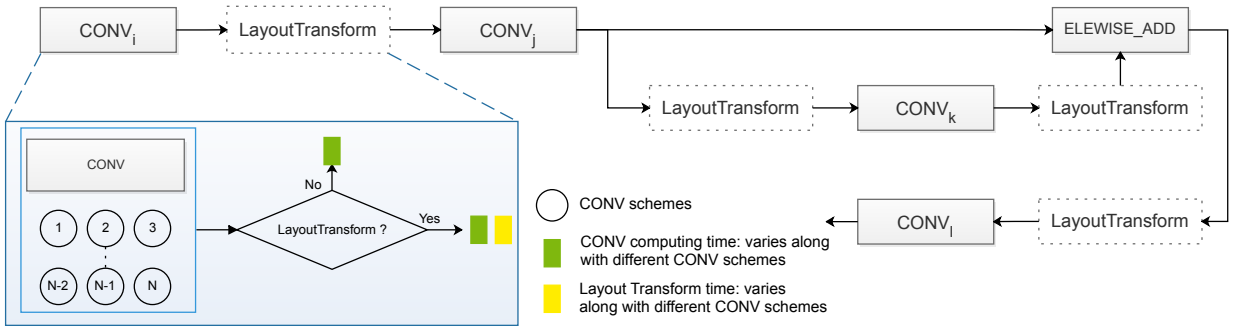


Figure 3: Global search for CNN model inference. *LayoutTransform* may or may not be in invoked according to the global decision. If invoked, an additional overhead of data transformation denoted in yellow needs to be paid.

Algorithm 2. In practice, a lot of CNN models has the structure as simple as a list, in which each *CONV* only has one predecessor [33, 41]. 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* [23], it is 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*).

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 in topological order do
    for each candidate scheme CSI_j of CONV_i do ▷ j is the j^th scheme of CONV_i
        t = execution_time(CSI_j)
        GSI_j = MAX ▷ initialize global optimal scheme of CONV_i under scheme j
        for each so-far globally optimal scheme GSX_k of predecessor x do ▷ k is the k^th scheme of CONV_x
            cur_opt = t + transform_time(k, j) + GSX_k
            if cur_opt < GSI_j then
                GSI_j = cur_opt
            end if
        end for
    end for
end for
return last node's shortest scheme
```

However, if the model structure becomes too complicated with many data dependency links between *CONV*s, the straightforward DP algorithm could go intractable, too. For example, in the object detection model SSD [36], the number of states can reach the order of trillions due to the occurrence of many concatenation blocks. In this case, we introduced an approximate solution to accelerate the search. Particularly, we reduced our global search problem to the register allocation problem in the canonical compiler domain with minor modification as follows. The register allocation problem is modeled as graph representation in which each node (variable) has a candidate list containing all possible register options, and each edge is associated with a cost matrix indicating the availability of registers between two nodes [20]. Similarly in our global search, each *CONV* has a list of candidate schemes and each edge is associated with the layout transformation cost matrix generated by the scheme lists of two *CONV*s. For other non-*CONV* nodes like *Elementwise_Add* which require all inputs in the same layout, we fixed the layout of one input and convert all other input layouts to it. Therefore, we defined the candidate list of a non-*CONV* node to be the same as the first input *CONV* and the cost matrix on the edge between these two nodes as all diagonal elements being 0 and all the other elements being infinite. For the edges between this non-*CONV* node and other input nodes, cost matrices are generated from the first input node and other input nodes. After such modification, all nodes and edges in our graph have the valid properties which are required by the register allocation modeling. This enables us to apply a heuristic solver based on partitioned boolean quadratic programming (PBQP) to our problem as it is done in register allocation [20].

In order to verify the result of this approximation algorithm, we compared it with the result of DP (the guaranteed best) on some simple networks where DP is tractable. It turns out that the approximation algorithm gets at least 88% of the best available result. Empirically, a typical DP search completes in 1 minute for most CNN models. In practice, we switch to the approximation algorithm if DP does not complete in 5 minutes. The approximation algorithm completes quickly, e.g. in 10 seconds. For the 15 popular networks we evaluated in Section 4, only SSD was done approximately.

4 Evaluation

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

1. What is the overall performance of *NeoCPU* comparing

with the start-of-the-art alternatives on various kinds of CPUs?

2. What is the individual contribution of each optimization idea we proposed?

All experiments were done on Amazon EC2 instances. We evaluated *NeoCPU* on three kinds of CPUs, Intel Skylake (C5.9xlarge, 18 physical cores, featured with AVX-512), AMD EPYC (M5a.12xlarge, 24 physical cores, featured with AVX2) and ARM Cortex A72 (A1.4xlarge, 16 physical cores, featured with NEON). Although testing on the cloud, our results of ARM CPUs apply to the ones at the edge devices such as Raspberry Pi and Amazon Echo Dot due to the same architecture. All cores have uniformed memory access.

NeoCPU was built on top of the code base of the TVM stack 0.4.0. For CPUs with x86 architecture, we chose two framework-specific solutions 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 1.3.1 with Intel MKL-DNN v0.15 enabled has the widest model coverage with the best inference performance compared to others (e.g. Intel Caffe). In addition, we chose TensorFlow 1.12.0 with ngraph v0.12.0-rc0 integration (empirically proved to be better than TensorFlow XLA on CPUs) due to its popularity. TensorFlow is known to have better performance on CPUs than another popular deep learning framework PyTorch [14]. The latest Intel OpenVINO Toolkit 2018 R5.445 served as the framework-agnostic solution. We used the official image-classification sample [3] and object-detection-ssd sample [4] for benchmarking. For ARM CPUs, we chose MXNet 1.3.1 with OpenBlas 0.2.18 and TensorFlow 1.12.0 with Eigen fd68453 [5] as the baselines. No framework-agnostic comparison was performed as on ARM CPUs there is no counterpart of OpenVINO to x86 CPUs. In addition, OpenMP 4.5 implemented in GCC 7.3 was used in the comparison with our own thread pool for multi-thread scalability. As a note, all implementations used direct convolution. Incorporating the advanced convolution algorithms to further improve the performance remains for future work.

We ran the model inference on a number of popular CNN models, including ResNet [23], VGG [41], DenseNet [26], Inception-v3 [43], and SSD [36] using ResNet-50 as the base network. Models consumed by MXNet and OpenVINO were from the Gluon Model Zoo [6]. Models consumed by TensorFlow were obtained mostly from TF-SLim [7] and for some missing ones (e.g. ResNet-34, DenseNet-169) we manually created them. The same model in different formats are semantically identical. As inherited from the TVM stack, *NeoCPU* is compatible to both Gluon and TF-slim formats, and in the evaluation we used the former one. The input data of the model inference are 224×224 images, except for the Inception Net (299×299) and SSD (512×512) by following the popular convention. 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. Therefore, we fix the value N in $NCHW[x]c$ as 1. *NeoCPU* works for larger batch sizes as well, in which cases we just need to add the N value to our configuration tuple.

Since our optimization does not change the semantics of the model, we do not expect any change of the model output. As a sanity check, we compared the results generated by *NeoCPU* with other baselines (prediction accuracy for image classification models and mean accuracy prediction for object detection models) to validate the correctness.

4.1 Overall Performance

We first report the overall performance we got for 15 popular CNN models comparing with the baselines on different CPUs in Table 2. The results were obtained by averaging the execution times of 1000 samples, doing inference for one at a time. In general, *NeoCPU* is more efficient across different models on different CPU architectures than any of the baselines (up to 11× speedup without considering the suspicious OpenVINO outliers which will be explained later). Compared to the *best available* baseline result for each model, *NeoCPU* gets 0.94-1.15× performance on the Intel Skylake CPU, 0.92-1.72× performance on the AMD EYPC CPU, and 2.05-3.45× performance on the ARM Cortex A72 CPU.

As framework-specific solutions, MXNet and TensorFlow were suboptimal for CNN inference on CPUs because it is lacking of flexibility to perform sufficient graph level optimization (e.g. flexible data layout management). MXNet has active MKL-DNN support from Intel so it performed quite well on CPUs with the x86 architecture. MXNet performed worse than TensorFlow on ARM due to the scalability issue (demonstrated in Figure 4c). TensorFlow performs significantly worse on SSD as it introduces branches to this model, which requires dynamic decisions to be made during the runtime. Comparatively, the framework-agnostic solution provided by the OpenVINO tries to further boost the performance by removing the framework limitation. However, the performance of OpenVINO was unstable across models. Although it gets appealing results on some cases, OpenVINO sometimes performed extremely slowly on certain models (e.g. 45× slower than us for ResNet-152 on AMD) for unknown reasons. When summarizing the speedup results, we do not include these outliers. It is also worth noting that the

[3] https://docs.openvinotoolkit.org/latest/_inference_engine_samples_classification_sample_README.html

[4] https://docs.openvinotoolkit.org/latest/_inference_engine_samples_object_detection_sample_ssd_README.html

[5] https://github.com/tensorflow/tensorflow/blob/r1.12/tensorflow/workspace.bzl#L128

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

[7] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim

Unit: ms	ResNet-18	ResNet-34	ResNet-50	ResNet-101	ResNet-152	VGG-11	VGG-13	VGG-16
MXNet	2.77, .01	**4.85, .02**	6.60, .00	12.90, .04	18.58, .07	12.05, .00	15.16, .00	18.55, .00
TensorFlow	4.07, .00	6.95, .00	11.93, .01	20.36, .00	37.33, .02	18.78, .01	24.28, .00	27.64, .02
OpenVINO	3.54, .00	5.43, .00	7.95, .00	12.55, .00	17.32, .01	138.07, .12	137.51, .14	140.95, .33
NeoCPU	**2.64, .00**	5.14, .00	**5.73, .00**	**11.15, .01**	**17.24, .01**	**11.91, .00**	**14.91, .00**	**18.21, .00**
	VGG-19	DenseNet-121	DenseNet-161	DenseNet-169	DenseNet-201	Inception-v3	SSD-ResNet-50	
MXNet	21.83, .00	14.72, .00	31.07, .01	19.73, .00	26.66, .00	**10.43, .00**	42.71, .00	
TensorFlow	35.94, .00	18.65, .01	32.97, .00	23.03, .01	29.19, .01	16.39, .04	358.98, .13	
OpenVINO	147.41, .12	9.03, .00	18.55, .01	11.80, .01	14.92, .01	10.65, .00	30.25*, .01	
NeoCPU	**21.77, .00**	**8.04, .01**	**17.45, .04**	**11.21, .01**	**13.97, .03**	10.67, .01	**31.48, .00**	

(a) Overall performance on a system with 18-core Intel Skylake CPU

Unit: ms	ResNet-18	ResNet-34	ResNet-50	ResNet-101	ResNet-152	VGG-11	VGG-13	VGG-16
MXNet	7.84, .36	14.66, .14	22.48, .48	40.57, 2.54	58.92, 3.21	49.17, 1.75	59.19, 1.35	72.57, 2.74
TensorFlow	13.95, .24	25.02, .49	38.14, .35	74.41, .56	108.38, .24	60.30, .22	71.16, .33	96.33, .22
OpenVINO	8.56, 1.02	15.18, .60	21.95, .42	1711.42, 1.59	2515.08, 2.51	662.09, 1.73	709.58, 1.78	828.17, 2.09
NeoCPU	**7.15, .49**	**14.10, .68**	**18.79, 1.01**	**39.32, .87**	**55.71, .54**	**28.58, .74**	**38.17, .29**	**57.63, .68**
	VGG-19	DenseNet-121	DenseNet-161	DenseNet-169	DenseNet-201	Inception-v3	SSD-ResNet-50	
MXNet	84.76, 1.91	35.00, 1.06	79.58, .63	47.82, 1.67	63.67, .15	30.12, .09	132.73, 2.59	
TensorFlow	121.04, .38	45.87, .15	98.39, .93	57.49, .28	77.37, .24	48.78, .45	747.78, 2.24	
OpenVINO	1113.17, 2.39	**22.36, .24**	818.86, 1.39	438.72, 1.27	453.12, 1.75	**25.75, .83**	93.65*, .81	
NeoCPU	**63.78, .18**	24.30, .54	**49.37, .09**	**31.70, .47**	**46.12, .51**	26.37, .32	**97.26, .54**	

(b) Overall performance on a system with 24-core AMD EYPC CPU

Unit: ms	ResNet-18	ResNet-34	ResNet-50	ResNet-101	ResNet-152	VGG-11	VGG-13	VGG-16
MXNet	75.82, 1.31	135.24, 2.49	149.65, 2.37	252.76, 3.25	351.60, 3.49	385.50, 2.39	505.06, 3.28	575.80, 2.98
TensorFlow	50.50, .07	96.50, .11	107.50, .12	223.83, .17	336.56, .19	245.97, .18	336.05, .27	381.46, .21
NeoCPU	**19.26, .08**	**37.20, .14**	**45.73, .02**	**86.77, .08**	**126.65, .13**	**87.66, .21**	**124.75, .05**	**162.49, .14**
	VGG-19	DenseNet-121	DenseNet-161	DenseNet-169	DenseNet-201	Inception-v3	SSD-ResNet-50	
MXNet	642.27, 4.30	211.54, 3.22	389.33, 2.98	264.36, 3.82	315.10, 3.49	275.28, 3.27	657.22, 3.29	
TensorFlow	459.91, .27	122.48, .07	301.51, .11	159.39, .08	204.79, .10	142.00, .07	1020.16, .47	
NeoCPU	**201.03, .49**	**44.00, .09**	**87.36, .15**	**58.93, .65**	**65.48, .54**	**84.00, .08**	**318.48, .11**	

(c) Overall performance on a system with 16-core ARM Cortex A72 CPU

Table 2: Overall performance of *NeoCPU* and the selected baselines. Each entry contains the mean value of 1000 runs and the corresponding standard error. The best performance of each model is in **bold**. (*OpenVINO on Intel and AMD CPUs does not measure the entire SSD execution time)

OpenVINO measures the execution time of SSD without taking into account a significant amount of operations including *multibox detection*. Since OpenVINO is not open-sourced, we were not able to modify it for apples-to-apples comparison on the SSD model. OpenVINO does not work for ARM CPUs as it relies on MKL-DNN which optimizes only for CPUs with x86 architecture. *NeoCPU* outperforms the baselines mostly because of the advanced optimization techniques we presented in Section 3. In addition, all baselines largely rely on the third-party libraries (MKL-DNN, OpenBlas, Eigen) to achieve good performance. *NeoCPU*, on the other hand, is independent from those high-performance libraries, which gives us more room to optimize the model inference as a whole.

4.2 Optimization Implications

This subsection breaks up the end-to-end performance gain of *NeoCPU* 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. We only report the performance results on Intel CPUs in Section 4.2.1-4.2.3. The optimization effect applies to AMD and ARM CPUs, too. Basically, Section 4.2.1 is the operation-level optimization, and Section 4.2.2 and 4.2.3 cover the operation- and graph-level joint optimization.

4.2.1 Layout optimization of *CONV*

Firstly, we compare the performance with and without organizing the data in a memory access and vectorized instruction utilization friendly layout (*NCHW{x}c*) for the *CONV* operations at the second row of Table 3. This is the operation-level optimization that is commonly applied by the compared baselines in Section 4.1. We replicate it as a template using TVM scheduling schemes without touching the assembly code or

Speedup	ResNet-50	VGG-19	DenseNet-201	Inception-v3	SSD-ResNet-50
Baseline	1	1	1	1	1
Layout Opt.	5.34	8.33	4.08	7.41	6.34
Transform Elim.	8.22	9.33	5.51	9.11	9.32
Global Search	12.25	10.54	6.89	11.85	12.49

Table 3: The individual speedup brought by our optimization compared to the *NCHW* baseline. The speedup of row *n* was achieved by applying the optimization techniques till this row.

intrinsics, which enables the subsequent optimization for various CNN models on different CPU architectures. From row 2 of Table 3 we see significant improvement compared to the default data layout (*NCHW*), whose performance is normalized to baseline 1. Both implementations are with proper vectorization and thread-level parallelization, as well as basic graph-level optimizations introduced by the original TVM stack, e.g. operation fusion, pre-computing, inference simplification, etc.

4.2.2 Layout transformation elimination

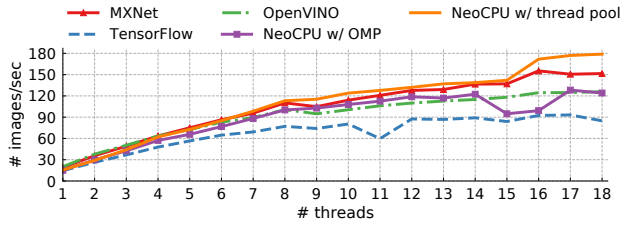
Secondly, we evaluate the performance boost brought by eliminating the data layout transformation overhead as discussed in Section 3.2. The results were summarized at the third row of Table 3. Compared to the layout optimization of *CONV* (second row of Table 3), layout transformation elimination further accelerates the execution time by $1.1-1.5\times$. *NeoCPU* uses a systematic way to eliminate the unnecessary data layout transformation by inferring the data layout throughout the computation graph and inserting the layout transformation nodes only if needed, which is not seen in other works.

4.2.3 Optimization scheme search

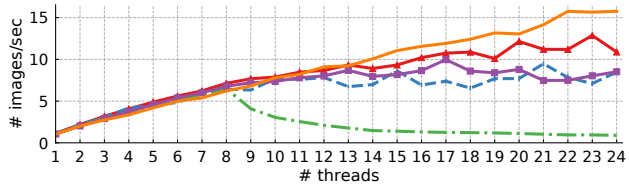
Next, we compare the performance between the optimization schemes produced by our search algorithm and the ones carefully picked by us manually. By comparing the third and fourth row of Table 3, our algorithm (described in Section 3.3) is able to find the (approximately) best combination of data layouts which outperforms the manually picked results by $1.1-1.5\times$. ResNet-50 (and its variants) gains more speedup from global search because the network structure is more complicated, hence leaving more optimization room. In contrast, VGG-19 (and its variants) gains less since the structure of this model is relatively simple. SSD utilizes the approximation algorithm and gets significant speedup, too. The results also verify that, with automatic search, we can get rid of the tedious manual picking of parameters by producing even better results. To the best of our knowledge, *NeoCPU* is the only one that does this level of optimization.

4.2.4 Multi-thread parallelization

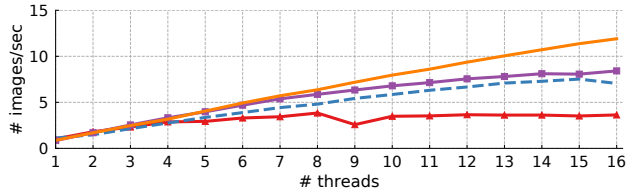
Lastly, we did a strong scalability experiment using the multi-threading implementations backed by our own thread pool described at Section 3.1.2 and the commonly used OpenMP API implemented in the GCC compiler. We also included the result of MXNet, TensorFlow and OpenVINO using Intel MKL-DNN, OpenBlas or Eigen (all realizing multi-threading via OpenMP) for comparison. We configured OpenMP via environment variables to make sure that the jobs are statically partitioned and each thread runs on a disjoint core, which resemble the behavior of our thread pool for apples-to-apples comparison. Figure 4 summarizes the number of images a model can inference one by one (i.e. *batch size* $= 1$) in a second as a function of the number of threads the model inference uses. For the sake of space, we demonstrate one result for one CPU type. The figure shows that our thread pool achieves better scalability than OpenMP in *NeoCPU* as well as in the baselines. Although the tasks are embarrassingly parallelizable, each model inference consists of a number of parallelization regions. The overhead of OpenMP to launch and suppress threads before and after a region is larger than our thread pool, which attributes to the less scalability of OpenMP. Furthermore, sometimes we observed that the performance obtained by OpenMP jitters, or even drops, while adding threads. In addition, the performance of OpenMP may differ across different implementations. In summary, our evaluation suggests that in our use cases, it is preferable to have a self-customized thread pool with full control.



(a) ResNet-50 on a system with 18-core Intel Skylake CPU



(b) VGG-19 on a system with 24-core AMD EPYC CPU



(c) Inception-v3 on a system with 16-core ARM Cortex A72 CPU

Figure 4: Scalability comparison between different multi-threading implementations. The standard errors (< 0.4) are too small to be visible in the diagrams.

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 on all kinds of hardware ranging from CPUs [24, 27, 44, 53], GPUs [11, 13], FPGAs [18, 22, 49], to special-purpose accelerators [12, 32]. 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 [38], ASICs [22]). *NeoCPU* is more flexible and combines the operation- and graph-level optimization intelligently. Although this paper focuses on CPUs, the ideas are applicable to other hardware targets.

NeoCPU is based on the TVM stack [9], an end-to-end framework inspired by Halide [39], which expresses a deep learning model into intermediate representations (IRs) and compiles to the machine code. There are several other similar deep learning compilers such as TensorFlow XLA [34], Tensor Comprehensions [46], Glow [40] and DLVM [47]. However, so far none of them has reported CPU inference results on par with what we did (e.g. Glow only optimized single-core performance on CPUs). We believe our proposed solution could be an integral part to these frameworks.

We follow the well-studied ideas implemented in other high-performance libraries [27, 51] to optimize the computationally-intensive *CONV* operations. In addition to the libraries, there are also highly customized optimization works for convolutions and matrix multiplications on Intel CPUs [19, 24]. These works are mostly about individual operation-level optimizations, which do not consider maintaining data layouts through the entire network. Specifically, they carefully investigate the computation nature of convolutions as well as the available CPU resources to fine tune the operations. This kind of optimization is able to maximize the convolution performance on the targeted CPUs but is not very flexible to extend to other platforms and to do joint optimization. Unlike others, we make the optimization as a configurable template so that it is flexible to fit to different CPU architectures and enable the opportunity to surpass manually tuned performance via operation- and graph-level joint optimization.

Our work utilizes auto search to look for optimal solutions. Similar *auto-tuning* ideas were used in other works as well [10, 46, 48]. However, they all focused on performance tuning for single operations, while ours extends the scope to the entire CNN model to search for optimal solutions globally. Recently, we also observed other work optimizing the DNN workloads at the graph level [30]. This work attempts to obtain better global performance using relaxed graph substitutions which may harm the local performance within a few operations. Its non-greedy search idea is conceptually similar to ours and potentially applicable to our solution. The approximation algorithm we employed to deal with the global search for the models with complicated structures (e.g. SSD) is inspired by the application of PBQP in the register allocation problem [6, 17, 20]. This paper leverages the previous idea and applies to a new domain by minor modification.

6 Conclusion

In this paper, we proposed an end-to-end solution to compile and optimize convolutional neural networks for efficient model inference on modern CPUs. The experiments show that we are able to achieve up to 3.45× speedup on 15 popular CNN models on the various kinds of CPUs (Intel Skylake, AMD EPYC and ARM Cortex A72) compared to the performance of the state-of-the-art solutions. The future work includes extending to other convolution computation algorithms such as Winograd and FFT, handling model inference in quantized values (e.g. INT8) and extending our operation- and graph-level joint optimization ideas to work on other hardware platforms (e.g. NVidia GPUs compared with TensorRT). Supporting the optimized model inference in dynamic shapes (e.g. RNNs [25, 50]) is another interesting direction to explore.

Acknowledgments

We would like to thank our shepherd Peter Pietzuch and the anonymous reviewers of the USENIX ATC program committee for their valuable comments which improved the paper a lot. We are also grateful to Tianqi Chen and Animesh Jain for helpful discussion and constructive suggestion.

References

[1] Amd vs intel market share. https://www.cpubenchmark.net/market_share.html. [Online; accessed 13-May-2019].

[2] Arm holdings. https://en.wikipedia.org/wiki/Arm_Holdings. [Online; accessed 13-May-2019].

[3] Neon. https://developer.arm.com/technologies/neon. [Online; accessed 13-May-2019].

[4] Eigen: a C++ Linear Algebra Library. http://eigen.tuxfamily.org/, 2017. [Online; accessed 13-May-2019].

[5] 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. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[6] Cooper K.D. Torczon L. Briggs, P. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst. 16(3) 428–455*, 1994.

[7] David Budden, Alexander Matveev, Shibani Santurkar, Shraman Ray Chaudhuri, and Nir Shavit. Deep tensor convolution on multicores. *arXiv preprint arXiv:1611.06565*, 2016.

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

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: End-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166*, 2018.

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

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

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

[14] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *NIPS ML Systems Workshop*, 2017.

[15] 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. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[16] Deanne Deuermeyer and Andrey Z. Openvino toolkit release notes. https://software.intel.com/en-us/articles/OpenVINO-RelNotes. [Online; accessed 13-May-2019].

[17] Erik Eckstein. *Code optimizations for digital signal processors*. PhD thesis, Vienna University of Technology, 2003.

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

[19] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of high-performance deep learning convolutions on simd architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841. IEEE, 2018.

[20] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with pbqp. *JMLC 2006. LNCS, vol.4228, pp. 346-361*, 2016.

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

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

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

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

[25] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 41. ACM, 2019.

[26] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[27] Intel. Intel math kernel library for deep neural networks (intel mkl-dnn). https://github.com/intel/mkl-dnn, 2018. [Online; accessed 13-May-2019].

[28] James R. (Intel). Intel avx-512 instructions. https://software.intel.com/en-us/blogs/2013/avx-512-instructions, 2013. [Online; accessed 13-May-2019].

[29] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. Optimizing n-dimensional, winograd-based convolution for manycore cpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 109–123. ACM, 2018.

[30] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *SysML*, 2019.

[31] Ziheng Jiang, Tianqi Chen, and Mu Li. Efficient deep learning inference on edge devices. In *SysML*, 2018.

[32] 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. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12. ACM, 2017.

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

[34] Chris Leary and Todd Wang. Xla: Tensorflow, compiled. *TensorFlow Dev Summit*, 2017.

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

[36] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

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

[38] NVIDIA. Nvidia tensorrt. https://developer.nvidia.com/tensorrt, 2018. [Online; accessed 13-May-2019].

[39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 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, pages 519–530. ACM, 2013.

[40] 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. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

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

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

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

[44] Linpeng Tang, Yida Wang, Theodore Willke, and Kai Li. Scheduling Computation Graphs of Deep Learning Models on Manycore CPUs. *ArXiv e-prints*, July 2018.

[45] Tensorflow. Tensorflow performance guide. https://www.tensorflow.org/performance/performance_guide#data_formats, 2018. [Online; accessed 13-May-2019].

[46] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

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

[48] R Clinton Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Supercomputing, 1998. SC98. IEEE/ACM Conference on*, pages 38–38. IEEE, 1998.

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

[50] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 951–965, 2018.

[51] Xianyi Zhang, Qian Wang, and Zaheer Chothia. Openblas. http://xianyi.github.io/OpenBLAS, 2014. [Online; accessed 13-May-2019].

[52] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. Fft convolutions are faster than winograd on modern cpus, here is why. *arXiv preprint arXiv:1809.07851*, 2018.

[53] Aleksandar Zlateski, Kisuk Lee, and H Sebastian Seung. 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*, pages 801–811. IEEE, 2016.