Holistic High Performance Convolution Algorithms on Modern CPUs*

Fuheng Wu† *Pingbo Zhang*‡ *Ivan Davchev*§

Abstract

Convolution, a fundamental operation in deep-learning neural networks, particularly in Convolutional Neural Networks (CNNs), is highly demanding in terms of computation. Consequently, it has been the subject of extensive investigation by both academia and industry for many years. The contribution of this paper is to provide the optimal register allocation and data reuse algorithms, along with convolution-friendly data layouts, for various types of convolutions on modern CPUs. These include standard convolution(SConv), depthwise convolution(DConv) and pointwise convolution(PConv), combined with essential convolution properties such as padding, stride, and dilation(PSD). Our proposed algorithms exhibit remarkable efficiency across different kernel sizes(from 1x1 to 51x51), data shapes and arbitrary PSDs on different hardwares(ARM, AMD and Intel). Our benchmarking demonstrates that the proposed methods can achieve a latency reduction of up to 10 times compared to the current state-of-the-art approaches. This paper also discusses discrepancy in results due to the difference of algorithms implemented accross different CPUs and the proposed holistic convolution algorithms can address the problem. The code accompanying this paper is open-sourced and available in the library wukong.

1 Introduction

Convolutional Neural Networks (CNNs) are a class of deep neural networks that are widely used in computer vision applications such as image and video recognition, object detection, and segmentation. At a high level, the numerical algorithm used in CNN computation involves performing a series of matrix multiplications or convolutions between the input data and learnable weights (aka filters or kernels) of the network. The output of each layer is then passed through an activation function and fed into the next layer. In all the layers, convolution layer is the most computationally intensive operations, so optimizing convolution is critical to improve inference performance. A significant restriction of CNN models is the considerable computational burden associated with consecutive convolution layers where high-end expensive GPU is unavailable. CPU serving is also attractive due to the high memory capacity, energy efficiency, high availability and affordable price. Even for most AI services providers, CPU serving is still a pragmatic solution due to the cost control pressure. In this paper, we only discuss convolution on CPUs unless it is specified otherwise.

1.1 SConv, DConv and PConv

Three types of convolutions are widely used in practice: standard convolution(SConv), depthwise convolution(DConv) and pointwise convolution(PConv).

Standard convolution(SConv), also known as traditional convolution, is a fundamental operation in deep neural networks (DNNs) commonly employed in computer vision tasks. It involves applying a convolutional filter, also referred to as a kernel or a weight matrix, to an input feature map(IFMap) to produce an output feature map(OFMap). This operation plays a crucial role in extracting local spatial patterns and learning hierarchical representations from the input data. In SConv, the filter slides over the input feature map, computing the dot product between its weights and the corresponding values in the receptive field. The receptive field refers to the local region in the input feature map that the filter is currently focused on. By scanning the entire input feature map, the convolutional filter captures different patterns and detects features of interest at various spatial locations. During the convolution operation, the filter's weights are multiplied element-wise with the corresponding values in the receptive field, and the resulting products are summed to obtain a single value in the output feature map. This process is repeated for each receptive field, producing the entire output feature map. The size and dimensions of the filter typically depend on the design of the convolutional layer. For example, a common choice is a square filter with a fixed kernel size, such as 3x3 or 5x5. However, the kernel size can vary depending on the specific architecture and requirements of the DNN model.

*This work is done at Oracle Cloud AI Services. Corresponding author: fuheng.wu@oracle.com. Mailing address: OCI Cloud AI Services, 100 Oracle Pkwy, Redwood City, CA 94065, USA.

Depthwise convolution(Dconv) is introduced to do convolution for large kernels and normally followed by pointwise convolution. It is a special grouped convolution(Krizhevsky et al., 2012), where the input channels and output channels are divided into groups, and each group has its set of kernels. The convolution operation is then performed independently within each group(Zhang et al., 2017). In depthwise convolution, the kernel number is 1 for each group, so that the input channel number equals to kernel batch number. We don't discuss other types of grouped convolution in this paper since they are rarely used in practice. Another similar concept is `Depthwise Separable Convolution`(DSC), which is basically a depthwise convolution(DConv) followed by a pointwise convolution(PConv). By separating the spatial and channel-wise operations, depthwise separable convolution significantly reduces the computational cost compared to SConv. Therefore, it is particularly useful when dealing with models that have limited computational resources or require real-time processing, such as on mobile devices or embedded systems.

Pointwise convolution(PConv) is a special standard convolution with kernel size 1x1. Initially, PConv was proposed in the Network-in-network paper. It was then highly used in the Google Inception paper and then becomes even more popular in MobileNet and Xception as the second step of `depthwise separable convolutions`. It is sometimes used independently without DConv like in VanillaNet(Chen et al., 2023a).

We will sometimes use `SConv`, `DConv` and `PConv` to represent them respectively.

1.2 PSD

There are three parameters used in convolution: padding, stride and dilation, which we denote them as `PSD` collectively. These parameters play a crucial role in determining the size, resolution, and receptive field of the output feature maps produced by the convolutional layers.

Padding refers to the process of adding extra elements (usually zeros) to the borders of the input feature map before performing the convolution operation. The main purpose of padding is to preserve the spatial dimensions of the input and ensure that the output feature map has the desired size. By padding the input, we can control the amount of spatial information retained during the convolution process.

Stride determines the step size at which the convolutional filter (also known as the kernel) moves across the input feature map. When applying convolution, the filter slides over the input with a certain stride value, and at each position, a dot product is computed between the filter weights and the corresponding receptive field in the input. The stride parameter controls the amount of spatial downsampling or upsampling that occurs during the convolution operation. A stride value greater than 1 leads to spatial downsampling, reducing the size of the output feature map. Conversely, a stride value of 1 preserves the spatial dimensions, producing an output with the same size as the input.

Dilation refers to the insertion of additional spaces between the elements of the convolutional filter, effectively increasing the receptive field of the filter without changing its parameters or size. This parameter allows for the integration of larger spatial contexts during convolution, which can be beneficial for capturing broader patterns or detecting objects at different scales. By adjusting the dilation, we can modify the effective receptive field of the filter. Larger dilation values result in a more expansive receptive field, while smaller dilation values limit the receptive field to a smaller region.

For 2D convolution, padding has default values of (0,0,0,0), while stride and dilation has (1,1). We call a convolution with default parameters `vanilla` convolution, otherwise it is an `exotic` convolution. Exotic convolution is widely used in practice. For example, semantic segmentation networks like DeepLab make extensive use of convolutions with non-default dilation.

1.3 Direct and Indirect Convolution Algorithms

As for convolution algorithms, there are two categories: direct and indirect algorithms. Direct convolution is to follow the definition of convolution to calculate dot product of input feature map and kernel. Indirect algorithms use lowering technique to convert the original problem to another problem and use matrix multiplication, winograd, or fast fourier transform to get the output feature map. Indirect convolution has a major drawback of big memory overhead and high CPU cache miss rate. The process involves packing overlapping image blocks, whose sizes match those of the kernel, into the columns of a sizeable temporary matrix. The extra packing operation and especially high CPU cache miss rate have been a big problem in many use cases, so that direct convolution is proposed as a more efficient algorithm.

1.4 Result Consistency

Accuracy discrepancy is a problem often ignored by practioners and researchers. Rounding error is a phenomenon of digital computing resulting from the com-

puter's inability to represent numbers exactly. Rounding error is prevalent in floating number calculation. For instance, using IEEE 754 standard's 32-bit single-precision floating-point format, the sum of $1.2 + 2.3 + 3.4$ is `4.77e-07` greater than $3.4 + 2.3 + 1.2$. When serving the same DNN model on different CPUs, we observed discrepancies in the results for the same input caused by different convolution algorithms implemented on different CPUs. When the model is relatively simple, the discrepancy is at 1e-6 or 1e-7 level so negligible. However, for complex models like composite model OCR, the rounding errors can accumulate to levels of 1e-2 or even 1e-1, resulting in noticeable differences in the results. To ensure consistent results across different CPUs, the exact same algorithm should be used for the same data input no matter what CPU it is running on. For example, when adding the same group of numbers, it is crucial to ensure that the order of the numbers being added remains unchanged. This may sacrifice some performance because, in terms of convolution algorithm, different platforms/CPUs have different THE best algorithm. It is necessary to strike the balance to find the best one for all the platforms, not for just one. In our use case, after unifying all the implementation with algorithms proposed by this paper, we achieved bit-wise identical results across all AMD, Intel and ARM CPUs.

Tab. 1: Notations in Convolution

var	description
N	batch of input/output
C	input channel number
H	input height
W	input width
B	input channel-wise block size
K	kernel batch size
C_k	kernel channel number
R	kernel height
L	kernel width
U	kernel batch-wise chunk size
C_o	output channel
H_o	output height
W_o	output width
B_o	output channel-wise block size
P_l, P_r, P_u, P_d	padding left/right/up/down to input
S_w, S_h	stride on width/height dimension
D_w, D_h	dilation on width/height dimension
DL, DR	dilated kernel width/height
G	groups of in/output channels
M	depthwise multiplier
ϵ	a learnable bias with size of C_o
T	floats per vector register

2 Conventions and Notations

We will discuss the most popular three types of convolutions: Sconv, Dconv and PConv. The convolution notations used in this paper are listed in table 1.

2.1 Standard and Pointwise Convolution

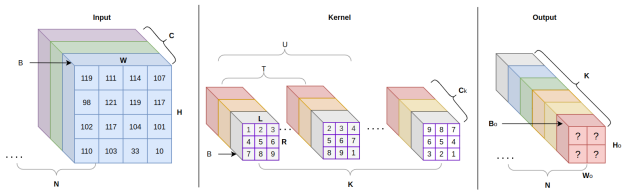

Fig. 1: Notations in Standard Convolution

Some of the notations do not exist in all the algorithms discussed in this paper. For example, B_o and U are meaningful in ZMC algorithm but not in indirect convolution. G is used in grouped convolution only. For SConv, C equals to $G \cdot C_k$. For DConv(Chollet, 2016), G equals to `C` and C_o equals to $K \cdot G$.

Please be noted that `N` is input batch, and also the output batch. We assume N equals to 1 in this paper, but the conclusion can be extended to other cases. G equals to 1 in SConv, so `K` is kernel batch number and also equal to output channel number C_o. Input channel number C and kernel channel number C_k are equal accordingly.

Figure 1 is a graphic view of three components in convolution: `input`, `kernel` and `output`. Sometimes, `kernel` is also called `filter` since its main purpose is to filter out features from input data.Considering their feature map property, terms `IFMap` and `OFMap` are also used to represent input and output feature maps respectively. We will use I, F and O to represent them respectively in our algorithm code later.

PConv is a special SConv where both R and L are 1, so the notation for PConv is the same as SConv.

When multiple notations are placed together without a dot between them, it implies they are multiplied with each other. For instance, $TUB = T \cdot U \cdot B$ and $SwB = Sw \cdot B$.

2.2 Depthwise Convolution

Figure 2 is a depthwise convolution. The input format is $NGHW$, kenrel format is $K1RL$ and output format is NKH_oW_o where $K = MG$.

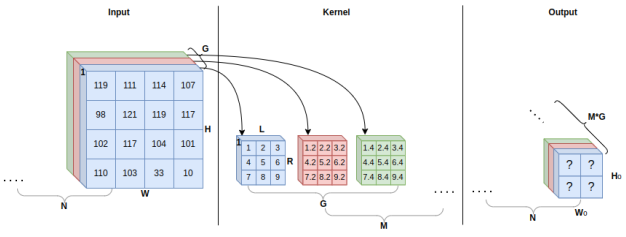

Fig. 2: Depthwise Convolution

DConv is a special grouped convolution. We assume M equals 1 when discussing depthwise convolution in this paper because in most cases M equals to 1 in the real world. When M is 1, the input format is $NGHW$, kernel format is $G1RL$ and output format is NGH_oW_o. We only discuss single precision floating number convolution but the results can be expanded to other precision. For example, in our discussion, one AVX2's YMM register can hold 8 numbers because one YMM register is 256-bit long and one single precision floating number has 32 bits.

2.3 Data Format

During the convolution operation, the input data is usually represented as a tensor. There are two common data formats for representing tensors in convolution: `NCHW` and `NHWC`. The meaning of the notation can be referred to table 1. `NCHW` is commonly used in deep learning frameworks such as PyTorch, Caffe and ONNX. `NHWC` is commonly used in deep learning frameworks such as TensorFlow and Keras. The choice of input data format can have big impact on the performance of the convolution operation. On CPUs, both of them incur low performance due to high CPU cache miss. $NC'HWC_B$ was proposed as a CPU-cache-friendly data format(Zhang et al., 2018, Liu et al. (2019)). It adds an innermost dimension C_B for better cache locality by splitting the original outer channel C to $C' = \lceil \frac{C}{B} \rceil$ where B is input channel-wise block size as described in table 1. $NC'HWC_B$ format has been widely adopted by mainstream inference engines (Intel OneDNN, Amazon NeoCPU, Microsoft ONNX Runtime). For example, Microsoft ONNX Runtime use $NC'HWC_8$ in its AVX2 implementation.

In this paper, we discuss convolution in CPUs and compare with the state-of-arts method which also use `NCHW` format in the original input, so we assume the input data format is `NCHW` in this paper.

It is a trade-off because larger B reduces the number of storing output from SIMD register back to RAM but incurs higher L1/2 cache load miss rate. In practice, C is a multiplier of 32 in most cases(He et al., 2015) and modern CPUs have L1 cache line 64 bytes as the standard. Our benchmarking shows ZMC(Zhang et al., 2018) and our algorithms consistently reach the best performance when B is 16 for SConv and Dconv, and 32 for PConv.

2.4 Kernel Size

The use of 3x3 kernels as a standard in SConv has become widely adopted, as evidenced by its inclusion in both traditional models such as VGG(Simonyan and Zisserman, 2014) and contemporary cutting-edge models such as Stable Diffusion(Chen et al., 2023b). This convention has emerged as a result of a combination of factors, including the computational efficiency of 3x3 kernels, their hierarchical structure and efficacy in promoting translation invariance. (Camgözlü and Kutlu, 2021)

As for DConv, 3x3 kernels are still popular(Howard et al., 2017, Sandler et al. (2018)) but large kernels like 5x5, 7x7(Szegedy et al., 2016, Yu et al. (2023)) are also widely used due to DConv's computation efficiency. After transformer models become popular and surpass traditional small-kernel computer vision models, researchers tries to increase receptive field to improve the model performance, therefore larger kernels such as 31x31(Ding et al., 2022), 51x51(Liu et al., 2023) have been more and more used. To our best knowledge, our paper is the first one discussing algorithms which can efficiently process large-kernel convolution on CPUs.

2.5 Register Allocation Schemes

Registers are small, high-speed storage locations within the CPU that store data for immediate processing. They are limited in number but can be quickly accessed by the processor. Vector registers, also known as SIMD (Single Instruction Multiple Data) registers, are specialized registers found in modern CPUs that are designed to store and operate on multiple data elements simultaneously. These registers are wider than the typical scalar registers, allowing them to hold multiple data elements of the same type in a single register. A good `register allocation scheme` is critical to fully leverage the power of vector registers to speed up the computation. On CPUs, Fused-Multiply-Add(FMA) is heavily used to speed up convolution algorithms. FMA is an instruction set extension that combines a multiplication and an addition operation into a single instruction. This allows for more efficient execution of mathematical operations, especially in applications that heavily rely on floating-point calculations. By performing both operations in a single

step, FMA can reduce latency and increase throughput, improving the overall performance of mathematical computations. Figure 3 showcases the register allocation in convolution. We call *broadcast* on Intel/AMD or *load* on ARM to load IFMap data from RAM to the red color registers. Similarly, kernel and OFMap data are loaded into the yellow and green registers respectively by calling *load* instruction. Then we call *fma* instruction to multiply V_i and V_k and sum the result with V_o, and finally write it back to OFMap in RAM. If there are p red color IFMap registers and q green color OFMap registers, we denote the register allocation scheme as $IpOq$. The remaining registers are normally used as temporary variables. In different use cases, the best register allocation scheme on the same CPU could be different. For instance, on AVX2, $I3O12$ is better than $I5O10$ for small kernels but $I5O10$ could outperform $I3O12$ for big kernels.

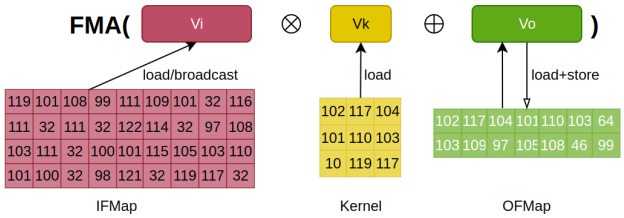

Fig. 3: $IpOq$ Register Allocation on CPUs

2.6 Related Work

SConv typically costs more than 70% computation resources in a DNN network. There is a long history of research and development on how to do SConv efficiently. Back in 1960s, indirect Fast Fourier Transform (FFT) algorithm was proposed to do SConv but it works better for large kernel convolution only. Winograd algorithm was proposed by Shmuel on 1976 to reduce computation in matrix multiplication. In 1990s, Yann Lecun etc proposed Im2Col+GEMM algorithm and Yangqiang Jia made the first code commit in Caffe library in 2014. Google's Tensorflow implemented Winograd algorithm and had the first code checkin to Tensorflow in 2015. Intel MKL-DNN(now OneDNN) and Google Tensorflow started to find $NC'HWC_B$ is good for better memory access on CPU. On late 2018, ZMC(Zhang et al., 2018) officially proposed an efficient direct convolution algorithm with $NC'HWC_B$ as a more convolution-friendly input data format. ZMC algorithm is described as a 9-layer nested loop in order $K'C'HW'RLC_BW_xK_{TU}$ but the innermost 5 loops can be easily unrolled. Almost at the same time or even earlier, Amazon NeoCPU(Liu et al., 2019) independently provided another similar direct algorithm with different loop order and graph level optimization considered. We use ZMC and AMZ to refer them respectively. Compared with ZMC, the only difference of AMZ is it has the blocked input channel C' inside loop W'. This reduces the times of writing back data from vector registers to RAM but increases data cache load and miss rate due to more non-continuous memory access. We will analyze the performance of ZMC and AMZ later. Unfortunately AMZ/NeoCPU(Liu et al., 2019) is not open-sourced, so we will provide the implementation in library wukong.

Many research-oriented algorithms have also been proposed including indirect GEMM-based algorithms such as (Dukhan, 2019), (Meng et al., 2022), MEC(Cho and Brand, 2017) and ECBC(Zhao et al., 2023), and direct convolution algorithms such as (Ofir and Ben-Artzi, 2022). (Ofir and Ben-Artzi, 2022) is super simple to implement. It only performs well for small channel input, but in practice the channel number is normally up to several hundreds. All the indirect convolution algorithms suffers from low performance due to runtime packing, high cache miss and inefficient vector register usage. ECBC(Zhao et al., 2023) overlooks the fact that Amazon NeoCPU(Liu et al., 2019) has optimized ZMC algorithm and is unaware that Microsoft ONNX Runtime has an opensource implementation of it. In some edge cases, indirect convolution algorithms could performs better than SoTA, but the performance degrades dramatically when input data size/channel and kernel batch number increase.

For DConv, previous studies(Zhang et al., 2020, Hao et al. (2022)) focus on small kernels, such as 3x3 and 5x5, on ARM CPU. (Zhang et al., 2020) proposes a direct convolution algorithm for DConv on ARM CPU. It doesn't work for large kernels because it loads all the kernel data into vector registers, and the algorithm is still suboptimal due to repeatedly data loading from memory to vector registers. (Hao et al., 2022) proposes another small-kernel direct convolution algorithm trying to maximize data reuse in vector register. In addition to suffering from high CPU cache miss rate, it is inefficient for exotic convolution with irregular padding, stride. The padding strategy wastes vector register by putting zeros in it. To handle irregular stride, different tile size of computing kernel need to be used. To our best knowledge, all existing DConv algorithms described in those papers are unable to process large kernels, inefficient to handle exotic convolution, and overlooks dilation or stride.

PConv, a special SConv with 1x1 kernel, often follows DConv in a real DNN as the second part of a depthwise separable convolution. (Zhang et al., 2020) discuss an algorithm for PConv using GEMM but no other literature covers it in direct convolution context. The

1x1 kernel actually has impact on algorithm design and we will cover it in this paper.

3 Our Approach

Since convolution is applied in a sliding window way, we propose to use a sliding window technique with direct convolution algorithm to maximize the data reuse. When sliding, we hold the input data inside vector register as long as possible. We primarily slide in width W instead of height H dimension because the memory is continuous in width dimension and our vector register number is limited. When kernel is small and vector register number is large like in case of 3x3 kernel Dconv in ARM Neon, we slide in height H dimensions too. The sliding technique works for both Sconv and Dconv, vanilla and exotic convolutions as long as Dw equals to Sw. Figure 4 describe how it works for SConv in ARM.

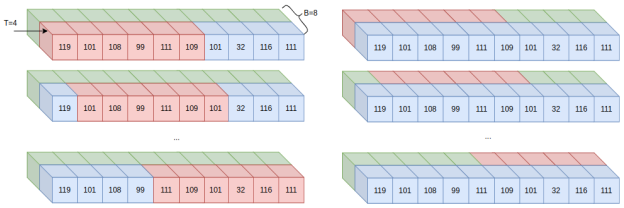

Fig. 4: Sliding Window in Convolution in ARM Neon

For PConv, sliding can not be used because window size is only 1. However, this brings another opportunity of optimization because R and L dimension are actually taken away from the nested loop. In Intel CPU, we recommend AMZ over ZMC, while in AMD and ARM, ZMC is preferred.

We will provide the details in the following sections. We also provide specialized version for regular SConv and DConv for best performance. Finally we will show the performance benchmarking results of our algorithm compared with mainstream DNN software including ONNX Runtimne, PyTorch, Tensorflow 2.0, TVM and OneDNN.

We discuss 2D convolution but the algorithm can be easily extended to 1D and 3D. For example, when H is 1, it is reduced to 1-D convolution. Our algorithm relies on vector registers and the utilization of the Fused-Multiply-Add (FMA) operation. To ensure clarity in our exposition, we employ the term `AVX2` to collectively refer to AVX1+FMA, AVX2, and AVX512. These technologies are part of the AVX family, offering support for vector registers and FMA operations. AVX512 stands out from AVX1 and AVX2 by doubling the number and length of vector registers. Notably, the algorithm designed for AVX2 can be straightforwardly extended to AVX512 with minimal modifications.

3.1 SConv

As for SConv, we observed that ZMC(Zhang et al., 2018) outperforms AMZ(Liu et al., 2019) and other algorithms due to its efficient cache load and reduced miss rate. However, both of them still face the issue of repeatedly loading data into register during different loop iterations. To address this concern, we propose to move L into C_B in the nested loop order to reuse the register data. Additionally, we set the value of channel block size B as 16 to align with the cache line size of 64 bytes. The IFMap data layout is $NC'HWC_{16}$, while kernel layout is $K'C'RLC_{16}K_{TU}$. Due to the difference between SIMD instruction sets, more specifically Intel/AMD's AVX2 and ARM's Neon, there is a slight difference in the algorithms.

3.1.1 AVX2

Because the memory is contiguous in W dimension, sliding operations in this dimension have minimal impact on cache misses. For kernel size less than 7x7, ZMC algorithm exhibits the lowest latency. However, for larger kernels, utilizing sliding techniques can significantly accelerate the convolution process. For register allocation, there are two schemes. The first schemes, denoted as $I3O12$, is to allocate 3 registers to IFMap, and 12 registers to OFMap data so that we can process 32 batches of kernel data in one loop($U = 4$). The second scheme, referred to as $I5O10$, allocates 5 registers for IFMap and 10 for OFMap, therefore 16 batches of kernel data can be processed in one loop($U = 2$). The remaining one register is used as a temporary variable to load kernel data.

When sliding kernel above IFMap data in exotic convolution, the kernel and IFMap data may not be fully overlapped because of padding, and the data used for calculation is non-continuous in memory due to stride and dilation. We call the overlapped part of kernel `valid kernel` and overlapped IFMap data `valid data`, otherwise it is labeled as `invalid`. Sliding technique can be employed when the W dimension's stride and dilation are identical. For padding, *NCNN* and *FeatherCNN* use a temporary space to explicitly pad input data which takes much latency and memory overhead. Other software adopt implicit padding by setting values in register as 0, which leads to inefficient utilization of computational resources. In our approach, we consider the first element of the kernel as the origin in the coordinate system. The relative

coordinate of the first item in IFMap is (x, y). When moving the kernel from left to right, the value of x starts at P_l and decreases until $DL - W - P_r$. Similarly, when moving from top to bottom, y, starting at P_u, decreases until $DR - H - P_d$. Our approach is more efficient because it only involves the necessary kernel/IFMap data required for convolution.

Algorithm 1 SConvRegisterSliding

Parameters: I, K, S_wB, L, reg, o, U
for `i:=1 →reg-1` **do** ▷ macro
  $v_i = broadcast(I + (i-1) \cdot S_wB)$
end for
for $\underline{x} := 0 \rightarrow L - reg$ by reg **do**
  for $y := 0 \rightarrow reg - 1$ **do** ▷ macro
    $v_{(reg-1+y)\%reg+1} =$
      $broadcast(I + S_wB \cdot (reg - 1 + y + \underline{x}))$
    for $u := 0 \rightarrow U - 1$ **do** ▷ macro
      $v_0 = vload(K + T \cdot (u + \underline{x} \cdot \underline{U}))$
      for $z := 0 \rightarrow reg - 1$ **do** ▷ macro
        $v_{o+z+u*reg} =$
          $fma(v_0, v_{y\%reg+z+1}, v_{o+z})$
      end for
    end for
  end for
end for
$m \leftarrow L\%reg$, $n \leftarrow L - m$
for $j := 0 \rightarrow reg - 1$ **do** ▷ macro
  if $m > j$ **then**
    $v_{(reg-1+j)\%reg+1} =$
      $vload(I + S_wB \cdot (n + reg - 1 + j))$
    for $u := 0 \rightarrow U - 1$ **do** ▷ macro
      $v_0 = vload(K + T \cdot (n \cdot U + j \cdot U + u))$
      for $z := 0 \rightarrow reg - 1$ **do** ▷ macro
        $v_{o+z} = fma(v_0, v_{j\%reg+z+1}, v_{o+z})$
      end for
    end for
  end if
end for

In H loop, we put the logic inside a big loop, but for W loop, it is more complicated because we will do register blocking and sliding so there will be many branches and if conditions. To improve the performance, our idea is to unroll the W loop to three loops where we can use `register blocking` and `register sliding` in the second loop. The first and third loop cannot use register blocking and sliding because, when the kernel moves to the right, each valid kernel and IFMap size are different due to the existence of padding. Take a size 3x3, stride 2x2 and dilation 2x2 SConv as an example. Figure 5 represents the first loop's scenario where some kernel data exist in left padding area. In W direction, the first loop invariant is $x > 0$.

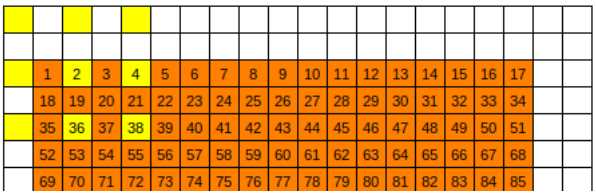

Fig. 5: Exotic SConv First Loop

The key point is to get the first valid index used in computation in W dimension for kernel and IFMap, and then we can use it as a base to move to the right. In algorithm 2, kw_i and iw_i are used to represent them. In figure 5, the first valid index in both kernel(kw_i) and IFMap(iw_i) is 1 with values 4 for kernel and 2 for IFMap respectively. With these information, we can eaily loop through in Kernel. We also pre-calculate the distance between the last element in one row and the first element in the next row for both kernel and IFMap, represented as i_delta_p and k_delta_p respectively. When looping through the kernel, we need them to find the next element quickly. r_{begin} and r_{end} are for looping in height dimension, and also pre-calculated in the upper loop since they don't change inside the W dimension loop. We don't use register blocking and sliding in algorithm 2.

Algorithm 2 SConvPSDLeft

for $x := P_l \rightarrow 0$ by $-S_w$ **do**
  $kw_i \leftarrow 1 + (x-1)/D_w$
  $l_{begin} \leftarrow kw_i \cdot D_w$
  $i\_delta\_p \leftarrow i\_row\_delta + kw_i \cdot D_wB$
  $iw_i \leftarrow l_{begin} - x$
  $l_{end} \leftarrow min(DL, W + x)$
  $k\_delta\_p \leftarrow TUB \cdot kw_i$
  $i \leftarrow i_{base} + iw_i \cdot B$
  $k \leftarrow k_{base} + k\_delta\_p$
  for $r := r_{begin} \rightarrow r_{end}$ by D_h **do**
    for l:=$l_{begin} \rightarrow l_{end}$ by D_w **do**
      $sconv\_comp\_reg\_1(i, k)$
      $i \leftarrow i + D_wB$; $k \leftarrow k + TUB$
    end for
    $i \leftarrow i + i\_delta\_p$
    $k \leftarrow k + k\_delta\_p$
  end for
  $sconv\_store\_reg\_1(output, obs)$
  $output \leftarrow output + B$
end for

The second loop invariant is $x \geq DL - W$. There is no kernel data falling in either left or right padding areas as showcased in figure 6. The algorithm $SConvPSDMiddle$ is described at algorithm 3. We use register blocking and sliding with reg_n registers. If

we use $I3O12$ register allocation scheme, reg_n equals to 3.

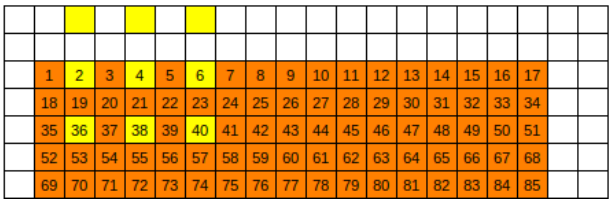

Fig. 6: Exotic SConv Second Loop

Algorithm 3 SConvPSDMiddle

if $x \geq DL - W$ **then**
  $wcount \leftarrow (x - DL + W)/S_w + 1$
  $rep, remain \leftarrow wcount \mod reg\_n$
  $i_{begin} \leftarrow i_{base} - x \cdot B$
  $x \leftarrow x - wcount \cdot Sw \cdot reg\_n$
  while $rep \neq 0$ **do**
    $i \leftarrow i_{begin}$; $k \leftarrow k_{begin}$
    for r:=$r_{begin} \rightarrow r_{end}$ by D_h **do**
      for $l := 0 \rightarrow L$ **do** ▷ unroll[opt]
        $sconv\_comp\_reg\_n(i, k)$
        $i \leftarrow i + D_w B$; $k \leftarrow k + TUB$
      end for
      $i \leftarrow i + i\_row\_delta$
    end for
    $sconv\_store\_reg\_n(output, obs)$
    $i_{begin} \leftarrow i_{begin} + SwB \cdot reg\_n$
    $output \leftarrow output + reg\_n \cdot B$
    $rep \leftarrow rep - 1$
  end while
end if

We adopt register blocking and optionally sliding 1 in $SConvPSDMiddle$ 3. For $I3O12$, we need to handle IFMap data with W value of not a multiplier of 3, such as 31, 32. $remain$ is used to process this kind of boundary condition where less than reg_n registers are used for IFMap data in register blocking. We can add a $switch - case$ code block after the while loop to process each case. The code is very similar to $SConvPSDMiddle$ so we don't write it down here.

Figure 7 showcases the third unrolled loop where some kernel data exist in right padding area, and the loop invariant is $x \geq DL - W - P_r$. The algorithm $SConvPSDRight$ is described at algorithm 4.

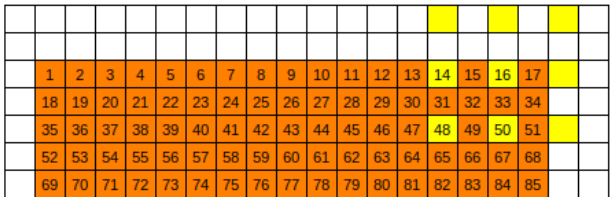

Fig. 7: Exotic SConv Third Loop

We can see the code structure is almost the same as $SConvPSDLeft$ except that loop invariant, l_{begin}, kw_i and i_delta_p are different. We can wrap the code in a function to make it simpler, but we observed the latency actually increased due to the function invocation overhead.

Algorithm 4 SConvPSDRight

while $x \geq DL - W - P_r$ **do**
  $kw_i \leftarrow ceil\_int(DL - W - x, D_w)$
  $i\_delta\_p \leftarrow i\_row\_delta + kw_i \cdot D_w B$
  $l_{end} \leftarrow min(DL, W + x)$
  $k\_delta\_p \leftarrow TUB \cdot kw_i$
  $i \leftarrow i_{base} - x \cdot B$
  $k \leftarrow k_{base}$
  for $r := r_{begin} \rightarrow r_{end}$ by D_h **do**
    for l:=0 $\rightarrow l_{end}$ by D_w **do**
      $sconv\_comp\_reg\_1(i, k)$
      $i \leftarrow i + D_w B$; $k \leftarrow k + TUB$
    end for
    $i \leftarrow i + i\_delta\_p$
    $k \leftarrow k + k\_delta\_p$
  end for
  $sconv\_store\_reg\_1(output, obs)$
  $output \leftarrow output + B$
  $x \leftarrow x - S_w$
end while

Finally, the full algorithm is listed in algorithm 5.

Algorithm 5 SConvPSD

for `k':=0→K';c:=0→C';` **do**
  for $y \leftarrow P_u$ to $DR - H - P_d$ by $-S_h$ **do**
    $kh_i \leftarrow max(ceil\_int(y, D_h), 0)$
    $r_{begin} \leftarrow kh_i \cdot D_h$
    $r_{end} \leftarrow min(DR, H + y)$
    $ih_i \leftarrow r_{begin} - y$
    $k_{base} \leftarrow kernel + TUBL \cdot kh_i$
    $i_{base} \leftarrow input + W \cdot B \cdot ih_i$
    call $SConvPSDLeft$
    call $SConvPSDMiddle$
    call $SConvPSDRight$
  end for
end for

3.1.2 ARM Neon

For Neon, compared with AVX2, the number of vector registers are doubled to 32 but bit size of each register is halved to 128, so it can hold 4 single floats only. Furthermore, there is no broadcast instruction in Neon. Broadcasting is implicitly done when FMA instruction is invoked. We just need to load data into register, and then specify the index of the data we want to broadcast in the register and call FMA directly with the index. This is more efficient since 1 register load in Neon is equivalent to 4 register broadcasts in AVX2. So we have the following for SConv in ARM Neon:

- Input Layout: $NC'HWC_{16}$
- Kernel Layout: $K'C'RLC_{16}K_{TU}$
- Loop Order: $K'C'HW'RC_{4\times\underline{4}}LW_{\hat{x}}K_{TU}$ where $C' = C/B, K' = K/(TU), W' = W/\hat{x}, B = 16$

In the loop order, different from the AVX2 version, we use $C_{4\times\underline{4}}$ to mean the 16 channels are processed by two nested loop where the inner loop is naturally unrolled because one vector register load can take 4 floats in channel dimension.

We also observed that register sliding *always* speeds up convolution no matter the kernel size on ARM. For instance, it boost the 3x3 kernel SConv speed by 25% in our benchmarking. This is because sliding with more registers has bigger positive impact on the latency than the negative impact caused by cache miss and it becomes the dominant factor to boost the performance. For register allocation, like AVX2, there are two comparable options: $I6O24$(U=4) and $I10O20$(U=2). $I3O27$(U=9) and $I5O25$(U=5) are also feasible theoretically, but we don't use them because most models in practice have kernel batch size as multiplier of 16 or 8. With register sliding, the latter have more computation efficiency because sliding with 10 registers saves more register data loading than that for 6 registers. The downside of the latter is it needs more data write-back from register to RAM because it only processes 8 batches of kernel in one loop, while the former processes 16 batches. Also the former is easier to code because it has less edge cases to consider. When W is not a multiplier of 6, the remainder could be any number from 1 to 5. Therefore, we need to write function or macro to process the 5 edge cases. With $I10O20$ scheme, the edge case number is 10. We use register allocation scheme $I6O24$ below due to its simplicity for boundary condition processing. The register allocation and loop order are showcased in figure 8. In figure 9, we slide register to the right by 1. The algorithm will be very similar to the AVX2 version so we don't repeat it here. There are some programming tips we can use in ARM Neon like preferring SIMD version instruction such as LD1 instead of regular LDR but we don't dive deep in this paper. The details can be found in our open source implementation library wukong.

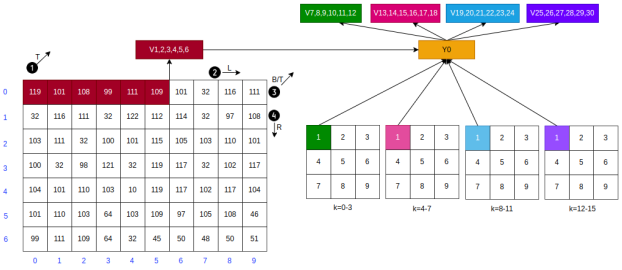

Fig. 8: I6O24 ARM Neon Step 1, $K'C'HW'RC_{4\times\underline{4}}L$

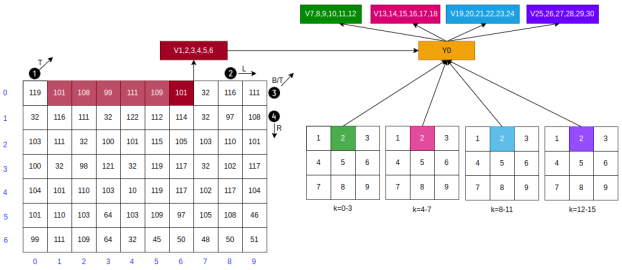

Fig. 9: I6O24 ARM Neon Step 2, $K'C'HW'RC_{4\times\underline{4}}L$

The computation efficiency metric is $24/(5+\frac{5}{L})$ with a limit value of 4.8.

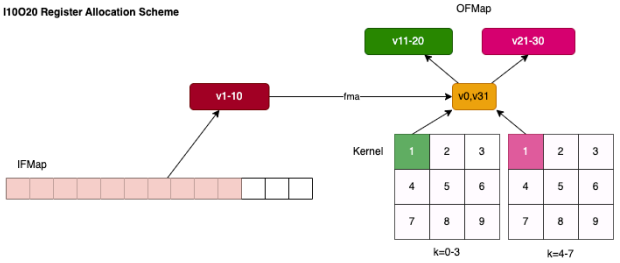

Fig. 10: I10O20 ARM Neon

3.2 DConv

We discuss efficient DConv algorithms in this section with $NC'HWC_x$ IFMap layout. With the best register allocation scheme and various optimization techniques such as register blocking and sliding, we get up to 10 times speed up compared with existing best inference engine in the market. Our algorithm is able to handle large kernels and outperforms other alternatives even more with large kernels.

In SConv, we broadcast IFMap in a specific channel and reuse the data by calling FMA with kernel

in the same channel but different batches. The data reuse is specified by U in our notation. In DConv, this kind of data reuse doesn't exist any more because the IFMap channel and kernel batch is 1-to-1 relation now. We still use register blocking but it doesn't provide that much benefit in DConv like in SConv. However, `register sliding` still boost the performance for DConv, and it is even more efficient since we get more registers for IFMap.

Different from SConv, the reordered kernel data layout is $G'1RLG_B$, which is essentially the same as IFMap's $NG'HWG_B$. The loop order is $G'ZHWRL$ where $Z = B/T$.

When exotic convolution is considered, the algorithm for SCovn still applies to DConv, except that the IFMap channel and kernel batch dimension are taken away in DConv.

3.2.1 AVX2

With 16 vector registers in AVX2 and sliding in W dimension, we can use 7 register for IFMap and 7 for kernel, which is represented as $I7O7$. We can also slide in both width and height dimensions, such as $I5O10$. For 3x3 kernel, we can even use $I1O9$ where 1, 6 and 9 registers are used for IFMap, kernel, and OFMap. We propose algorithm 6 *SConvRegisterSliding* to apply register sliding technique in W direction for `arbitrary` size of kernel.

L is the valid kernel width. For dilated kernel, L equals to dilated kernel width DL divided by D_w. v_i is a vector register, which could be $YMMi$ or $ZMMi$ for AVX, or V_i for Neon. reg is the number of registers used in the sliding with constraint of $count > reg$. o is the output register base number. For example, in AVX2 $I7O7$ scheme, o equal to 8 means 7 registers(YMM8-YMMM14) are allocated as OFMap registers. The IFMap register base number is fixed to 1 by default.

Please be noted that algorithm 6 is a macro instead of a regular function. It generates high performance SIMD source code repeatedly in a for loop manner. All the for loops with `macro` comment are more like `.rep` macro directive in GAS (GNU Assembler), which is used to repeat instructions multiple times primarily for loop unrolling. Thus only the second for loop will appear in source code after code generation. At the time of this writing, macro in C++ and NASM doesn't provide loop feature. GNU AS has loop macro `.rep` but doesn't support math operator like modulo, which is required by algorithm *DConvRegisterSliding* to generate variable names. For small loop, we can hard-code it. But for bigger one, we use a Python script to generate assembly code.

Algorithm 6 DConvRegisterSliding

Parameters: I, K, S_wB, L, reg, o
for `i:=1 →reg-1` **do** ▷ macro
  $v_i = vload(I + (i-1) \cdot S_wB)$
end for
for $\underline{x} := 0 \rightarrow L - reg$ by reg **do**
  for `y:=0 →reg-1` **do** ▷ macro
    $v_{(reg-1+y)\%reg+1} =$
      $vload(I + S_wB \cdot (reg - 1 + y + \underline{x}))$
    $v_0 = vload(K + T \cdot (y + \underline{x}))$
    for `z:=0 →reg-1` **do** ▷ macro
      $v_{o+z} = FMA(v_0, v_{y\%reg+z+1}, v_{o+z})$
    end for
  end for
end for
$m \leftarrow L\%reg$, $n \leftarrow L - m$
for `j:=0 →reg-1` **do** ▷ macro
  if $m > j$ **then**
    $v_{(reg-1+j)\%reg+1} =$
      $vload(I + S_wB \cdot (reg - 1 + n + j))$
    $v_0 = vload(K + T \cdot (j + n))$
    for `z:=0 →reg-1` **do** ▷ macro
      $v_{o+z} = FMA(v_0, v_{y\%reg+z+1}, v_{o+z})$
    end for
  end if
end for

3.2.2 ARM Neon

In ARM Neon, algorithm 6 *DConvRegisterSliding* still works for W dimension sliding. But in Neon, we have much more registers so we can slide in both height and width dimensions to get more performance boost. In our benchmarking, $I10O20$ is much faster than $I15O15$. This is because we use larger register tiles and update the tiles by register sliding, instead of data re-loading. Other choices such as $I6O24$, $I5O25$ will work well too, but it is very complicated to process boundary conditions. $I6O24$ has 23 edge casesm and $I5O25$ has 24 edge cases. With $I10O20$, we already got very good result so we didn't try other schemes. The code structure is the same as the AVX2 version with slight difference in register broadcasting as mentioned above.

3.3 PConv

Compared with DConv, PConv occupies a larger share of total runtime in large networks because PConv's time complexity is $O(C^2)$ while complexity of most other neural network operators, including DConv, is linear $O(C)$. Compared with SConv, it is more

lightweight and mergerable(Chen et al., 2023a). Notably, the best algorithm for PConv is related to CPU type and even input size because of the difference in CPU cache design. Generally speaking, AMZ(Liu et al., 2019) is faster than ZMC(Zhang et al., 2018) in Intel CPUs, but slower than ZMC in AMD and ARM CPUs. If unifying the algorithm in different CPUs for absolute idential results is required, ZMC algorithm with channel blocking size 16 is a good choice.

In $NC'HWC_B$ format, the latency is very sensitive to B. This is because, in ZMC algorithm, larger B means less write-back of register data to RAM. When B equals to 16, the latency can be 20-100% less than that when B equals to 8 in our benchmarking.

Noteworthy, PConv is also very friendly to GEMM, because *im2col* operation becomes a simple reshape operation since there are no need for sliding kernels with overlap.

PConv doesn't have padding and stride in most cases. Dilation doesn't exist for PConv either. Register blocking and tiling can boost performance but sliding doesn't work any more. The Sconv algorithm still works for PConv. It is simpler since R, L dimension and exotic convolution properties are taken away.

4 Experiments

4.1 CPU

- Intel CoffeeLake, HasWell, Skylake
- AMD Zen 2, Zen 3
- ARMv8 Neoverse N1

4.2 Language

Most of the wukong library is written with assembly language. We observed C++ compilers gave very inconsistent and sub-optimal binary code when it comes to dealing with CPU SIMD register allocation. Sometimes, for the same algorithm, we observed the same implementation in assembly was 4 times faster than that in C++. We tested g++11 ~ 13, clang++ 14 and 15. The newer version are normally better but they still emits different code with inconsistent performance in different OS/CPUs. Therefore we adopt assembly as the main language.

- ASM: The convolution core algorithms are implemented with assembly language, more specifically `GNU AS` for ARM, and `NASM` for Intel and AMD.
- C++: C++ is used for higher level interface of the library
- Python: We generated macro with Python script.

4.3 Compiler

- ASM: GAS 2.35, NASM 2.10.07
- C++: g++ 12+
- Python: 3.8+

Our algorithm requires SIMD instruction set, more specifically at least AVX 1.0 and FMA in Intel and AMD CPUs, and Neon in ARM CPU.

5 Conclusion

This paper presents a holistic direct convolution algorithm that is optimized for modern CPUs. It outperforms the existing algorithms by fully leveraging hardware acceleration features and optimizing cache load and vector register allocation. With proper implementation, it is up to 10 times faster than existing solutions in some cases such as large kernel and exotic convolutions, which was poorly handled by other frameworks. We adopted it in our production and it has been working stably for more than one year. Compared with the existing best open-source solution in the market and combined with other system-level hacks and optimization, it reduced the latency to 4 times less while improved the throughput by 3 times, and reduced memory usage by 90%. It can be used in inference engine in servers with modern CPUs. We believe the same optimization can be applied to other environments like mobile or IOT devices.

References

Yunus Camgözlü and Yakup Kutlu. Analysis of filter size effect in deep learning. *CoRR*, abs/2101.01115, 2021. URL https://arxiv.org/abs/2101.01115.

Hanting Chen, Yunhe Wang, Jianyuan Guo, and Dacheng Tao. Vanillanet: the power of minimalism in deep learning, 2023a.

Yu-Hui Chen, Raman Sarokin, Juhyun Lee, Jiuqiang Tang, Chuo-Ling Chang, Andrei Kulik, and Matthias Grundmann. Speed is all you need: On-device acceleration of large diffusion models via gpu-aware optimizations, 2023b.

Minsik Cho and Daniel Brand. MEC: memory-efficient convolution for deep neural network. *CoRR*, abs/1706.06873, 2017. URL http://arxiv.org/abs/1706.06873.

François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016. URL http://arxiv.org/abs/1610.02357.

Xiaohan Ding, Xiangyu Zhang, Yizhuang Zhou, Jungong Han, Guiguang Ding, and Jian Sun. Scaling

up your kernels to 31x31: Revisiting large kernel design in cnns, 2022.

Marat Dukhan. The indirect convolution algorithm. *CoRR*, abs/1907.02129, 2019. URL http://arxiv.org/abs/1907.02129.

Ruochen Hao, Qinglin Wang, Shangfei Yin, Tianyang Zhou, Siqi Shen, Songzhu Mei, and Jie Liu. Towards effective depthwise convolutions on armv8 architecture, 2022.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL http://arxiv.org/abs/1704.04861.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

Shiwei Liu, Tianlong Chen, Xiaohan Chen, Xuxi Chen, Qiao Xiao, Boqian Wu, Tommi Kärkkäinen, Mykola Pechenizkiy, Decebal Mocanu, and Zhangyang Wang. More convnets in the 2020s: Scaling up kernels beyond 51x51 using sparsity, 2023.

Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/liu-yizhi.

J. Meng, C. Zhuang, P. Chen, M. Wahib, B. Schmidt, X. Wang, H. Lan, D. Wu, M. Deng, Y. Wei, and S. Feng. Automatic generation of high-performance convolution kernels on arm cpus for deep learning. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2885–2899, nov 2022. ISSN 1558-2183. doi: 10.1109/TPDS.2022.3146257.

Amir Ofir and Gil Ben-Artzi. Smm-conv: Scalar matrix multiplication with zero packing for accelerated convolution. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 3066–3074, 2022. doi: 10.1109/CVPRW56347.2022.00346.

Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018. URL http://arxiv.org/abs/1801.04381.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016. URL http://arxiv.org/abs/1602.07261.

Weihao Yu, Pan Zhou, Shuicheng Yan, and Xinchao Wang. Inceptionnext: When inception meets convnext, 2023.

Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High performance zero-memory overhead direct convolutions. *CoRR*, abs/1809.10170, 2018. URL http://arxiv.org/abs/1809.10170.

Pengfei Zhang, Eric Lo, and Baotong Lu. High performance depthwise and pointwise convolutions on mobile devices. *CoRR*, abs/2001.02504, 2020. URL http://arxiv.org/abs/2001.02504.

Ting Zhang, Guo-Jun Qi, Bin Xiao, and Jingdong Wang. Interleaved group convolutions for deep neural networks. *CoRR*, abs/1707.02725, 2017. URL http://arxiv.org/abs/1707.02725.

Tianli Zhao, Qinghao Hu, Xiangyu He, Weixiang Xu, Jiaxing Wang, Cong Leng, and Jian Cheng. Ecbc: Efficient convolution via blocked columnizing. *IEEE Transactions on Neural Networks and Learning Systems*, 34(1):433–445, 2023. doi: 10.1109/TNNLS.2021.3095276.