UC Davis
Electrical & Computer Engineering

Title
Towards Flexible and Compiler-Friendly Layer Fusion for CNNs on Multicore CPUs

Permalink
https://escholarship.org/uc/item/9v75738g

Authors
Lin, Zhongyi
Georganas, Evangelos
Owens, John D.

Publication Date
2021-08-25

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Towards Flexible and Compiler-Friendly Layer Fusion for CNNs on Multicore CPUs*

Zhongyi Lin[1] [0000−0002−5992−3913], Evangelos Georganas[2], and John D. Owens[1] [0000−0001−6582−8237]

[1] University of California, Davis, USA
[2] Parallel Computing Lab, Intel Corporation

Abstract. In deep learning pipelines, we demonstrate the performance benefits and tradeoffs of combining two convolution layers into a single layer on multicore CPUs. We analyze when and why fusion may result in runtime speedups, and study three types of layer fusion: (a) 3-by-3 depthwise convolution with 1-by-1 convolution, (b) 3-by-3 convolution with 1-by-1 convolution, and (c) two 3-by-3 convolutions. We show that whether fusion is beneficial is dependent on numerous factors, including arithmetic intensity, machine balance, memory footprints, memory access pattern, and the way the output tensor is tiled. We devise a schedule for all these fusion types to automatically generate fused kernels for multicore CPUs through auto-tuning. With more than 30 layers extracted from five CNNs, we achieve a 1.04x geomean with 1.44x max speedup against separate kernels from MKLDNN, and a 1.24x geomean with 2.73x max speed up against AutoTVM-tuned separate kernels in standalone kernel benchmarks. We also show a 1.09x geomean with 1.29x max speedup against TVM, and a 2.09x geomean with 3.35x max speedup against MKLDNN-backed PyTorch, in end-to-end inference tests.

Keywords: Layer fusion · CNN · Multicore CPUs · Auto-tuning.

1 Introduction

Convolutional neural networks (CNNs) have played an increasingly important role in research and industry for the past decade. CNNs are constructed with a series of layers/operators (*ops*). In the vast majority of CNN implementations, ops have a one-to-one correspondence with compute kernels. For reduced memory requirements and/or better producer-consumer locality, production CNNs perform *fusion* for certain ops, i.e., combining two or more neighboring ops into one and computing it with one single kernel. At its core, the fusion problem is a balancing problem between computation and communication, or between the communication of different memory hierarchies, with the hope that reducing main-memory communication will result in only modest amounts of extra computation or data movement in high-level memory and hence an overall

* Supported by Intel Labs.

speedup. In particular, today's deep learning (DL) frameworks and compilers (e.g., TVM [?], Tensorflow [?], PyTorch [?], and MXNet [?]), and proprietary kernel libraries like cuDNN [?] and MKLDNN often trivially fuse a convolution layer with the following element-wise layers to save the cost of data movement. Other fusion opportunities such as parallel convolution branch fusion for structures in models like Inception-V3 [?] can be realized by methods like relaxed graph substitution, as proposed by Jia et al. [?].

More complex is the fusion of neighboring *complex ops*, e.g., convolution (conv) and depthwise convolution (dw-conv). As these ops often appear as the hotspot of CNNs, which are compute and/or memory bandwidth intensive, fusing them is a potential way to further enhance compute performance. Showing performance improvements for this fusion type is a significant challenge for three reasons. First, unlike the aforementioned fusion types, simple techniques like inlining or data concatenation and split (to reuse existing proprietary libraries) would not work due to the memory access pattern of these ops. Instead, new compute kernels are necessary, and the optimization space of a fused op is so complex, with so many parameters, that a straightforward search would be impractical. Second, the standalone kernels used as a performance baseline are already highly optimized. Finally, integrating a fused kernel into current DL frameworks is also difficult, as modern frameworks are primarily designed at the granularity of one-op that maps typically to one-kernel.

In this paper, we focus on three opportunities for the fusion of two consecutive complex ops: (1) 3-by-3 dw-conv with 1-by-1 conv, (2) 3-by-3 conv with 1-by-1 conv, and (3) two 3-by-3 convs, all of which are commonly seen in CNNs. The challenge we address in this paper is to show not just *where* we can show performance improvements from complex op fusion but also *why*. We first propose a set of tiling and scheduling principles to intelligently reduce the otherwise intractable parameter space and search for the combination that leads to the best performance. These principles can also be extended to problems with complex loop structures, including other fusion types. Based on these principles, we devise a schedule template for auto-tuning these fused kernels on multicore CPUs, and propose an approach of integrating the fused kernels/ops into DL compiler pipelines. Both of these ideas can be adopted by production DL compilers. We implement the fused kernels by incorporating LIBXSMM's [?] batch-reduce GEMM [?] micro-kernels and extending AutoTVM, TVM's auto-tuning tool, to search for the best schedule. We also integrate these kernels into TVM's compiler infrastructure for end-to-end tests by creating a new fused op.

We make the following contributions:

- We analyze the fusion of two complex ops and answer the question when and why such fusion is beneficial.
- We propose a methodology with tiling and scheduling principles of composing fused kernels for multicore CPUs, and implement kernels for three types of two complex-op fusion.
- We achieve 1.24X and 1.04X geomean speedup in a standalone kernel-level benchmark against TVM and MKLDNN respectively, and 1.09X and 2.09X

geomean speedup in end-to-end tests, by testing with more than 30 workloads extracted from five real CNN models on three multicore CPU platforms.

2 Related Work

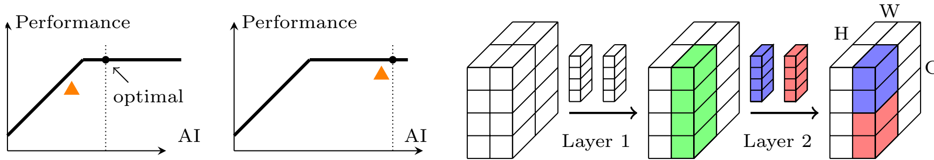
Kernel Optimizations and Auto Tuning Convolutions are the ops that take the largest runtime in current CNNs. They can be implemented in many styles, e.g., direct convolution, im2col + GEMM [?], FFT-based [?], and Winograd-based [?], etc, on different types of compute devices. For direct convolution on CPUs, Georganas et al. [?, ?] proposed batch-reduce GEMM (BRGEMM) as a basic building block for tensor contractions and convolution and claimed to achieve better runtime performance than MKLDNN (now renamed to oneDNN). In our work, we adopt the BRGEMM micro-kernel implementation provided in the LIBXSMM [?] library as a building block for our fused kernels.

Autotuning is a common approach for optimizing kernel implementations and has benefited from continuous development through the years. Autotuning can be employed together with the idea of decoupling compute and schedule central to Ragan-Kelley et al. [?, ?] and developed by Mullapudi et al. [?] in Halide as well as Chen et al. [?, ?] in TVM, easing the process of developing high-performance implementations for DL workloads. Recently, the idea of autoscheduling [?, ?] breaks through auto-tuning's limit of relying on a well-composed schedule template by automatically searching for schedules. In this research we adopt AutoTVM as our tool for auto-tuning; in future work, we plan to also port our approach to autoscheduling tools.

Layer Fusion The fusion of multiple consecutive convolutions first appears in Alwani et al. [?]. Their kernel implementation is completely hand-tuned on FPGAs and the fusion space is explored with a dynamic programming approach. Wang el al. [?] improved inference time on GPUs using a designated subset of fused convolutional layers. However, these works neither explore the large space of code optimization nor show a clear way how the fused kernels can be handily integrated into production end-to-end tests. Our work is the first that focuses on multicore CPUs and addresses the above issues.

3 Principles of Effective Layer Fusion

Consider a CNN model that runs on a multicore system, e.g., a multicore CPU. A well-optimized single-op kernel on this system will take advantage of all cores. Typically such a kernel will output a tensor and that tensor is divided into multi-dimensional *tiles*, with an equal number of tiles computed on each core in parallel. This tiling may be a spatial tiling (e.g., across one or more axes among N, H, W in a typical 4D activation tensor) and/or a tiling across channels (e.g., the C axis). At the end of the kernel, all data is written back to memory before the next kernel begins. The next kernel will then read its input from memory assuming the input is large enough and does not to fit in any level of cache.



(a) Roofline model examples. Dotted lines are theoretical fused kernel AI. Orange triangles are empirical fused AI (efAI) derived from separate kernel stats. Fusion is likely: **(left)** beneficial, as efAI is not close to optimal; or **(right)** not beneficial as efAI is very close to optimal.

(b) Tiling across the channel axis (C) of the second layer results in recomputation on different cores: the blue and red tensors reside on different cores, while both cores need to compute the green tensor.

Fig. 1

In this paper, we show performance gains from fusing kernels of two complex ops. The important contribution of this paper, however, is not the performance gains but instead why and how we achieve these performance gains. What kind of kernels should be fused, and how should we fuse them?

One of our key tools for analysis and verification is the roofline model [**?**], with which we can determine if a kernel's performance is bound by memory or compute. We characterize the combination of the two separate kernels in the roofline model, with the total compute equal to the sum of the compute in the two kernels, and the memory requirements equal to the sum of the reads and writes for both kernels. If the fused-kernel result is memory-bound, the two kernels are *usually* good candidates for fusion, because fusion's primary benefit is saving the memory writes and reads between the two kernels, with the hope to replace slow (main memory) accesses with fast (cache) accesses. However, this is trickier in practice, since many successful fusions we perform tend to be compute-bound. We also see successful fusion for two compute-bound kernels (as we show in a later example) as well as failure for fusion involving extremely memory-bound kernels like 5-by-5 dw-conv. Nevertheless, the results from the roofline model are generally predictive. We can also refer to the empirical results of proprietary separate kernels to select workloads that might benefit from fusion. As shown in Figure 1a, fusion is likely beneficial if the derived empirical fused roofline is not too close to the peak throughput at the theoretical fused AI, as fusion moves the roofline towards the *upper right* if it speeds up. In contrast, fusion is likely not beneficial if the derived empirical fused roofline is almost optimal.

We begin by looking at the per-core output of the first kernel and the per-core input of the second kernel. For some pairs of kernels, these are identical, and we can simply concatenate these into one kernel in a straightforward fashion. More often, though, these two do not match. In these cases, writing to memory at the end of the first kernel serves two purposes. The first is to allow a reshuffling of data through the memory system (essentially, a permutation of the intermediate output tensor, distributed across cores). The second is to allow a broadcast so that the output of one core in the first kernel can serve as the input for multiple cores in the second kernel. If we implement a fused kernel, our implementation must either perform a significant amount of intra-core communication for data reuse or perform redundant recompute. Notice that the memory footprint of

fused workloads might fit in any level of cache or none of them. For the extreme case that the footprint fits in L1, there is no data movement cost of the intermediate output to reduce, as it always stays in this fastest cache. We discover that this almost never happens with real CNN workloads, and therefore, from this point of view, fusion is always worth trying.

We employ *tiling* and *scheduling* to find the balance between fusion and speedup. Tiling expresses the subdivision of tensor input/intermediate/output data in a way that allows effective and scalable parallel execution. The computation of each part of the output tensor is typically expressed as a series of nested loops, and we also have the freedom to schedule (i.e., reorder, split, merge, parallelize, etc.) these loops to optimize for locality and execution. Because the two unfused kernels are almost certainly highly optimized for the target architecture when using proprietary libraries, we must make near-optimal decisions for tiling and scheduling to achieve competitive performance with our fused kernel.

Tiling Principles One of the most important decisions in tiling is choosing along which axis to tile. In our fastest kernels on CPUs, we prefer tiling the second layer along spatial axes to tiling along channel axes, because the latter requires (redundant) recomputation of layer-one entries that are inputs to multiple different tiles along layer-two's channel axis, as shown in Figure 1b. This rules out some compute-heavy kernels like the last few layers of ResNets (i.e., $res_4x/5x$ as examples), which typically tile along a (relatively long) channel axis.

The spatial axes of the second layer also need to offer sufficient parallelism for a full tiling; without enough parallelism here, fusion does not make sense. In general, CNN kernel implementations on CPUs tend to pack tensors so that the channel axis is packed as vectors in the last dimension, so CPUs with longer vector length, e.g., AVX-512, are better candidates for fusion, since they exploit the parallelism along the channel axis and compensate for our reluctance to tile/parallelize that axis across cores. Also, more cores make fusion more attractive if batch size goes up and/or the spatial dimension is large, since either of these cases expose more potential parallelism.

Finally, for effective fusion, the tiles for the first and second kernels in the separate case should be comparable in size. Given a fixed-size cache, a significant mismatch in size between the two tiles reduces the opportunity for capturing producer-consumer locality, as it leverages the cache poorly. The $mnb1$ workloads shown later in Table 2 are examples of such a failure.

Scheduling Principles Once we have determined our tiling, we turn to the problem of scheduling. A typical fused kernel in our pipelines of interest has on the order of a dozen loops as well as the option to split these loops. Any sort of exhaustive search over valid reorderings of these loops is virtually intractable. Yet our experience is that some search is necessary; the performance landscape of the many possible implementations is complex enough that auto-tuning is necessary to find the fastest fused kernel.

Our approach is to restrict the search space down to a manageable level by only searching over a subset of the loops. In particular, we do not attempt to

change the order of the outermost and innermost loops as they either do not affect data locality or are fixed for optimal register usage within a micro-kernel. In contrast, we do search the remaining loops, whose reordering can have a significant impact on data locality. This will be discussed in the next section.

4 Implementation

We mainly focus on three types of 2-layer fusion: (1) 3-by-3 depthwise convolution (dw-conv) followed by 1-by-1 convolution (conv); (2) 3-by-3 conv followed by 1-by-1 conv; (3) two 3-by-3 convs. The first type occurs commonly in computationally lightweight CNN models, e.g., MobileNet-V1 [**?**], MobileNet-V2 [**?**], MNasNet-A1 [**?**], etc. The second and third types occur in computationally heavyweight CNN models like ResNets [**?**], etc. In this section, we first introduce how we compose schedules to generate kernels for these fusion types, followed by how these kernels are integrated into the TVM inference pipeline.

Kernel-level optimization Often a part of a domain-specific compiler, modern autotuners usually input the *schedule* that we described above, which describes how the mathematical expression of an op is mapped to the hardware (e.g., loop orderings and manipulations, as well as the search for the split loop lengths and ordering combinations that lead to the best performance) to tune the kernel. This may result in many possible mappings and hence a large search space. For example, two axes of length 4 being split and reordered has a search space size of 3 (split of first axis) $\times$ 3 (split of second axis) $\times$ 2 (reordering) = 18. If two layers are naively fused, the search space size grows exponentially and becomes intractable, even without considering the extra possibility of loop unrolling in the innermost loops. As an aside, though they do not affect the search space, an autotuned fuser must also efficiently integrate element-wise post ops like batch-normalization, ReLU, etc. that follow all complex ops except for the last one.

Our goal is to achieve high performance on the fused kernel and meanwhile limit the search space to make searching tractable. We accomplish this by classifying loops into three categories: parallel loops, micro-kernel loops, and tunable loops. We determined robust, fixed strategies for the first two categories and thus reduce our search space to only searching for the optimal configuration of the third.

We choose to fix the ordering of the first two categories of loops, because reordering parallel loops is trivial for batch size 1, while micro-kernel loops are mapped to ready-to-use micro-kernels with fixed loop order. In our schedules, we place the parallel loops at the outermost location, micro-kernel loops at the innermost location, and tunable loops in between. The skeleton of our implementation structure is shown in Algorithm 1 and Fig 2. We implement our kernel with the BRGEMM micro-kernels from LIBXSMM. Instead of using the common $NCHW$ or $NHWC$ formats for convolution, we use a packed format, e.g., $NCHW[x]c$ for feature maps, and $(OC)(IC)H_fW_f[x]ic[y]oc$ for weights, where $x, y = 8, 16, 32, 64\ldots$, for better data locality on CPUs [**?**]. At line 3 to 5 of the

Algorithm 1 Fused kernel schedule template with BRGEMM micro-kernels.

1: **Inputs:** $input \in \mathbb{R}^{N \times IC_1 \times IH \times IW \times ic_1}$, $weights_1 \in \mathbb{R}^{OC_1 \times IC_1 \times FH_1 \times FW_1 \times ic_1 \times oc_1}$, $weights_2 \in \mathbb{R}^{OC_2 \times IC_2 \times FH_2 \times FW_2 \times ic_2 \times oc_2}$, optional post ops parameters, e.g., $bias_1 \in \mathbb{R}^{OC_1 \times oc_1}, bias_2 \in \mathbb{R}^{OC_2 \times oc_2}$
2: **Outputs:** $output \in \mathbb{R}^{N \times OC_2 \times OH \times OW \times oc_2}$
3: **Split** OH into H_t, H_o, and H
4: **Split** OW into W_t, W_o, and W
5: **Split** IC_2 into IC_o and IC_i
6: **for** fused($n = 0 \dots N-1$, $ht = 0 \dots H_t - 1$, $wt = 0 \dots W_t - 1$) **do**
7:   **Exhaustively search** the order of OC_2, IC_o, H_o, and W_o, and mark them as loop 1, 2, 3, 4, and the parallel loop as loop 0 {e.g., IC_o is loop 2.}
8:   **Arbitrarily pick** a loop x from loop 0, 1, 2, 3, 4 {e.g., x is 3.}
9:   **for** loop 1 **do**
10:     **for** loop 2 **do**
11:       **for** loop 3 **do** {Sub-tensors compute here.}
12:         **for** loop 4 **do**
13:           BRGEMM micro-kernel for layer 1 sub-tensor
14:         **end for**
15:         Compute post ops of layer 1 if necessary
16:         **for** loop 4 **do**
17:           BRGEMM micro-kernel for layer 2 sub-tensor
18:         **end for**
19:       **end for**
20:     **end for** {IC_o finishes.}
21:     Compute post ops of layer 2 if necessary
22:   **end for**
23: **end for**

algorithm, OH, OW, and the reduce loop IC_2 of the *output* tensor are split into 10 loops including the unsplit N. Each loop is split so that the lengths of each sub-loop are factors of its length. We do not split OC_2 as tiling across it brings recomputation as we discuss above. The parallel loops, i.e., N, H_t, and W_t, are fused and parallelized across multiple CPU cores at line 7. Loops H, W, and $oc_{1/2}$ and reduce loops, including FH, FW, IC_i, and $ic_{1/2}$, are all micro-kernel loops expressed within BRGEMM micro-kernels. The remaining four loops, i.e. OC_2, IC_o, H_o, and W_o, are left as the tunable loops to be searched by the auto-tuner.

In our implementation, we exhaustively search for all the orderings of these four loops and at which loop layer we place the computation, i.e., *compute_at*. The sub-tensors of both layers are computed at a loop that is picked among any of these four loops or the fused parallel loop. From a cache point of view, the input feature map sub-tensor of layer one is distributed to different cores on the CPUs, while the weights of each layer are streamed to cache. The sub-tensor output of layer one is computed with vanilla loop ordering if layer one is a dw-conv, or by calling the BRGEMM micro-kernel if layer one is a conv. Its size is inferred by the compiler given the output size of its consumer, i.e., (H, W, oc_2) is known. This sub-tensor is always complete, i.e., all its reduce loops are fully contracted, before it is consumed by layer two, because otherwise it incurs extra

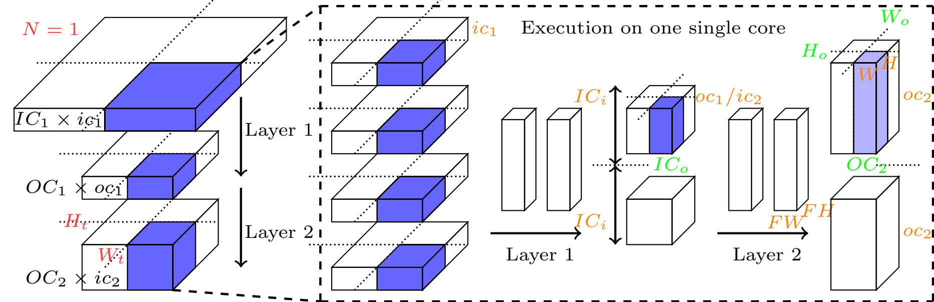


Fig. 2: Algorithm visualization. (**Left**) Tensors are tiled for parallel multicore execution. (**Right**) A light blue (incomplete, as loop IC_o is not fully contracted) sub-tensor of size (H, W, oc_2) is computed from all blue (complete) sub-tensors. Red/green/orange mark the parallel/tunable/micro-kernel loops, respectively.

computation for each of the incomplete slices. Therefore, it is safe to insert any post ops computation directly after it. Subsequently, this sub-tensor always stays in cache and is consumed by layer two to produce a slice with size (H, W, oc_2). This slice is not necessarily complete as the loop x it computes could be inside the outermost reduce loop of all, IC_o. We insert the post ops right after IC_o because they can only be computed when IC_o is fully contracted. Therefore, the sub-tensor of layer two always stays in cache for proper tiling factors until it is complete and no longer needs access.

We found that grouping the loops in this way greatly limits the search space, making auto-tuning feasible, but still results in high-performance fusion. It can also be extended more generally to polyhedral problems including fusing other types of layers. In this case, a performant micro-kernel is necessary to serve as the core of the output schedule.

AutoTVM implementation of fused kernel We realize this implementation as an AutoTVM schedule. We first define TE compute functions for the fused layer workloads, then we create AutoTVM tuning tasks with these compute functions and schedules so that AutoTVM can auto-tune them using the XGBoost algorithm. For fast convergence, the schedule is tuned without post ops being added since they do not affect the results, while when the tuning config is produced, we apply it to a new inference schedule where post ops are added as necessary. The methodology can be handily extended to multiple layers with the (straightforward) addition of compiler support, as we can simply follow the same rule by only blocking the last layer being fused and stacking all previous layers at loop x. In fact, this methodology resembles the (manual) 'pyramid' method proposed by Alwani et al. [**?**], but we generate the fused kernel code automatically via auto-tuning. We also notice that at the cost of searching a much bigger space by further splitting the second loop group into eight loops and reordering them, higher fused kernel performance could potentially be achieved with better cache blocking options. However, the advantage of this schedule against ours is marginal, which also requires both smartly designed heuristics and a more powerful autotuner. Hence, we leave the exploration of this idea as future work.

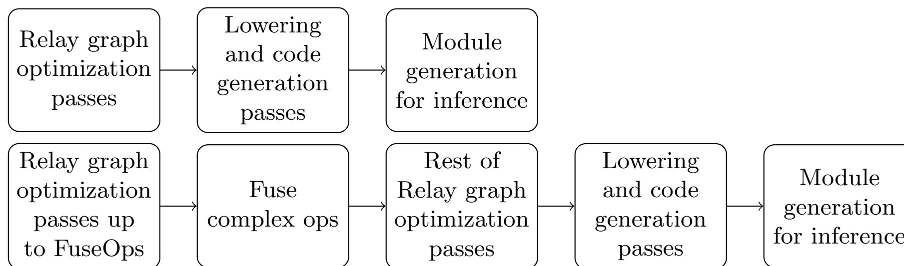


Fig. 3: **Top:** TVM pipeline without complex-op fusion; **Bottom:** TVM pipeline with complex-op fusion.

Compiler integration of fusion for end-to-end test To integrate the fused kernel into end-to-end (full-CNN) tests, we need (1) a new op that defines the compute of the fused conv layers in a compute graph and (2) a compiler pass that is inserted into the pipeline to rewrite the graph for fusion. Both of these are missing in all DL frameworks. Also, post ops like batch-normalization (BN) of the layers being fused need to be properly handled. Since BN ops are usually simplified to *bias-add*s with parameters such as *mean* and *gamma* folded into *weights* for inference, we do not keep them in the compute function for our *fused-conv2D* op. We simply keep the *bias* tensors of each layer together with the input and weights as the inputs to the *fused-conv2D* op, such that the new op is equivalent to a sub-graph of two fused layers and their post ops, e.g., dw-conv/conv+bias-add+relu+conv+bias-add+relu. As currently most DL frameworks fuse only element-wise ops, we must ensure that the complex-op-fusion pass is inserted *after inference simplification* where ops like BN are simplified, and *before element-wise op fusion* where all element-wise ops are fused.

Figure 3 presents how we leverage TVM's existing compiler pipeline to integrate our design into it. Following the above principles, we create a new Relay op for the fused conv layers, as well as a new compiler pass to detect the fusable patterns and rewrite them with the *fused-conv2D* ops. In the TVM pipeline, a Relay compute graph is passed through passes that optimize the graph structure, and then passes for code generation, and finally module generation. We insert the new *fuse-conv* pass right before the *fuse-ops* pass such that post ops like *bias-add* ops are properly handled for the *fused-conv2D*, while post ops are still normally handled for other unfused complex ops in the following *fuse-ops* pass. Our *fused-conv2D* op is also compatible with TVM's graph tuning [**?**] for layout optimization for CPU inference.

5 Results and Analysis

We extract 33 eligible layers with batch size 1 from five CNN models, including MobileNet-V1, MobileNet-V2, MNasNet-A1, ResNet-18, and ResNet-50. The full list of layers can be found in Table 2. We auto-tune each kernel for 4000 iterations using AutoTVM with XGBoost as the searching algorithm. We conduct both throughput and roofline analysis on these workloads, and integrate them into the

Fig. 4: Throughput comparison between fused and separate kernels without post ops, normalized with respect to MKLDNN. A "TVM fused" result of greater than one indicates our fusion delivers better performance than MKLDNN. **Top:** Intel i7_7700K (511 GFLOPS); **middle:** GCP Intel (711 GFLOPS); **bottom:** GCP AMD (413 GFLOPS).

end-to-end tests of the five models. The experiments are conducted on one Intel Core(TM) i7_7700K CPU @ 4.2 GHz, one Intel Xeon quad-core Google Cloud Platform (GCP) server (unknown model with Cascade Lake microarchitecture) @ 3.1 GHz, and one AMD EPYC 7B12 quad-core GCP server @ 2.25GHz, all with Linux Ubuntu 18.04, Python 3.8, and PyTorch 1.8.1. We run our kernel-level experiments for fused kernels against separate kernels shipped by MKLDNN and those generated by AutoTVM as baselines, and model-level experiments against the baselines of AutoTVM + graph tuner as well as MKLDNN-backed PyTorch. For roofline analysis, we measure the DRAM bytes with PCM, and measure the cache bytes and FLOP counts of the kernels with SDE.

Kernel-level Experiments In Figure 4, we show the throughput of all the kernels on three platforms normalized with respect to MKLDNN in the standalone kernel benchmark. Overall, we achieve {geomean, maximum, and minimum} speed-ups of fused-kernel against MKLDNN-separate-kernels of {1.04X, 1.44X, and 0.53X}, and against TVM-separate-kernels of {1.24X, 2.73X, and 0.63X}. As expected, we see that workloads with big activations in their first layer such as $mv1_1$, $mv2_1$, $mv2_2_1$, etc. have higher speed-ups. Comparing all the MobileNet-V1

and MobileNet-V2 workloads, i.e., *mv1_x* and *mv2_x*, we can see that although the spatial size of the activation changes almost the same way throughout the model, e.g., $112 \rightarrow 56 \rightarrow 28 \rightarrow 14 \rightarrow 7$, we see a higher overall fusion speed-up on *mv2_x* than on *mv1_x*. This is because *mv2_x* tends to scale *down* the channel axis between its fused layers, e.g., *mv2_1* has 32 channels for its 3-by-3 dw-conv and only 16 channels for its 1-by-1 conv, while *mv1_x* tends to scale *up*. Therefore, generally *mv2_x* is less compute-bound than *mv1_x* and in these cases, closer to the machine balance, as we will show later. We can thus suggest to model designers that without sacrificing accuracy, models can benefit more from fusion if the sizes of adjacent layers are designed to have more balanced AI. In addition, we still see a few examples where considerably compute-bound ResNet workloads are sped up by fusion, which suggests that even compute-bound kernels could also benefit from our methods.

Across platforms we observe that GCP Intel, the only AVX-512 CPU with the highest peak throughput in our evaluation, achieves the highest geomean speed-up (1.13X) on fused kernels against MKLDNN, versus (1.01X) and (0.99X) on the other two platforms. This implies the fact that fusion works better on platforms with higher peak throughput (making workloads 'less compute-bound').

We also plot the roofline model of all the fused and separate kernels we test on the i7_7700K CPU in Figure 5. We treat each pair of TVM separate kernels and MKLDNN kernels as one standalone kernel and derive its AI as $\mathrm{AI}_s = (\mathrm{flop}_1 + \mathrm{flop}_2)/(\mathrm{bytes}_1 + \mathrm{bytes}_2)$, where bytes_1 and bytes_2 are measured memory traffic for either DRAM or cache. We observe a trend that the roofline of the fused kernel gets closer and closer to that of the separate kernels for both DRAM and L2, especially in workloads from *mv1_1* to *mv1_13* in MobileNet-V1. This again verifies that fusion tends to get less benefit at later layers than earlier ones. Typically, for workloads in MobileNet-V2 and MNasNet-A1 such as *mv2_2_1*, *mv2_3_1*, etc, the fused kernels have a smaller advantage on AI compared to their predecessor or successor workloads that have either similar input or output feature sizes. This is because the bottleneck of these workloads is the stride-2 access of their input feature maps that reduces the data reuse; this bottleneck is not relieved by fusion. We also verify that overall the theoretical peaks DRAM AI for *mv2_x* tend to be closer to the machine balance than *mv1_x* so that they benefit more from fusion. We see that most of the fused kernels still do not reach the theoretical peak DRAM AI and incur extra DRAM accesses, which means there is still room for optimization by extending the schedule's search space. Notice that in all cases the L2 AI moves to the *right*, indicating that fusion increases data reuse in L2 cache. For workloads being sped up, the L2 AI moves towards the *upper-right*, which matches our expectation. Among those not sped up, the rooflines of ResNet workloads are very close to the peak throughput and theoretical peak fused AI intersection, matching our previous synthetic examples that project such workloads are less likely to benefit from fusion.

End-to-End Experiments We present the results of the end-to-end inference tests for the five models on three CPU platforms in Table 1. For each model, we tune

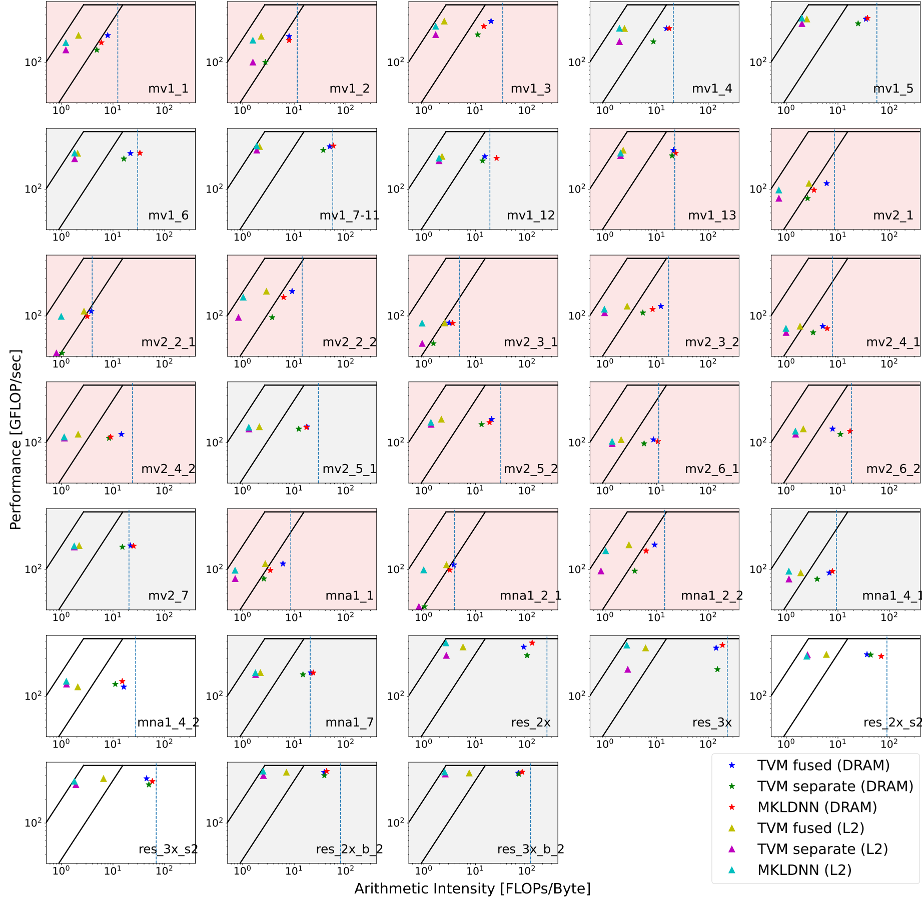


Fig. 5: Standalone kernel rooflines of all the fused and separate kernels without post ops on the Intel i7_7700K CPU. In most cases, both DRAM and L2 AI of the fused kernel moves towards the *upper-right* as fusion delivers better performance. The left and right slopes represent the L2 bandwidth (198.9 GB/s) and DRAM bandwidth (34.7 GB/s). The vertical blue dotted line represents the theoretical peak DRAM AI for the fused kernel. The red and grey background mark layers in which our fused kernel beats both and at least one, respectively.

variants of the compute graph that use a fused form of kernel pairs where we have seen a kernel-level advantage, and for layers that fusion does not have an advantage, e.g., *mv2_7*, *mna1_7*, etc., both fused and not fused, as the fused version might still perform better if layout transformations are needed for the unfused versions. Then we select the instance with the shortest inference time. In practice, we see only a few variants per model and this tuning step is short.

In all test cases except for ResNet-18 on i7_7700K and GCP AMD, fusion speeds up inference, with up to 3.35X against PyTorch for MobileNet-V2 on GCP Intel and 1.29X against TVM-separate for MobileNet-V2 on GCP AMD. We observe that the end-to-end speed-up is not exactly aligned with the aggregation of individual kernel speed-ups. In the standalone kernel benchmark, the cache is flushed for each iteration and the input data of layer one is always read from DRAM. But in end-to-end tests, the output data of a layer stays in cache and

Table 1: End-to-end inference time (in millisecond) of five models on three CPUs. Shortest inference time across tools are shown in bold. Fusion doesn't apply to ResNet-18 on GCP AMD, since the fusion performance of two 3-by-3 convs is inferior to that of both TVM separate and MKLDNN.

CPU types	Models	TVM fused	TVM separate	PyTorch
Intel i7_7700K	MobileNet-V1	**3.38**	3.58	6.11
	MobileNet-V2	**2.44**	2.92	7.28
	MNasNet-A1	**3.31**	3.73	6.53
	ResNet-18	9.69	**9.35**	10.58
	ResNet-50	**20.42**	20.66	26.80
GCP Intel	MobileNet-V1	**2.77**	3.00	5.97
	MobileNet-V2	**2.35**	2.89	7.88
	MNasNet-A1	**3.42**	3.66	7.62
	ResNet-18	**7.17**	7.24	9.79
	ResNet-50	**16.18**	16.19	24.96
GCP AMD	MobileNet-V1	**7.72**	8.42	8.83
	MobileNet-V2	**4.65**	6.01	10.23
	MNasNet-A1	**6.79**	7.35	9.83
	ResNet-18	-	**15.38**	15.47
	ResNet-50	**38.86**	39.82	39.60

may allow the next layer to benefit from producer-consumer locality. Also, the framework itself might also affect the end-to-end results.

We see that the speedups of ResNets are marginal (up to 1.02X) on all three CPU platforms, matching our expectation that fusion for compute-bound layers has limited benefit. We only fuse res_2x for ResNet-18, and res_2x and/or res_3x for ResNet-50, while it is the number of res_4x that primarily drives the depth of ResNets; thus we expect to see marginal benefit from kernel fusion on ResNets that have more than 50 layers.

6 Conclusion and Future Works

Individual kernels, such as those inside MKLDNN or NVIDIA's cuDNN, are highly optimized. They set a high bar for the implementation of fused kernels. One conclusion we draw is that the benefits of fusion are dependent on both the characteristics of the workloads and the CPU on which they are run, and the benefits of fusion are difficult to predict and, at this point, require actually implementing and running the kernels.

In future work, we hope to integrate the idea of fusion with TVM's autoscheduler [**?**] so as to leverage its power to search for high-performance schedules for fused layers. Next, we plan to target the compiler level to enable intermediate padding so that fusion can be extended to more layer types. Finally, we would also like to study fusion for the cases when batch size is greater than 1.

Table 2: Layer table. Input sizes are in (N,H,W,C) format, while layer configs are (filter HW, output channel or multiplier, stride HW, layer type, post op). Layers that do not benefit from fusion are crossed out and not shown in the result section.

Models	name	input	layer 1	layer 2
MobileNet-V1	mv1_1	(1, 112, 112, 32)	(3, 1, 1, dw-conv, relu)	(1, 64, 1, conv, relu)
	mv1_2	(1, 112, 112, 64)	(3, 1, 2, dw-conv, relu)	(1, 128, 1, conv, relu)
	mv1_3	(1, 56, 56, 128)	(3, 1, 1, dw-conv, relu)	(1, 128, 1, conv, relu)
	mv1_4	(1, 56, 56, 128)	(3, 1, 2, dw-conv, relu)	(1, 256, 1, conv, relu)
	mv1_5	(1, 28, 28, 256)	(3, 1, 1, dw-conv, relu)	(1, 256, 1, conv, relu)
	mv1_6	(1, 28, 28, 256)	(3, 1, 2, dw-conv, relu)	(1, 512, 1, conv, relu)
	mv1_7-11	(1, 14, 14, 512)	(3, 1, 1, dw-conv, relu)	(1, 512, 1, conv, relu)
	mv1_12	(1, 14, 14, 512)	(3, 1, 2, dw-conv, relu)	(1, 1024, 1, conv, relu)
	mv1_13	(1, 7, 7, 1024)	(3, 1, 1, dw-conv, relu)	(1, 1024, 1, conv, relu)
MobileNet-V2	mv1_1	(1, 112, 112, 32)	(3, 1, 1, dw-conv, relu6)	(1, 16, 1, conv, bias)
	mv2_2_1	(1, 112, 112, 96)	(3, 1, 2, dw-conv, relu6)	(1, 24, 1, conv, bias)
	mv2_2_2	(1, 56, 56, 144)	(3, 1, 1, dw-conv, relu6)	(1, 24, 1, conv, bias)
	mv2_3_1	(1, 56, 56, 144)	(3, 1, 2, dw-conv, relu6)	(1, 32, 1, conv, bias)
	mv2_3_2	(1, 28, 28, 192)	(3, 1, 1, dw-conv, relu6)	(1, 32, 1, conv, bias)
	mv2_4_1	(1, 28, 28, 192)	(3, 1, 2, dw-conv, relu6)	(1, 64, 1, conv, bias)
	mv2_4_2	(1, 14, 14, 384)	(3, 1, 1, dw-conv, relu6)	(1, 64, 1, conv, bias)
	mv2_5_1	(1, 14, 14, 384)	(3, 1, 1, dw-conv, relu6)	(1, 96, 1, conv, bias)
	mv2_5_2	(1, 14, 14, 576)	(3, 1, 1, dw-conv, relu6)	(1, 96, 1, conv, bias)
	mv2_6_1	(1, 14, 14, 576)	(3, 1, 2, dw-conv, relu6)	(1, 160, 1, conv, bias)
	mv2_6_2	(1, 7, 7, 960)	(3, 1, 1, dw-conv, relu6)	(1, 160, 1, conv, bias)
	mv2_7	(1, 7, 7, 960)	(3, 1, 1, dw-conv, relu6)	(1, 320, 1, conv, bias)
MNasNet-A1	mna1_1	(1, 112, 112, 32)	(3, 1, 1, dw-conv, relu)	(1, 16, 1, conv, bias)
	mna1_2_1	(1, 112, 112, 96)	(3, 1, 2, dw-conv, relu)	(1, 24, 1, conv, bias)
	mna1_2_2	(1, 56, 56, 144)	(3, 1, 1, dw-conv, relu)	(1, 24, 1, conv, bias)
	mna1_4_1	(1, 28, 28, 240)	(3, 1, 2, dw-conv, relu)	(1, 80, 1, conv, bias)
	mna1_4_2	(1, 14, 14, 480)	(3, 1, 1, dw-conv, relu)	(1, 80, 1, conv, bias)
	mna1_7	(1, 7, 7, 960)	(3, 1, 1, dw-conv, relu)	(1, 320, 1, conv, bias)
~~MNasNet-B1~~	~~mnb1_3_1~~	(1, 56, 56, 72)	(5, 1, 2, conv, relu)	(1, 40, 1, conv, bias)
	~~mnb1_3_2~~	(1, 28, 28, 240)	(5, 1, 1, conv, relu)	(1, 40, 1, conv, bias)
	~~mnb1_5_1~~	(1, 14, 14, 480)	(3, 1, 1, conv, relu)	(1, 112, 1, conv, bias)
	~~mnb1_5_2~~	(1, 14, 14, 672)	(3, 1, 1, conv, relu)	(1, 112, 1, conv, bias)
	~~mnb1_6_1~~	(1, 14, 14, 672)	(5, 1, 2, conv, relu)	(1, 160, 1, conv, bias)
	~~mnb1_6_2~~	(1, 7, 7, 960)	(5, 1, 1, conv, relu)	(1, 160, 1, conv, bias)
ResNet-18	res_2x	(1, 56, 56, 64)	(3, 64, 1, conv, relu)	(3, 64, 1, conv, bias)
	res_3x	(1, 28, 28, 128)	(3, 128, 1, conv, relu)	(3, 128, 1, conv, bias)
	~~res_4x~~	(1, 14, 14, 256)	(3, 256, 1, conv, relu)	(3, 256, 1, conv, bias)
	~~res_5x~~	(1, 7, 7, 512)	(3, 512, 1, conv, relu)	(3, 512, 1, conv, bias)
	res_2x_s2	(1, 56, 56, 64)	(3, 64, 2, conv, relu)	(3, 64, 1, conv, bias)
	res_3x_s2	(1, 28, 28, 128)	(3, 128, 2, conv, relu)	(3, 128, 1, conv, bias)
	~~res_4x_s2~~	(1, 14, 14, 256)	(3, 256, 2, conv, relu)	(3, 256, 1, conv, bias)
	~~res_5x_s2~~	(1, 7, 7, 512)	(3, 512, 2, conv, relu)	(3, 512, 1, conv, bias)
ResNet-50	res_2x_b_2	(1, 56, 56, 64)	(3, 64, 1, conv, relu)	(1, 256, 1, conv, relu)
	res_3x_b_2	(1, 28, 28, 128)	(3, 128, 1, conv, relu)	(1, 512, 1, conv, relu)
	~~res_4x_b_2~~	(1, 14, 14, 256)	(3, 256, 1, conv, relu)	(1, 1024, 1, conv, relu)
	~~res_5x_b_2~~	(1, 7, 7, 512)	(3, 512, 1, conv, relu)	(1, 2048, 1, conv, relu)