PERFORMANCE PRIMITIVES FOR ARTIFICIAL NEURAL NETWORKS

A Thesis
Presented to
The Academic Faculty

by

Marat Dukhan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology
May 2021

PERFORMANCE PRIMITIVES FOR ARTIFICIAL NEURAL NETWORKS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Optimized software implementations of artificial neural networks leverage primitives from performance libraries, such as the BLAS. However, these primitives were prototyped decades ago and do not necessarily reflect the patterns of computations in neural networks. I propose modifications to common primitives provided by performance libraries to make them better building blocks for artificial neural networks, with a focus on inference, i.e., evaluation of a pre-trained artificial neural network. I suggest three classes of performance primitives for the convolutional operators and two optimized building blocks for softmax operators.

High-intensity convolutional operators with large kernel sizes and unit stride benefit from asymptotically fast convolution algorithms based on Winograd transforms and Fast Fourier transforms. I jointly consider Fourier or Winograd transforms and the matrix-matrix multiplication (GEMM) of blocks of transformed coefficients and suggest a tuple-GEMM primitive, which balances the number of irregular memory writes in the transformation with sufficient register blocking and instruction-level parallelism in the matrix-matrix multiplication part. Tuple-GEMM can be thought of as a batched GEMM with a fixed architecture-dependent batch size and can be efficiently implemented as a modification of the Goto matrix-matrix multiplication algorithm. I additionally analyze small 2D Fast Fourier transforms and suggest options that work best for modern wide-SIMD processors.

Lower-intensity convolutional operators with small kernel sizes, non-unit strides, or dilation do not benefit from the fast convolution algorithms and require a different set of optimizations. To accelerate these cases I suggest replacing the traditional

GEMM primitive with a novel Indirect GEMM primitive. Indirect GEMM is a slight modification of GEMM and can leverage the extensive research on efficient GEMM implementations. I further introduce the Indirect Convolution algorithm. It builds on top of the Indirect GEMM primitive, eliminates the runtime overhead of patch-building memory transformations, and substantially reduces the memory complexity in convolutional operators compared to the traditional GEMM-based algorithms.

Pointwise, or 1x1, convolutional operators directly map to matrix-matrix multiplication and prompt yet another approach to optimization. I demonstrate that neural networks heavy on pointwise convolutions can greatly benefit from sparsifying the weights tensor and representing the operation as a sparse-matrix-dense-matrix multiplication (SpMM). I introduce neural network-optimized SpMM primitives. While SpMM primitives in Sparse BLAS libraries target problems with extremely high sparsity (commonly 99% or more) and non-random sparsity patterns, the proposed SpMM primitive is demonstrated to work well with moderate sparsity in the $70-95\%$ range and unpredictable sparsity patterns.

The softmax operator is light on elementary floating-point operations but involves evaluation of the exponential function, which in many implementations becomes the bottleneck. I demonstrate that with a high-throughput vector exponential function the softmax computation saturates the memory bandwidth and can be further improved only by reducing the number of memory access operations. I then constructively prove that it is possible to replace the traditional three-pass softmax algorithms with a novel two-pass algorithm for a runtime reduction of up to 28%.

I implemented the proposed ideas in the open-source NNPACK, QNNPACK, and XNNPACK libraries for acceleration of neural networks on CPUs, which at the time of their releases delivered state-of-the-art performance on mobile, server, and Web platforms.

CHAPTER I

INTRODUCTION

Artificial neural networks are a powerful tool. They are the state-of-the-art models for image classification [41], object recognition [80, 62, 79], artistic image manipulation [51, 29, 54], speech recognition [4], speech synthesis [91], machine translation [97], and many other artificial intelligence tasks. However, neural networks stand out not only in impressive effectiveness on a variety of artificial intelligence problems, but also in their insatiable appetite for computational power. A higher computational budget lets artificial intelligence researchers build larger models, scaling them up to the limits of the modern hardware results in progressively higher accuracy [78, 89, 65]. Fig. 1, 2, 3 illustrate this relationship on three popular families of neural networks architectures: MobileNet v1 [46], MobileNet v2 [82], and EfficientNets [89].

Machine learning on live perceptual data, such as video and audio streams, is particularly challenging as real-time processing imposes a strict time limit for neural network inference, while low latency requirements and privacy considerations associated with perceptual data dictate that neural network processing happen locally, on the same end-user device that produced the audio or video stream. These devices can

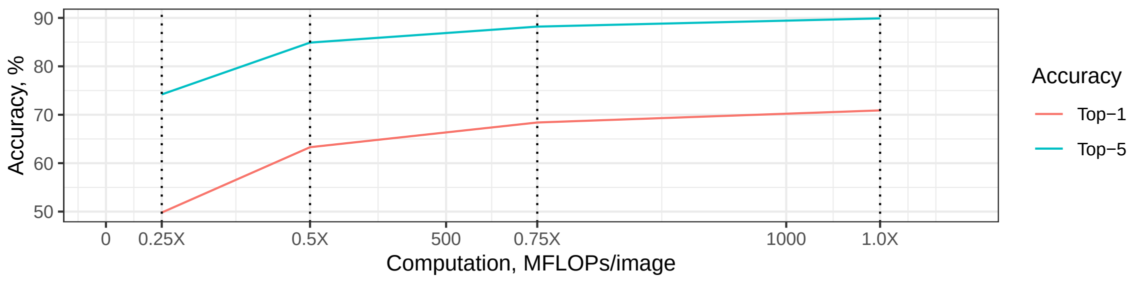


Figure 1: Accuracy and inference computations for the MobileNet v1 family of neural network architectures

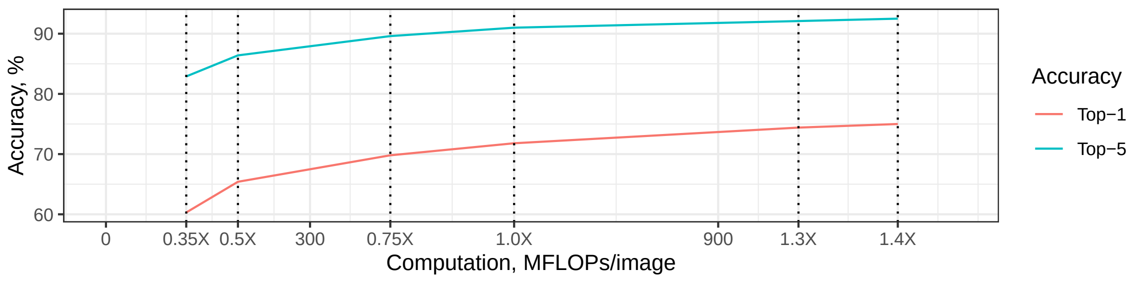


Figure 2: Accuracy and inference computations for the MobileNet v2 family of neural network architectures

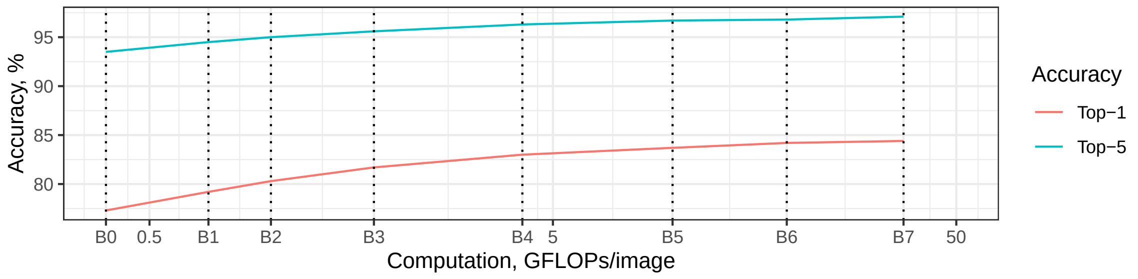


Figure 3: Accuracy and inference computations for the EfficientNet family of neural network architectures

range from IoT with microcontrollers to mobile phones to desktop computers, but share two features: a CPU as the least common denominator of fragmented hardware and orders of magnitude lower computational capabilities than the HPC-class GPUs and neural network accelerators (TPUs) used to train neural network models. Table 1 illustrates practically achievable performance on four types of end-user devices and suggests that neural network models operate in the mode where even slight reductions in model complexity to improve inference time costs a steep degradation in the quality metrics for model outputs.

Table 1: Single-core performance on MobileNet v3 Large model on TensorFlow Lite framework with the XNNPACK inference engine.

Device	Form-Factor	Environment	Performance, GFLOPS
MacBook Pro 11,3	laptop	macOS	46
MacBook Pro 11,3	laptop	WAsm	5.1
Pixel 2	phone	Android	9.9
Pixel 2	phone	WAsm	1.5
Raspberry Pi 4	embedded	Linux	5.9
Raspberry Pi Zero W	IoT	Linux	0.26

Researchers have explored two different paths to improve CPU performance on neural network layers, which can be termed as the performance library approach and the compiler approach. The performance library approach, implemented in most deep learning frameworks, is to leverage existing scientific computing libraries for portable, yet optimized implementation. In particular, the most computationally intensive neural network components – convolutional, recurrent, and fully-connected operators – can be built upon the GEMM primitive from the BLAS libraries. For fully connected and recurrent operators, a GEMM-based implementation is trivial, but convolutional layers need certain data repacking that was first suggested by Chellapilla et al [10], and then rediscovered and popularized by the author of Caffe [53]. Mathieu et al [67] justified using a Fourier transform for convolutional layers, and Zlateski et al [103] presented an implementation on top of Intel's MKL FFT, fftw, and cuFFT libraries. Later, Lavin [58] suggested using Winograd transforms for convolutional layers and demonstrated implementation of top of the Intel's MKL SGEMM primitive[1]. Another approach is to build a compiler to synthesize implementations optimized for

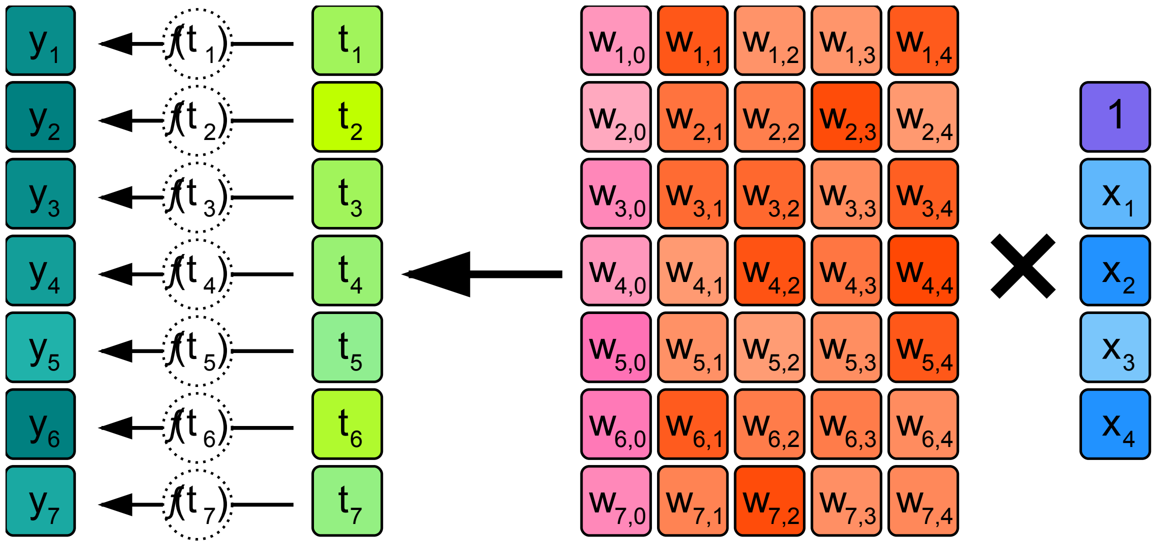
[1]A later version of the same paper, co-authored by Lavin and Gray, presents results only on GPU, with a custom GEMM implementation.

specific cases that appear in neural networks. This approach can exploit all available information, including parameters of the layers and their position in the computational graph. A modern take on this method is represented by TensorFlow XLA [59], Apache TVM [12], and PyTorch Glow [81] frameworks, which leverage LLVM infrastructure for code-generation down to machine instructions. The compiler approach, however, tends to suffer from less optimal code-generation in the critical loops than expert-tuned assembly in performance libraries.

Here, I present a middle-ground approach. I aim to improve the performance of neural network layers built on top of a low number of scientific computing primitives, while preserving the portability inherent to this approach. I suggest that a **slight modification of conventional scientific computing primitives – basic linear algebra subroutines (BLAS), Fast Fourier transform, and elementary functions – is sufficient to produce a high-performance implementation of common neural network layers**. This approach leverages the wealth of existing research on efficient algorithms for scientific computing primitives and improves code portability by sharing high-level parts of algorithms implementation, as well as unit tests and benchmarks, for architecture-specific micro-kernels. Some of the proposed modified primitives fit into other modern developments in scientific computing, e.g., batched and mixed-precision BLAS, and could share parts of implementation with next-generation performance libraries that address these needs.

CHAPTER II

CONVOLUTIONAL NEURAL NETWORKS



y_1 y_2 y_3 y_4 y_5 y_6 y_7
$f(t_1)$ $f(t_2)$ $f(t_3)$ $f(t_4)$ $f(t_5)$ $f(t_6)$ $f(t_7)$
t_1 t_2 t_3 t_4 t_5 t_6 t_7
$w_{1,0}$ $w_{1,1}$ $w_{1,2}$ $w_{1,3}$ $w_{1,4}$
$w_{2,0}$ $w_{2,1}$ $w_{2,2}$ $w_{2,3}$ $w_{2,4}$
$w_{3,0}$ $w_{3,1}$ $w_{3,2}$ $w_{3,3}$ $w_{3,4}$
$w_{4,0}$ $w_{4,1}$ $w_{4,2}$ $w_{4,3}$ $w_{4,4}$
$w_{5,0}$ $w_{5,1}$ $w_{5,2}$ $w_{5,3}$ $w_{5,4}$
$w_{6,0}$ $w_{6,1}$ $w_{6,2}$ $w_{6,3}$ $w_{6,4}$
$w_{7,0}$ $w_{7,1}$ $w_{7,2}$ $w_{7,3}$ $w_{7,4}$
$\times$
1 x_1 x_2 x_3 x_4

Figure 4: Traditional neural network layer with 4 input and 7 output neurons

Artificial neural networks are machine learning models loosely inspired by neuroscience, a scientific discipline about the organization of the nervous system. In artificial neural networks computations are organized in layers. A layer applies a linear transformation to its input (a vector), produces an output vector, and then transforms the output vector with a nonlinear activation function. Fig. 4 illustrates this process. It is common to augment the input vector with an element hard-wired to 1.0 (purple input on fig. 4) so that, with some transformation parameters, the linear transformation could produce negative outputs even if all inputs are positive. Most practical activation functions transform each element independently, and the most popular activation function – the rectified linear unit (ReLU) – just replaces negative components with zeros.

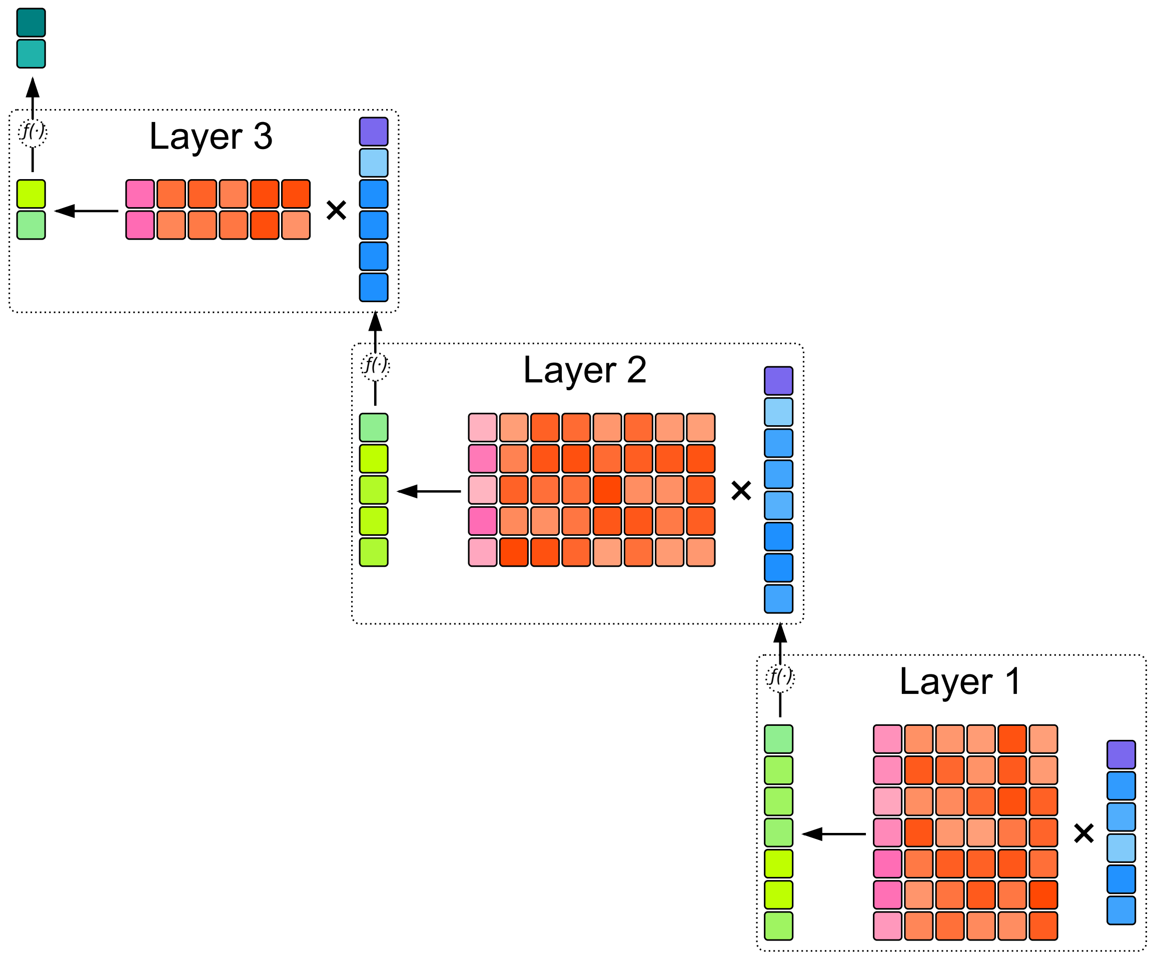


Figure 5: Example of a deep neural network with three layers

A single layer has a limited capability of learning relationships in the data, so artificial neural networks use multiple layers stacked on top of each other: the output of one layer serves as input for another layer. For this reason, multi-layer artificial neural networks are often called **deep neural networks**, and in advances in the structure and initialization of neural networks increased the viable number of layers from about a half-dozen to over a thousand [85, 88, 41]. Fig. 5 demonstrates the data flow in a three-layer deep neural network. Besides stacking layers, it is common for neural network architectures to concatenate outputs of several layers [88, 49, 47] or combine them through addition [41] or maximum [31] operations.

The linear transformations in the neural network layers can express contributions of every input element to every output element, but this flexibility is not always good. LeCun et al [60] discovered that artificial neural networks achieve better results on computer vision tasks if they restrict the possible connections between inputs and outputs to preserve locality and suggested the LeNet architecture for handwritten digit recognition. Unlike previous neural network architectures, LeNet used new types of layers, which take into account the 2-dimensional structure of images: convolutional layers and pooling layers. A convolutional layer restricts the linear transformation so that each output pixel depends only on a small neighborhood of input pixels. A pooling layer aggregates a tile of neighboring pixels into a single output pixel, and, in general, is not a linear transformation. Neural networks with convolutional layers are called **convolutional neural networks**, and convolutional layers are typically their most computationally intensive parts. Later research generalized convolutional and pooling layers to arbitrary numbers of dimensions and demonstrated the usefulness of 1D convolutions for speech recognition [2] and natural language processing tasks as well as 3D convolutions for video classification.

Parameters of the linear transformation in neural network layers are learned in a training process. During training samples from a dataset are supplied as inputs to the first layer of a neural network, and weights are adjusted to make outputs closer to the reference outputs from the dataset. Usually this process involves a two-step backpropagation algorithm: first, in the forward pass, propagate the signals from the input layers towards output layers and calculate the loss between the computed output and reference output; then, in the backward pass, propagate the gradient of the loss from the output layers towards the input layers. Additionally, the backward pass computes the gradients of layer parameters with respect to loss. These gradients are then used to update the parameters' values. Most commonly, the stochastic gradient descent with a momentum algorithm [57] is used to update the parameters, albeit several advanced algorithms were suggested [21, 99, 56, 63, 44]. For better efficiency, training usually consumes a dataset in batches of multiple samples. The batch size is limited by the memory footprint of the network, including all intermediate signals and gradients. Batches of up to $8,096$ were reported to be practical [34], and a typical batch size is $64 - 512$ samples.

After the training process converges to a good set of parameters, the neural network is ready to make predictions on new inputs – a process called inference. In inference only the forward pass of a neural network is computed, and no gradients are involved. Most applications of neural networks to live perceptual data are latency-sensitive and limit the batch size in inference to a single image.

Table 2: 2D image tensor layouts in popular neural network frameworks

Name	Shape	Frameworks
NCHW	batch × channels × height × width	PyTorch [76], Caffe [53], Theano [3]
NHWC	batch × height × width × channels	Tensorflow [1], CNTK [98]
CHWN	channels × height × width × batch	cuda-convnet2, Neon

A number of high-level frameworks [76, 1, 53, 16, 3, 11, 98] were developed to assist artificial intelligence research. These frameworks differ in the tensor layout they use in their layers' implementation, summarized in Table 2. NCHW layout enjoys the widest support: even Tensorflow and CNTK frameworks, which started with native NHWC-layout layers, can optionally use NCHW. However, NHWC and CHWN layouts have their benefits, too: NHWC simplifies implementation of convolutional layers via matrix-matrix multiplication, and the CHWN layout with large batch sizes is particularly efficient for training on GPUs.

2.1 Fully Connected Layers

A fully connected layer represents a general linear transformation without any constraints. These layers consume and produce 2D tensors. If the layer input has more than 2 dimensions, e.g., a multi-channel 2D image, its dimensions are reshaped into a 2D matrix. Thus, in both the NCHW and NHWC frameworks, the inputs to the fully connected layer have the same dimensions. Layer parameters are represented by a 2D weights matrix, which has the same dimensions in NCHW, NHWC, and CHWN layouts. Table 3 details dimensions of matrices in all three layouts.

Table 3: Shape of input, output, and weights matrices in a fully-connected layer

Layout	Input matrix	Output matrix	Weights matrix
NCHW	batch $\times$ $\text{channels}_{\text{in}}$	batch $\times$ $\text{channels}_{\text{out}}$	$\text{channels}_{\text{out}} \times \text{channels}_{\text{in}}$
NHWC	batch $\times$ $\text{channels}_{\text{in}}$	batch $\times$ $\text{channels}_{\text{out}}$	$\text{channels}_{\text{out}} \times \text{channels}_{\text{in}}$
CHWN	$\text{channels}_{\text{in}}$ $\times$ batch	$\text{channels}_{\text{out}}$ $\times$ batch	$\text{channels}_{\text{out}} \times \text{channels}_{\text{in}}$

A fully connected layer can be thought of as an affine transformation of the input matrix and can be implemented via matrix-matrix multiplication (GEMM) in a BLAS library. If the input vectors are augmented with unit elements, this augmentation is performed implicitly: the matrix-matrix multiplication of explicit input elements is followed by the addition of a bias vector, which contains the column of weights corresponding to the implicit unit element. This case is demonstrated in Fig 6.

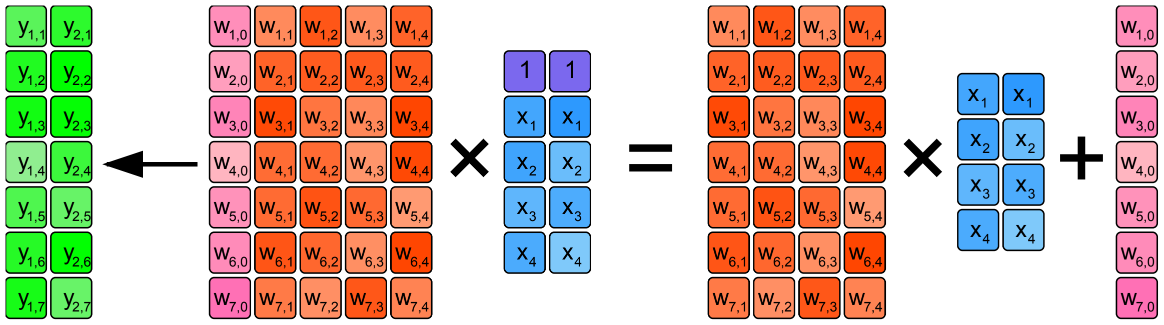


Figure 6: The representation of a fully connected layer with implicit unit inputs as a matrix-matrix multiplication with bias. The bias is broadcasted along the column dimension and added to each column of the weights-input product.

High-performance BLAS libraries for CPUs typically implement the Goto algorithm for matrix-matrix multiplication [33]. Goto and van de Geijn suggested that the most important factor for efficient matrix-matrix multiplication is cache blocking, i.e., reuse of loaded memory blocks, on multiple levels of the memory hierarchy: registers, level-1 cache, and higher level caches. Their algorithm for matrix-matrix multiplication, now widely adopted, arranges computations in several layers of blocking. On the lower level, it relies on a “micro-kernel”, which accumulates in registers a product of long panels of the two matrices.

The micro-kernel is the key to the overall algorithm’s efficiency: Van Zee et al [92] demonstrated that so long as the micro-kernel is well-optimized, all other parts of the algorithm can be portably implemented in a high-level language and still produce a state-of-the-art implementation. The micro-kernel is typically implemented in assembly and makes heavy use of SIMD, architecture-specific instruction features, software prefetching, and pipelining. The micro-kernel expects one panel to reside in the level-1 cache, and another panel to be streamed from the level-2 cache to the level-1 cache. The width of the panels is limited by the number of registers, and their length is dictated by the level-1 cache size. A loop around the micro-kernel iterates panels of the second matrix factor inside level-2 cache, and another, outer, loop iterates panels of first matrix factor inside level-3 cache. Additionally, two other outer loops traverse level-3 cache-sized blocks of the first matrix factor and level-2 cache-sized blocks of second matrix factor to produced the complete matrix-matrix multiplication algorithm.

To satisfy cache-associativity limitations, input matrices are typically repacked so that the panels accessed by the micro-kernel are contiguously stored in memory. With large matrices, the cost of repacking is negligible compared to the cost of matrix-matrix multiplication: if the matrices are $N \times N$, then the cost of repacking is $O(N^2)$ while computational cost is $O(N^3)$. However, for small matrices repacking can take a considerable fraction of time and may offset the benefits of the blocked implementation. Recent studies have proposed several methods to improve BLAS efficiency on small problems.

First, some BLAS libraries provide an interface to separately repack input matrices and perform computations on repacked representations. This interface enables reuse of a repacked representation for multiple operations and is particularly helpful for neural network inference. In inference mode, the weights matrix is static and can be repacked during initialization, and then re-used for multiple forward passes.

Secondly, it is possible to modify micro-kernels to access one or both panels in the original layout without repacking. If the matrix is small, it may fit into cache without associativity conflicts, but the efficiency of the micro-kernel would still suffer from an imperfect operation of hardware prefetchers.

The third method is a variation of this approach where the code is generated for the specific matrix shapes of the application. Heinecke et al [42] evaluated the use of just-in-time (JIT) compilation for small-size BLAS problems and demonstrated possible speedups of over 10x.

Unbatched inference deserves special attention, because in this common case matrix-matrix multiplication becomes matrix-vector multiplication, and BLAS libraries provide a separate primitive (GEMV) for this operation. While matrix-matrix multiplication is often compute-bound, matrix-vector product is intrinsically bandwidth-bound. Matrix-vector multiplication can be decomposed into a series of dot products and implemented on top of the DOT primitive from BLAS. In this approach the input vector would be loaded into registers and, potentially, a higher level of the memory hierarchy for each row of the factor matrix, which limits performance of this bandwidth-bound operation. The number of loads can be reduced by combining the computation of several dot products in a fused-DOT operation [92].

Alternatively, it can be thought of as a matrix-vector multiplication where the matrix has a fixed small number of rows. The number of rows is limited by two factors: floating-point registers and cache associativity. The multiple dot products are accumulated in floating-point registers, and thus the number of floating-point registers on the target architecture puts an upper bound on the fusion factor. But a fused-DOT micro-kernel also reads multiple rows of the matrix in parallel, and if the matrix length is close to a large power of 2, these loads may compete for the same sets in the cache. Thus, cache associativity puts another, and usually tighter, limit on the fusion factor.

2.2 Convolutional Layers

A convolutional layer a special linear transformation that consumes and produces multi-channel 1D, 2D, or 3D signals[1]. Convolutional layers reuse the same set of learned parameters at multiple locations of the image, which induces locality, and improves model accuracy, especially on computer vision tasks. Due to the reuse of parameters, convolutional layers are more computationally intensive than fully-connected layers. Together with the closely related deconvolutional layers, they are typically the most computationally expensive parts of neural networks.

[1]Generalizations to higher dimension exist, but are rarely used in practice.

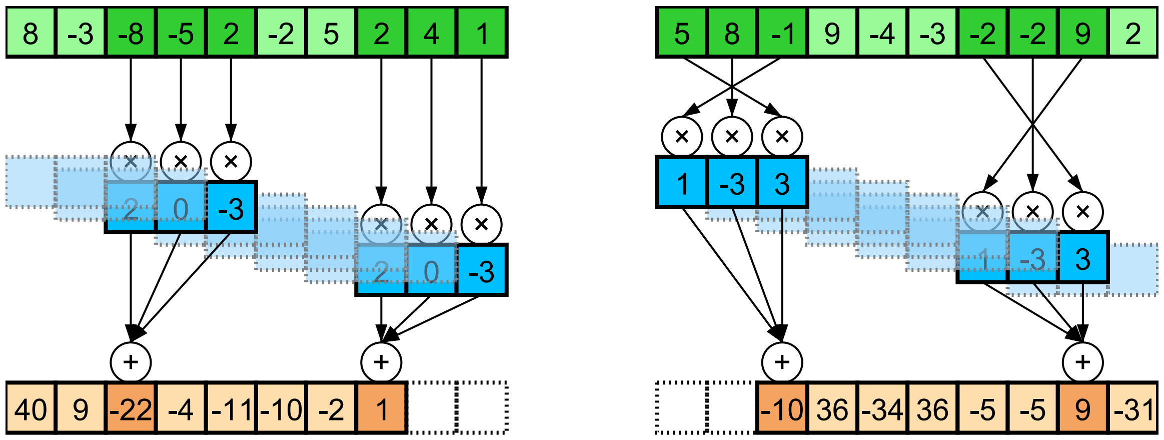


Figure 7: One-dimensional cross-correlation (left) and convolution (right). The key difference is the traversal order of the kernel over the input.

A convolutional layer is built on top of two signal processing operations: cross-correlation and convolution. Fig. 7 illustrates this operation in the 1D case. A kernel (blue) slides along the input vector (green) and produces output elements (orange) through a dot product operation. The same kernel parameters are reused in all locations. The major difference between cross-correlation and convolution is in the order in which the sliding kernel traverses the input: in cross-correlation the kernel elements start traversing from the beginning of the input signal and are multiplied by the signal in their direct order; in convolution the kernel traverses from the end towards the beginning of the input, and its elements are multiplied by the signal in reverse order. Importantly, the number of elements in the input, kernel, and output vectors is linked through the relation

$$N_{\text{out}} = N_{\text{in}} - K + 1 \tag{1}$$

where N_{out} and N_{in} is the number of elements in the output and input vector respectively, and K is the number of elements in the kernel vector. As seen in Fig. 7, this equation holds because there are not enough valid inputs to produce the last (first) outputs of the cross-correlation (convolution).

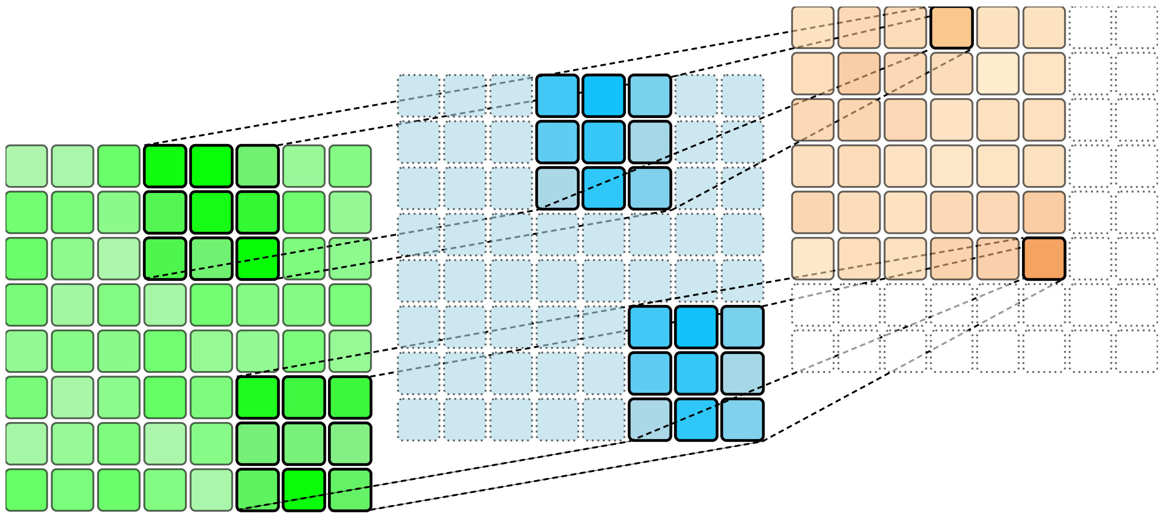


Figure 8: Two-dimensional cross-correlation. A 2D kernel (middle) slides along the 2D input (left) to produce elements of the 2D output (right). The same kernel elements are reused at different positions of the input.

The two-dimensional case is presented in Fig. 8 and is widely used in image processing. Here, a 2D kernel slides along the input image and produces the output image through a dot product with input subimages. The shapes of the input, kernel, and output images are connected through the equations,

$$\begin{aligned} H_{\text{out}} &= H_{\text{in}} - H_{\text{K}} + 1 \\ W_{\text{out}} &= W_{\text{in}} - W_{\text{K}} + 1 \end{aligned} \tag{2}$$

where H_{out}, W_{out}, H_{in} and W_{in} are the height and width of the output and input images respectively, and H_K and W_K are the kernel height and width.

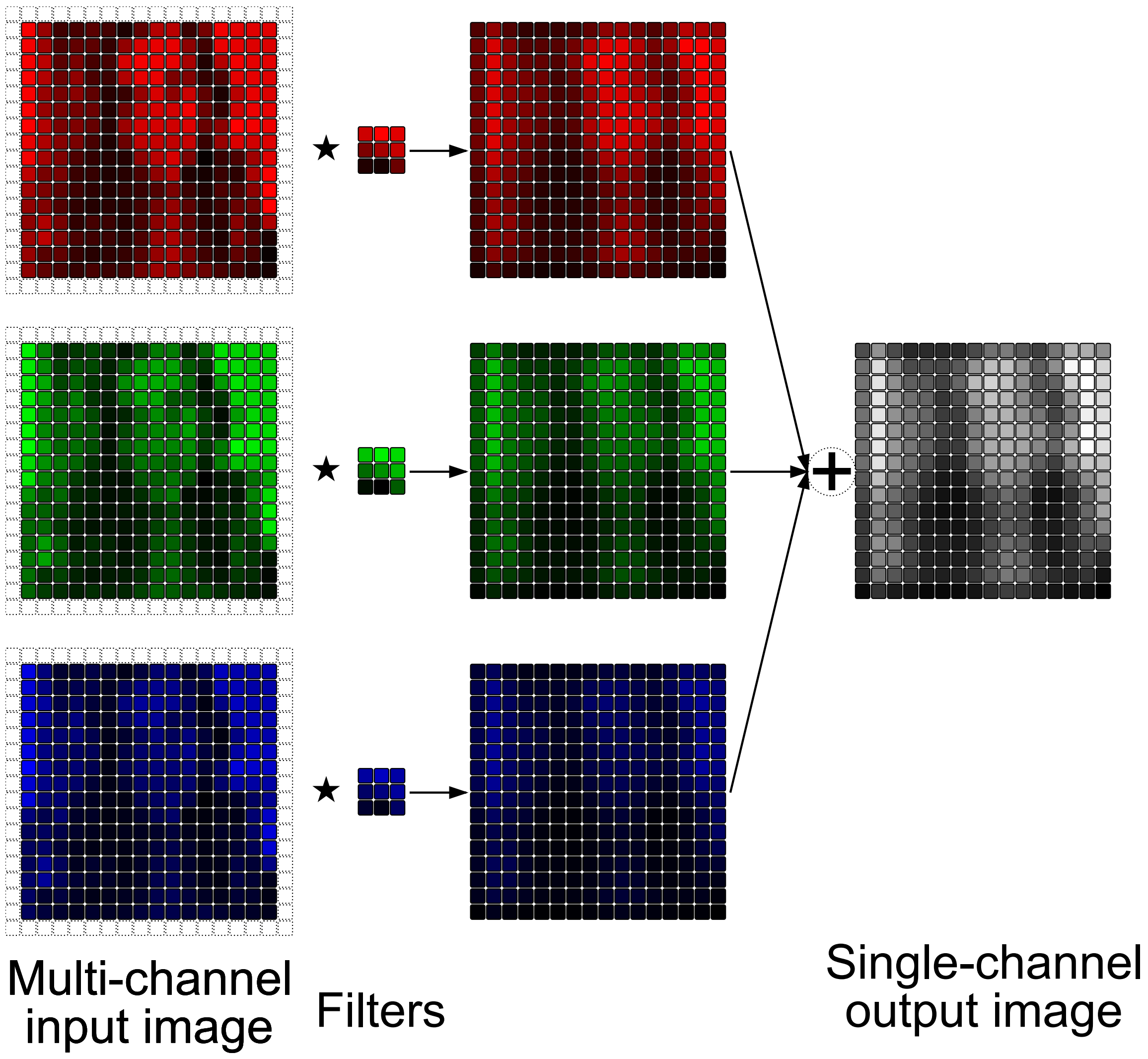


Figure 9: Forward propogation in a 2D convolutional layer with 3-channel input. The $\star$ operator denotes cross-correlation operation.

Convolutional layers add another dimension to the cross-correlation and convolution operations: channels. A convolutional layer combines multiple kernels – as many as there are channels in the input – into a filter. In the forward pass of a 2D convolutional layer, every channel of the input is cross-correlated with a channel of a filter, and then all channels are accumulated to produce a single-channel output. Figure 9 illustrates this computation.

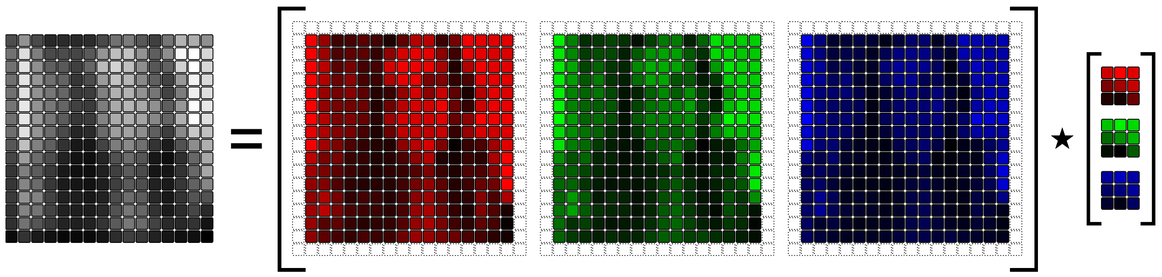


Figure 10: Representation of forward propagation in a 2D convolutional layer as a dot product operation.

Computations in the convolutional layer exhibit structure similar to BLAS operations. If one replaces multiplication ($\times$) with a cross-correlation ($\star$), the computation in Fig. 9 could be instead represented as a dot product operation, as in Fig. 10. This representation is useful for analyzing the computational properties and limits of convolutional layers, which are well-studied for BLAS operations.

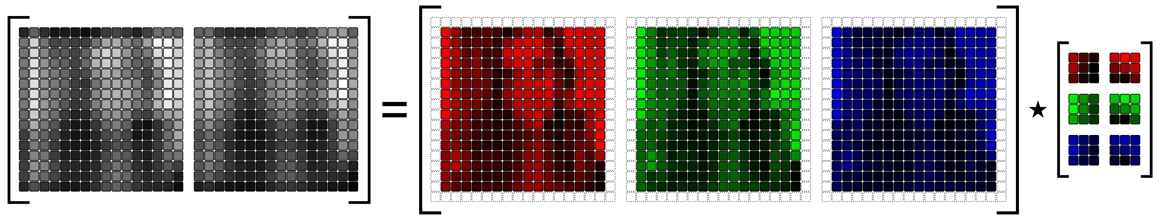


Figure 11: Representation of forward propogation in a 2D convolutional layer with multiple filters as a vector-matrix product operation.

In practical convolutional neural networks, convolutional layers have not one, but many, filters. In the forward pass, the multi-channel input is cross-correlated with each multi-channel filter to produce multiple output channels (as many as there are filters). In the space with $(+, \star)$ operations, this computation can be represented as a vector-matrix multiplication, depicted in Fig. 11.

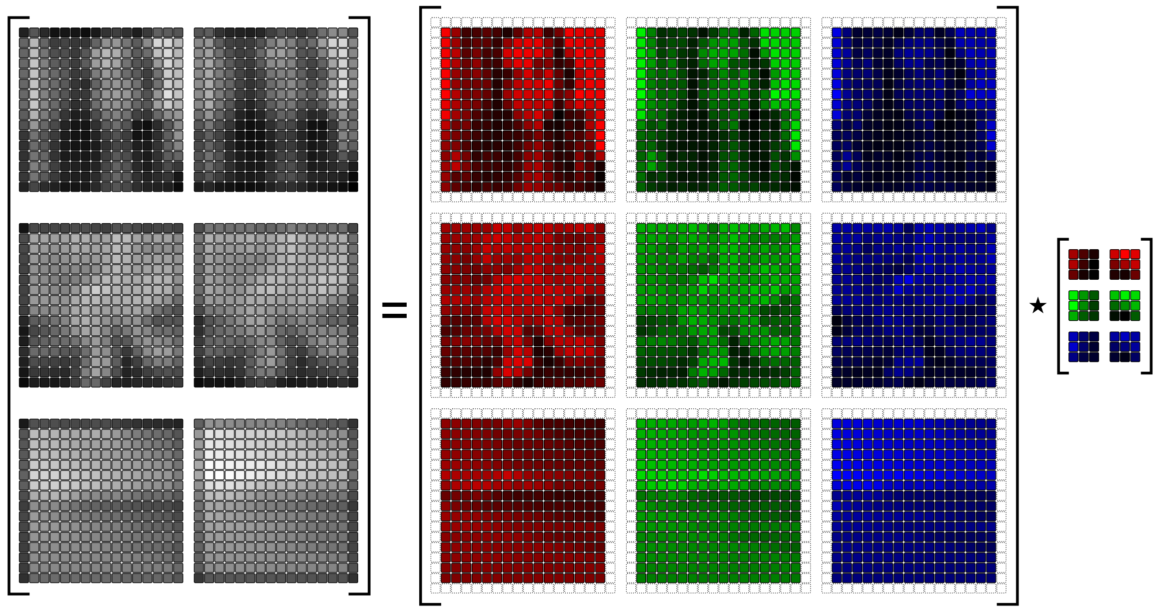


Figure 12: Matrix-matrix multiplication-like structure of computations in the forward propagation of a 2D convolutional layer with multiple filters and a batch of input/output images.

During model training and, in many use-cases, during inference, convolutional layers are fed with a batch of inputs and produce a batch of outputs. In this operation, every channel of every input image is cross-correlated with every channel of every filter, and structure of computations, illustrated in Fig. 12 resembles matrix-matrix multiplication. The dimensions of the input, filter, and outputs matrices in this representation are batchsize × channels, channels × filters, and batchsize × filters respectively.

Matrix-matrix multiplication performs $O(n^3)$ operations on $O(n^2)$ elements and is compute bound for large problem sizes. Thus, the forward pass in the convolutional layer is compute bound when batch size, number of channels, and number of filters are all large, regardless of kernel size. Moreover, the commonly-used kernel sizes of 3×3 (18 FLOPs per output) and 5×5 (50 FLOPs per output) shift the balance even further towards saturation of compute units.

2.2.1 im2col+GEMM algorithm

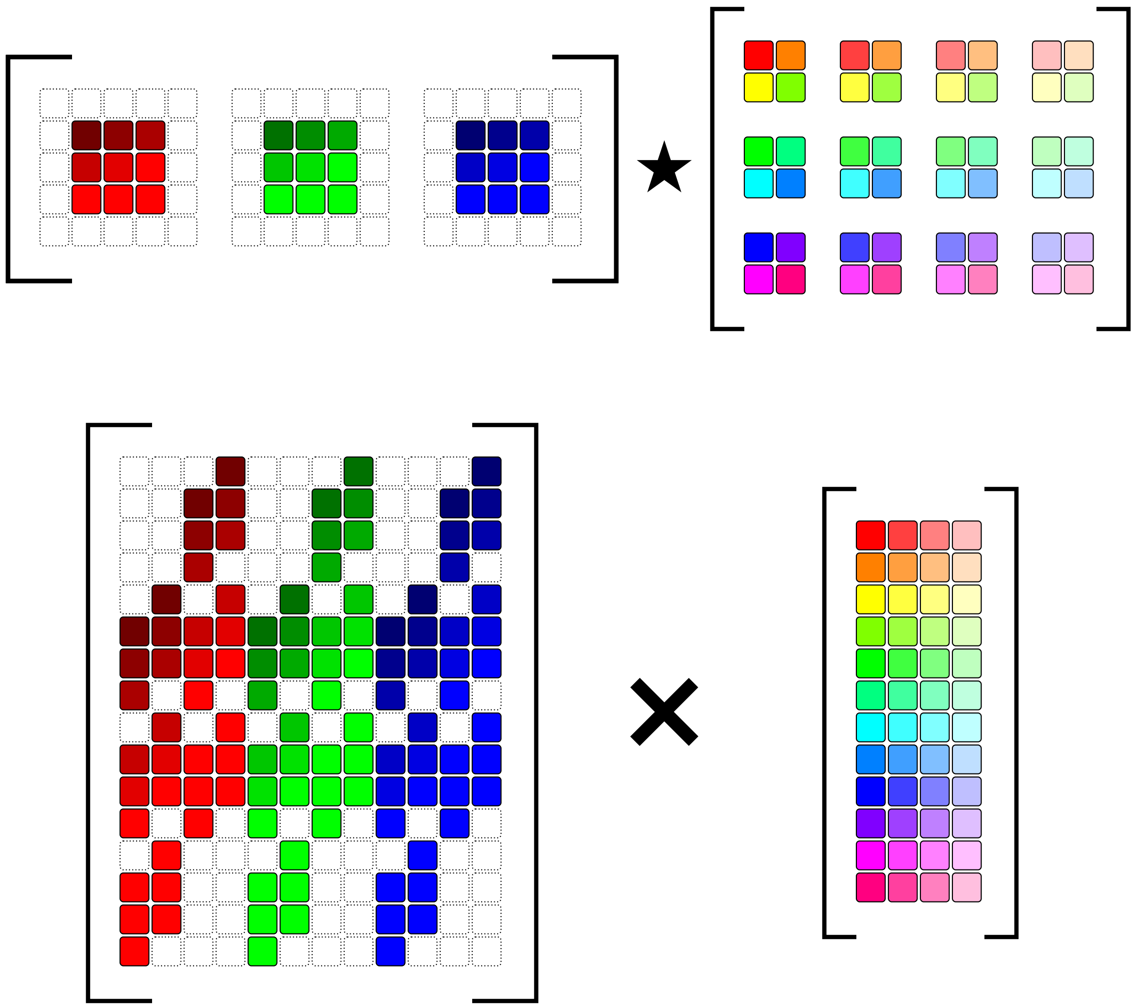


Figure 13: Transformation of forward propagation in a convolutional layer into a real matrix-matrix multiplication problem.

A popular method to implement forward and backward propagation in convolutional layers is to transform a convolution into a matrix-matrix multiplication and leverage matrix-matrix multiplication (GEMM) routines in a BLAS library to do the actual computation. This method, illustrated in Fig. 13 and commonly known as the im2col+GEMM algorithm, was first suggested by Chellapilla et al [10], and popularized in the Caffe framework [53].

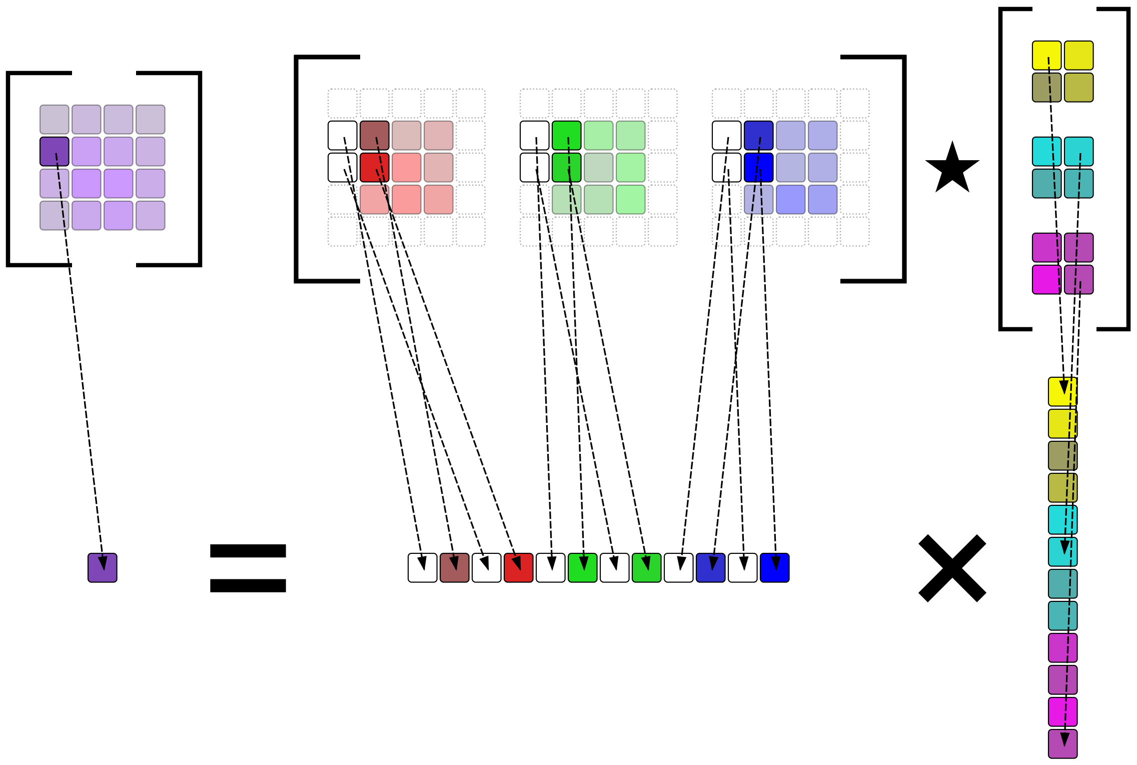


Figure 14: Repacking of elements in the im2col+GEMM algorithm.

Fig. 14 provides the intuition behind the transformation of forward propagation in a convolutional layer into a matrix-matrix multiplication problem. Each element of the output tensor (on the left) is a dot product of a filter (on the right) by a multi-channel subimage of the input (middle). Different pixels of the same channel of the output tensor are produced as a dot product of different multi-channel subimages of the input and the same filter. Pixels in the same position in other channels are produced via dot product of the same subimage of the input by different filters.

Table 4: High-performance BLAS implementation for popular platforms

Platform	OpenBLAS	BLIS	Eigen	Accelerate	MKL	ACML	ESSL	QSML
x86	Yes	Yes	Yes	Yes	Yes	Yes	No	No
x86-64	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
ARM	Yes	Yes	Yes	Yes	No	No	No	Yes
ARM64	Yes	Yes	Yes	Yes	No	No	No	Yes
POWER	Yes	Yes	Yes	Yes	No	No	Yes	No
PNaCl	No	Yes	Yes	No	No	No	No	No
Asm.js	No	Yes	Yes	No	No	No	No	No

The im2col+GEMM algorithm has its pros and cons. On the plus side, it combines good performance with universal portability: popular platforms provide a choice of multiple high-performance BLAS implementations (see Tab. 4), and exotic platforms can bootstrap an optimized BLAS implementations through auto-tuning [95]. The major drawback of this algorithm is the substantial memory overhead of $K_H \times K_W$ times the size of layer input. Intermediate tensors are usually the largest memory consumers in convolutional neural networks, especially with non-unit batch size, and the proportial increase in memory consumption can be prohibitive in mobile, Web, and IoT environments.

2.2.2 Fast Convolution algorithms

The convolution theorem states that, under certain conditions, a Fourier transform of a convolution of two signals is a pointwise product of their Fourier transforms. Combined with a Fast Fourier Transform algorithm, the convolution theorem suggests a less computationally intensive alternative to the im2col+GEMM algorithm. Fig. 15 demonstrates the FFT-based 2D convolution algorithm: zero-pad the kernel to the size of the input, transform the padded kernel and the input to the Fourier domain, elementwise multiply the Fourier images, and compute the inverse FFT on the product to get the convolution output.

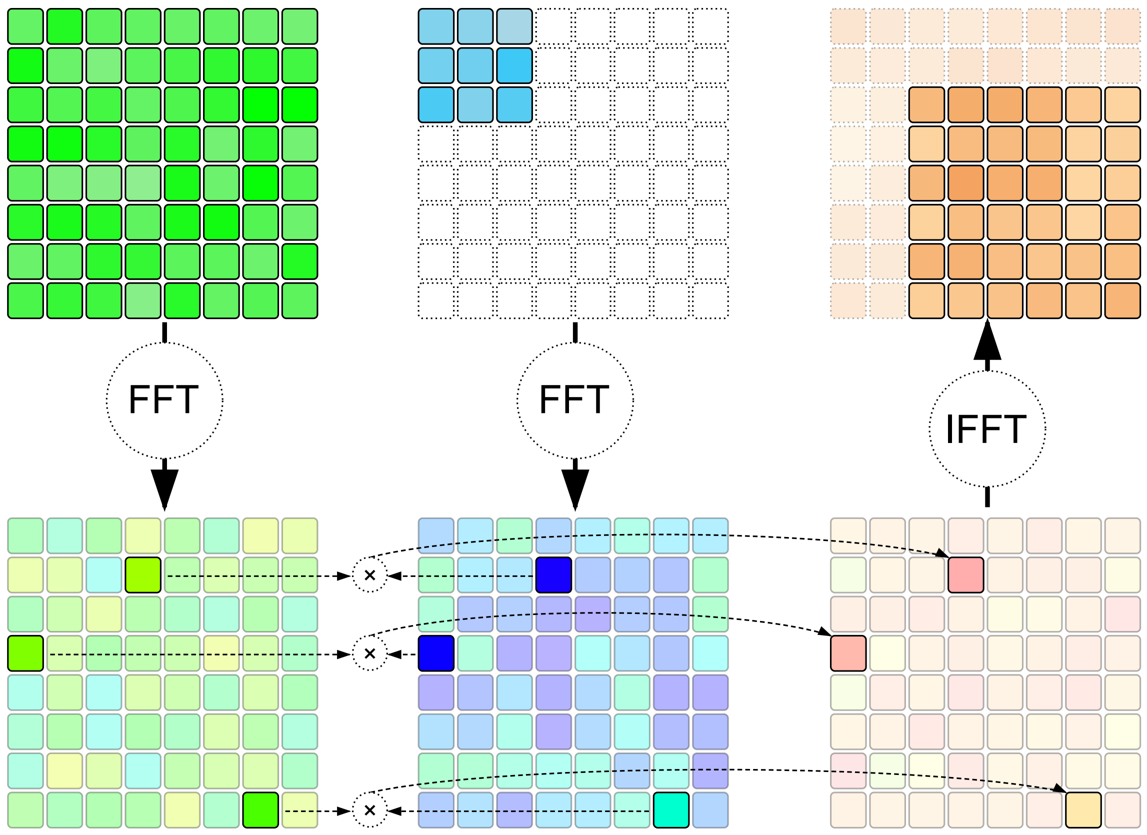


Figure 15: 2D convolution algorithm based on Fourier transform.

The Fourier transform assumes periodic signals, and the full output computed through the convolution theorem represents a circular convolution: its outputs near the top-left border depend on a block of input that wraps around its border, as shown in Fig. 16. These outputs are meaningless for the normal convolution operation, thus it discards them and only uses the bottom-right block of $(H_{\text{in}} - H_{\text{K}} + 1) \times (W_{\text{in}} - W_{\text{K}} + 1)$ outputs.

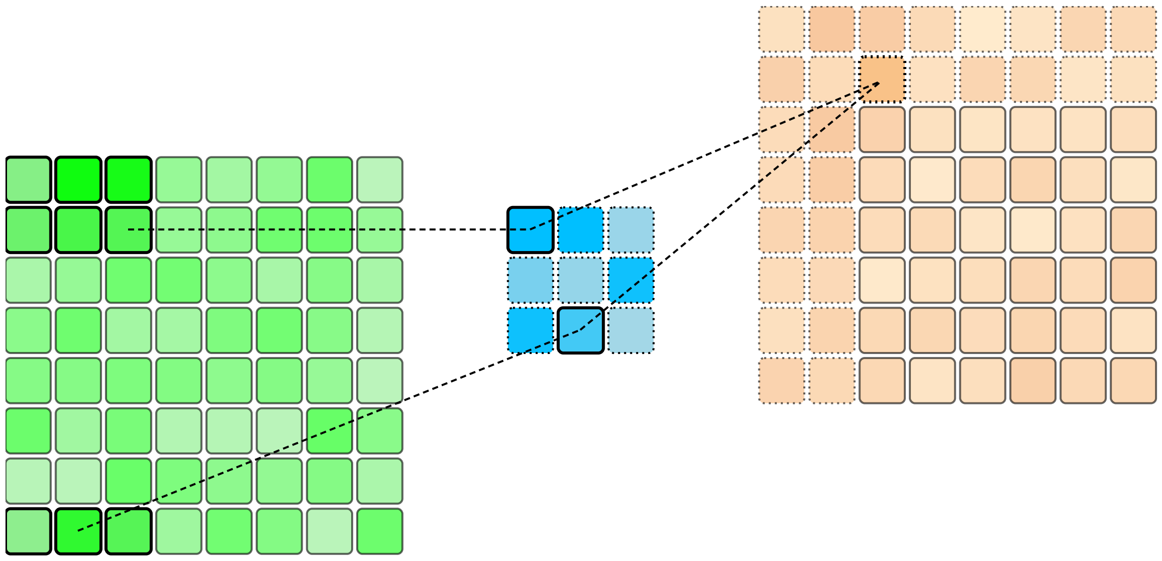


Figure 16: Circular 2D cross-correlation.

Computation of a 2D cross-correlation on top of the Fast Fourier Transform and the convolution theorem differs in two details: we multiply the Fourier transform of the input by the complex conjugate of the Fourier transform of the zero-padded kernel and use the top-left block of inverse FFT outputs while discarding the outputs near the bottom-right border. Fig. 17 illustrates the computations involved.

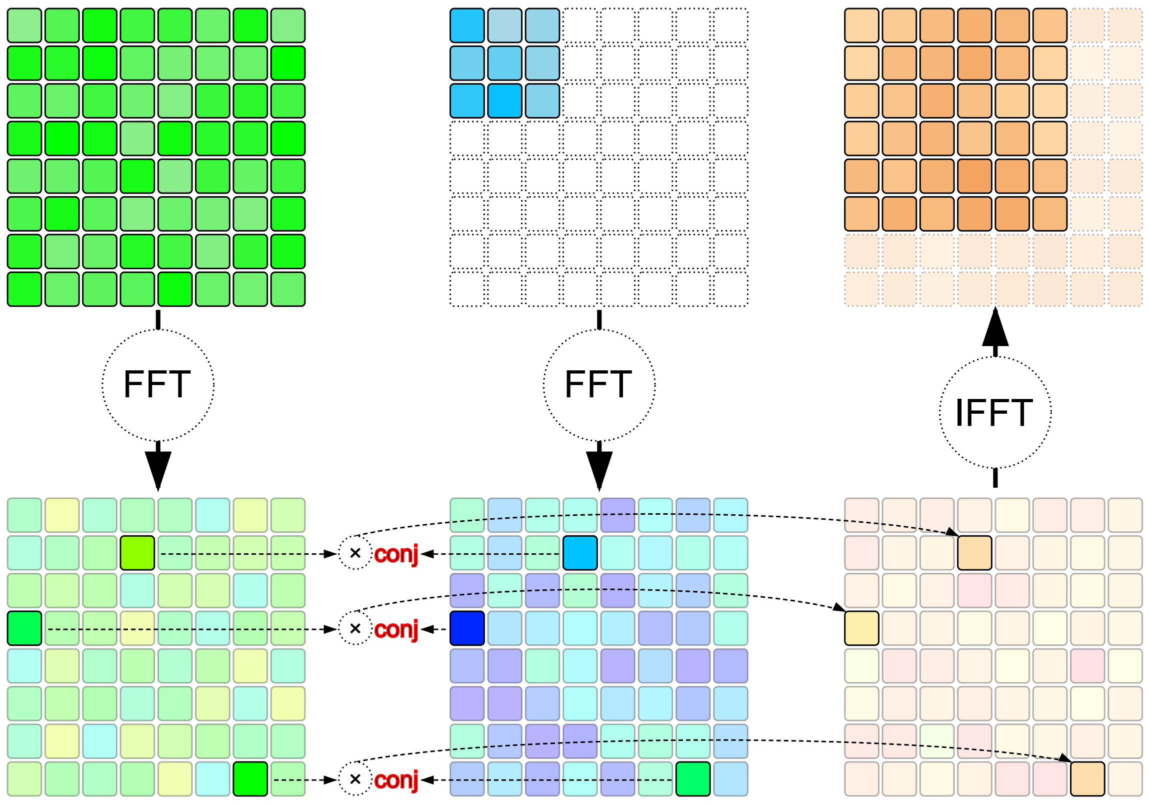


Figure 17: 2D cross-correlation algorithm based on Fourier transform.

A 2D cross-correlation trivially extends to multiple channels, as illustrated in the first three columns of Fig. 18. The algorithm from Fig. 17 applies to each channel of the input and the filter to produce channels of the output image, which are subsequently accumulated together. However, as the Inverse Fourier Transform is a linear transformation, it can be factored out after the accumulation. The improved algorithm, illustrated in the fourth column of Fig. 18, accumulates blocks of FFT coefficients for the output image, and then computed the Inverse FFT of the result.

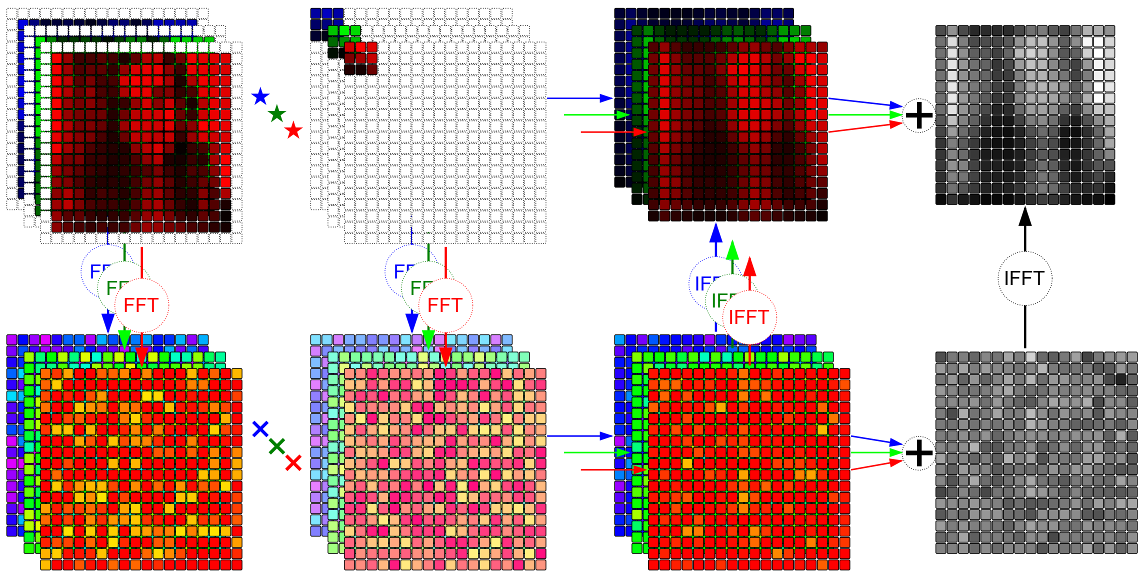


Figure 18: Forward pass of a 2D convolutional layer based on Fourier transform.

By leveraging Fast Fourier Transforms and the convolution theorem, the algorithm in Fig. 18 reduces the number of floating-point operations in the forward pass of a 2D convolutional layer compared to direct computation, but also involves two inefficiencies that dramatically limit its practical applicability.

First, the filter needs to be zero-padded to the size of the input image. This procedure is an intrinsically memory-intensive operation, and it greatly increases the memory footprint of the filter. Commonly-used filter sizes are 3×3 and 5×5, while input images can be on the order of QVGA (320×240) or higher, resulting in over $1,000$ times expansion in the memory footprint! Such an increase in would typically exceed the capacity of the system, especially on mobile and embedded systems.

Secondly, the algorithm implies doing a Fast Fourier Transform on input image-sized blocks, which are generally unconstrained. Although algorithms for doing Fast Fourier Transform on arbitrary-sized blocks are known [27], their efficiency varies across input sizes, resulting in unexpected performance cliffs.

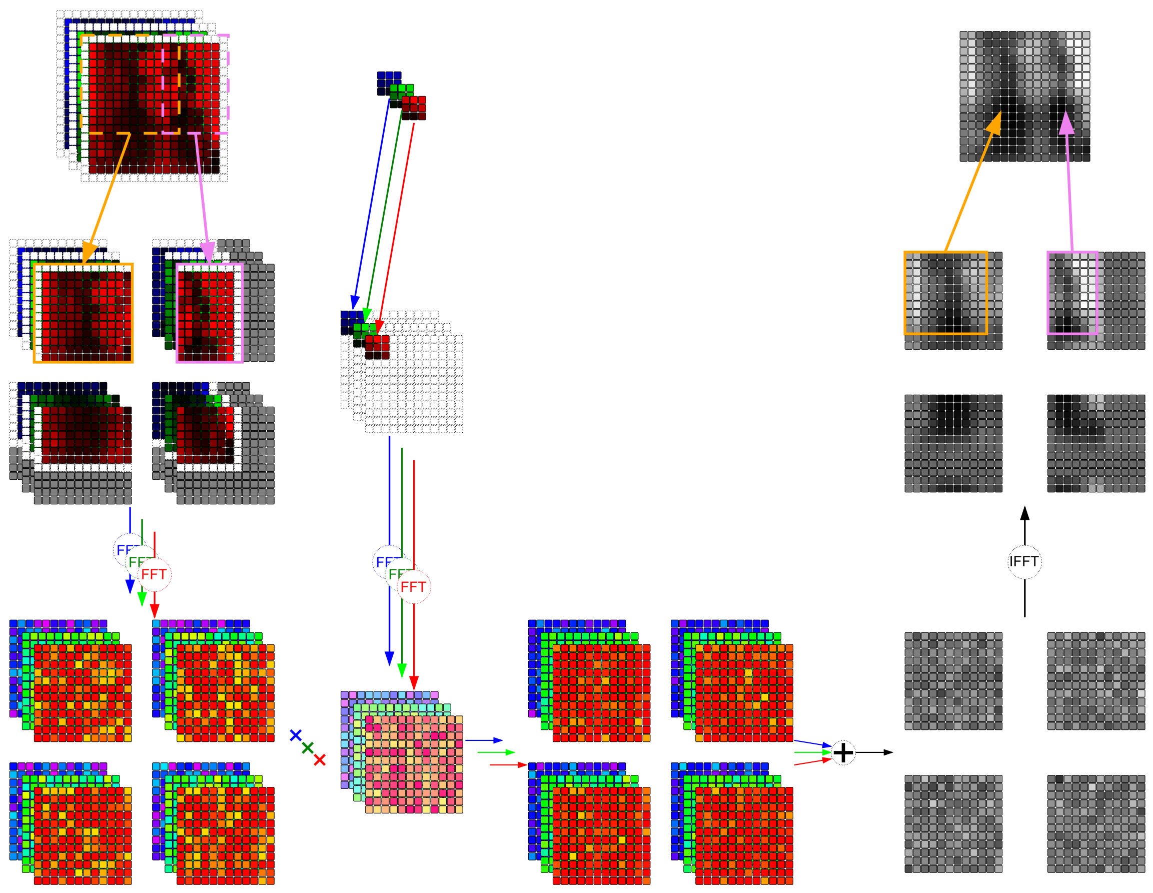


Figure 19: Forward pass of a 2D convolutional layer based on Fourier transform with tiling.

These inefficiencies can be resolved by tiling the Fourier transform-based convolution algorithm [93]. This modification, illustrated in Fig. 19, rather than computing the Fourier transform of the input image all-at-once, splits the input image into fixed-sized input tiles, computes the convolution of these tiles with the kernels using the Fourier transform and the convolution theorem, and then concatenates output tiles to cover the whole output image. Importantly, the output is densely covered with non-overlapping tiles, but the corresponding input tiles do overlap, because they are $(H_K - 1) \times (W_K - 1)$ pixels bigger.

Tiling works around the inefficiencies of the naive Fourier transform-based algorithm. First, the filter is zero-padded only to the size of a tile, which is typically much smaller than input image. Second, an implementation may choose the tile size that is convenient for a Fast Fourier transform algorithm, e.g., a small power of two.

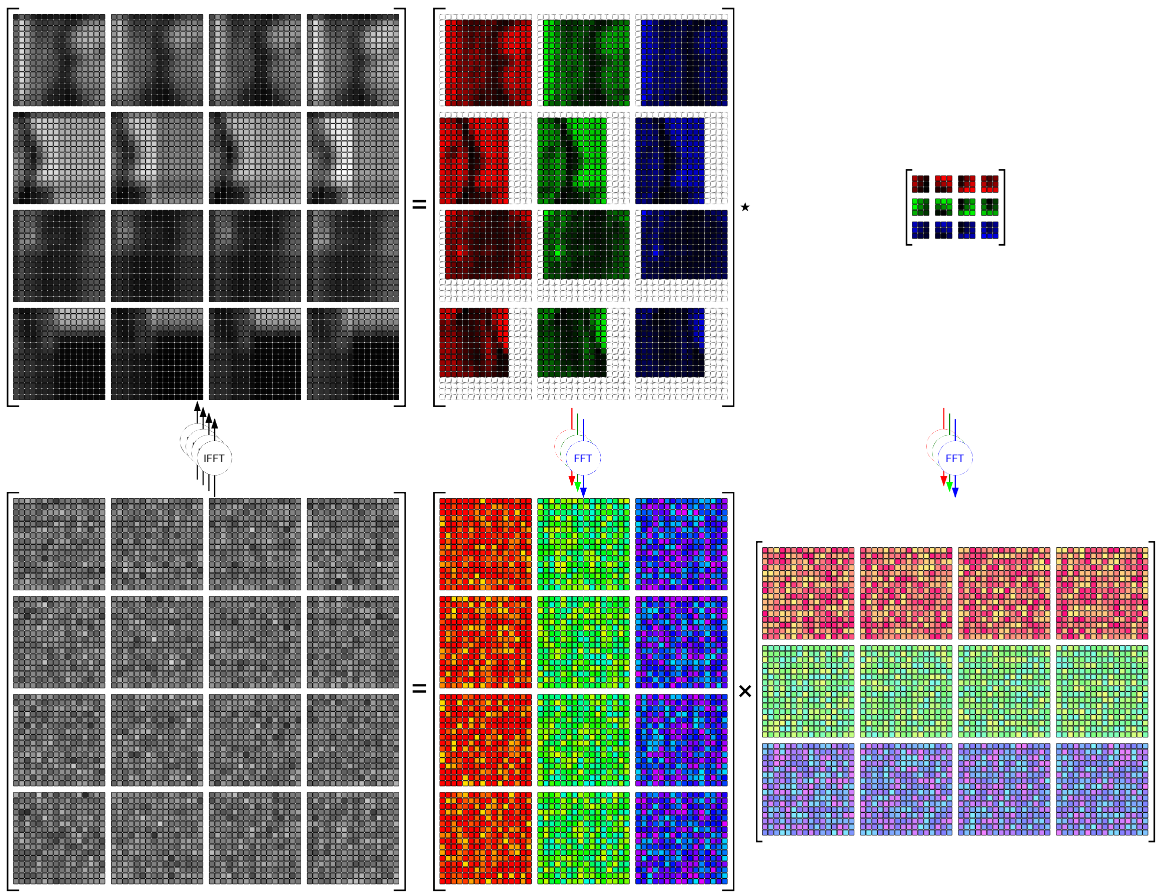


Figure 20: 2D convolution algorithm based on Fourier transform.

Fig. 20 illustrates how the Fast Fourier Transform with tiling transforms the nature of the computation in the convolution operator. In the top row, the matrix elements represent tiles of the input, output, and filter tensors, and "multiplication" of these elements is represented by 2D cross-correlation operation. Importantly, the number of multiply-adds in the 2D cross-correlation operation is directly proportional to the kernel size in the filters. In the bottom row, the matrix elements represent frequency-domain tiles of input, output, and filter tensors, and multiplication of these elements is represented by elementwise multiplication of their complex-domain components. Thus, in the bottom row, each multiplication costs two multiply-adds, regardless of the kernel size in the filters. This simplification of the elementary operations is the principal reason for the high efficiency of the FFT-based convolution algorithm.

Conversion between the top row and the bottom row representations requires a Fourier transform on the tiles, and its cost cuts into the savings from multiplying individual tiles. However, as seen in Fig 20, this conversion has lower asymptotic complexity than the "matrix multiplication" part: we transform $O(N^2)$ elements in each matrix to make the $O(N^3)$ matrix-matrix multiplication cheaper. Thus, when the number of input channels, output channels, and input tiles is sufficiently large, the cost of a Fourier transform on the tiles is negligible compared to the multiplication-accumulation of the transformed tiles.

The Fourier transform-based convolution algorithm in Fig. 20 drastically reduces the number of floating-point operations compared to im2col-based algorithms, but it is possible to do even better. The Winograd transform-based convolution algorithms [58] further reduces the number of floating-point operations by close to $2\times$. Fig. 21 illustrates the general structure of computations in this algorithm, with its substantial similarities to a Fourier transform-based algorithm. In fact, there are just two differences between the Winograd transform-based and Fourier transform-based algorithms:

1. These algorithms replace Fast Fourier transform (FFT) and Inverse FFT with Winograd transforms. The algorithm uses three different Winograd transforms, each specific to the type of blocks of coefficients: input Winograd transform (IWT), kernel Winograd transform (KWT), and output Winograd transform (OWT). Each Winograd transform is a linear transformation of a block of coefficients with a rather sparse transformation matrix.

2. While Fourier-transformed blocks contain complex elements, Winograd-transformed

blocks use only real coefficients. Multiplication-accumulation of complex numbers involves four floating-point operations, but multiplication-accumulation of real numbers requires only two floating-point operations. Thus, the matrix-matrix multiplication part in Fig. 21 is twice cheaper than the same part in Fig. 20, and constitutes the main advantage of a Winograd transform-based algorithm over a Fourier transform-based one.

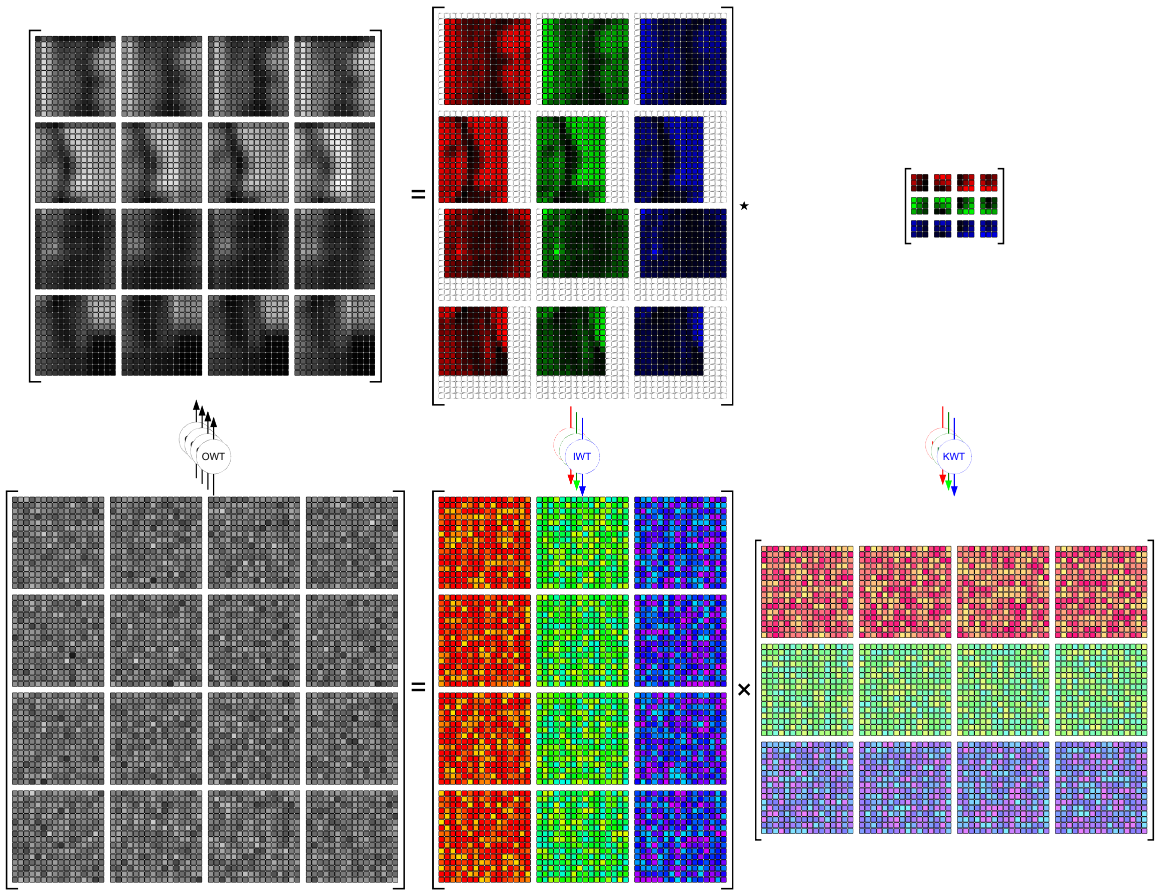


Figure 21: 2D convolution algorithm based on the Winograd transform.

The Winograd transform-based method comes with its own disadvantages, too. First, input/kernel/output Winograd transforms are specific not only to the tile size, but also to the kernel size. If an implementation intends to support arbitrary kernel sizes, it needs to provide many variants of Winograd transforms. Secondly, while FFT involves only order-of-one coefficients, Winograd transform involves a combination of large and small coefficients, which results in poor numerical stability. Thirdly, while FFT can be used with very large tiles without becoming inaccurate, a Winograd transform loses too much accuracy beyond 8×8 tiles.

CHAPTER III

PRIMITIVES FOR HIGH-INTENSITY CONVOLUTIONS

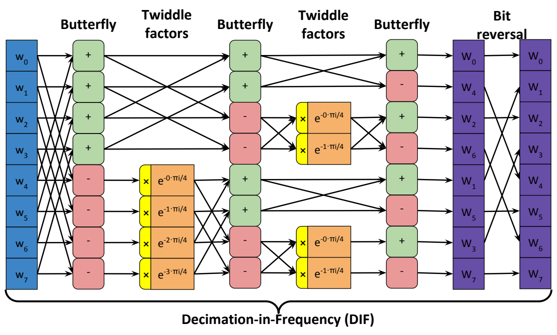
3.1 Introduction

Model	1×1	3×3	5×5
VGG	Yes	Yes	No
U-Net	Yes	Yes	No
ResNet	Yes	Yes	No
AlexNet	No	Yes	Yes
Overfeat	No	Yes	Yes

Table 5: Configurations of parameters in convolutional layers in early CNN architectures.

Early convolutional neural network architectures for computer vision extensively relied on convolutions with large kernels to aggregate information across large neighborhoods of pixels. Table 5 illustrates that many of these networks incorporated 3×3, 5×5, or even larger convolution kernels. Even though these neural network architectures are no longer state-of-art in terms of accuracy and efficiency, for a variety of reasons they remain widely used in both research and industry, for the following reasons.

- These neural network architectures are well-understood, covered in even introductory books on deep learning, and implemented in all major neural network frameworks.
- The above neural network architectures are relatively simple, and can be easily adapted to new datasets and experimental deep learning techniques, such as quantization and pruning. ResNet and U-Net are often considered as a baseline that new methods should be evaluated on or against.



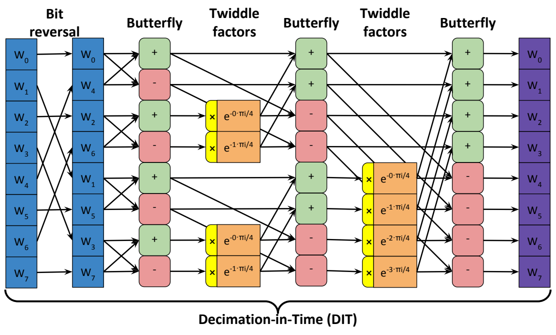


Figure 22: Decimation-in-frequency and decimation-in-time FFT algorithms for 8-sample sequence

- Image features produced by VGG architectures are knows to produce more visually appealing results than features extracted by other architectures. This advantage is intrinsically subjective and its origins are not well-understood, but research on style transfer tends to prefer VGG architectures as feature extractors.

Due to the above factors, achieving good performance and efficiency on legacy neural network architecture remains an important goal. These architectures devote the majority of their FLOPs budget to convolutions with large kernel sizes, and thus the Fast Convolution algorithms that can reduce the computational complexity of such convolutions are of crucial importance to accelerating legacy neural network architectures. In this chapter, I present the details of adapting Fast Convolution algorithms to modern wide-SIMD processors and suggest modifications to the standard FFT and GEMM primitives that make them efficient building blocks for the Fast Convolution algorithms.

I consider two types of transformations for the Tiled Fast Convolution algorithm: the Fast Fourier transform and the Winograd transform.

3.2 Fast Fourier Transforms

The Fast Fourier Transform (FFT) is a family of algorithms to compute discrete Fourier transforms in $O(n \log n)$ operations instead of $O(n^2)$ operations in its naive implementation as a matrix-vector multiplication. Several variants of FFT algorithms were proposed in the literature; for this research I choose to implement the radix-2 Cooley-Tukey algorithm, which has a simple structure and operates on blocks that can be efficiently implemented on SIMD architectures. The radix-2 Cooley-Tukey algorithm recursively reduces an FFT of N elements to two FFTs of $\frac{n}{2}$ elements.

Two variants of radix-2 Cooley-Tukey FFT algorithm are known: decimation-in-time (DIT) and decimation-in-frequency (DIF) algorithm. The algorithms differ by the order of recursive FFT steps: DIF algorithm starts with high-order FFT steps and then descends to a two-sample FFT while DIT starts with a two-sample FFT and then ascends to high-order FFT steps. The DIF algorithm expects the input in normal order and produces output in bit-reversed order; a separate shuffle step is required to rearrange the output to normal order. The DIT algorithm expects the input in bit-reversed order and produces output in normal order. Fig. 22 illustrates the difference between the two algorithms.

Additional optimizations are possible for Fourier transforms of real data. One algorithm performs the FFT on a real sequence of N samples using a complex FFT of $\frac{N}{2}$ samples. Another algorithm is known to do an FFT of two interleaved sequences of N real samples each using a complex N-sample FFT.

Table 6: Instruction characteristics on Intel Skylake [26]

Instructions	Execution Port	Latency
FP ADD/SUB	0 or 1	4
FP MUL/FMA	0 or 1	4
FP LOGICAL	0, 1 or 5	1
FP STATIC BLEND	0, 1 or 5	1
FP PERMUTE/SHUFFLE	5	1 or 3
INT ALU	0, 1, 5 or 6	1
INT MUL	1	3

A large fraction of floating-point performance in modern CPUs comes from SIMD instructions. Intel Skylake features 256-bit wide SIMD with AVX2 and FMA3 instruction sets, with instruction characteristics detailed in Tab. 6. The processor is capable of issuing two floating-point additions, multiplications, or FMA instructions each cycle, which produces a result after 4 cycles. Thus, at least eight independent floating-point instructions are needed to saturate the computational resources. Two kinds of instructions could lower the compute peak: integer multiplications and SIMD shuffles. Integer multiplications run on port 1, same as floating-point instructions, so each integer multiplication reduces the achievable compute peak. SIMD shuffles issue in parallel with floating-point computations, but the processor is capable of executing only one shuffle per cycle, and it can become a bottleneck. Logical and blend SIMD instructions, as well as simple scalar integer instructions can issue on many execution ports, and are less likely to become a bottleneck.

3.2.1 Layout of FFT coefficients

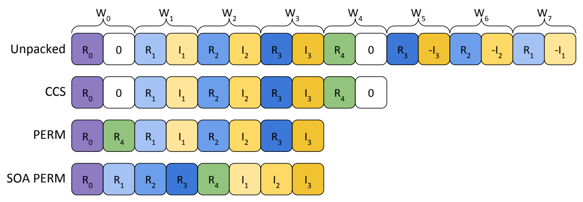


Figure 23: Layouts of real-to-complex coefficients FFT of 8 samples

A Fourier transform of N real samples produces N complex samples, which can be represented with $2N$ real numbers. Many of these numbers are redundant, and the layout of coefficients of real-to-complex FFT is an important aspect of a high-performance implementation. Fig. 23 illustrates several possible layouts, with mathematically related numbers sharing the same color. The top row displays the layout of N complex FFT coefficients. The coefficients exhibit Hermitian symmetry:

$$W_{\frac{N}{2}+i} \equiv W_{\frac{N}{2}-i} \quad \textbf{for } i = 1 \ldots \frac{N}{2} - 1.$$

The symmetry makes it possible to drop the last $\frac{N}{2} - 1$ coefficients. The resulting layout, called CCS, is shown in the second row. CCS layout is nearly standard and supported by all major FFT libraries, including Intel MKL, FFTW, and FFTS. However, this layout is not favorable to wide SIMD units: when N is a power of two, which is convenient and efficient for SIMD implementation, the $\frac{N}{2}$th element would alone occupy a SIMD register. One solution is to pack this coefficient into the imaginary component of element 0, which is implemented in PERM format in Intel MKL, shown in row 3. In my implementation I use MKL PERM format and its structure-of-arrays variant in row 4.

3.2.2 Composition of 2D FFTs

There are four options for implementing a 2D FFT, illustrated in Fig. 24. The implementation of all four options would be very labor-intensive, so I argue about

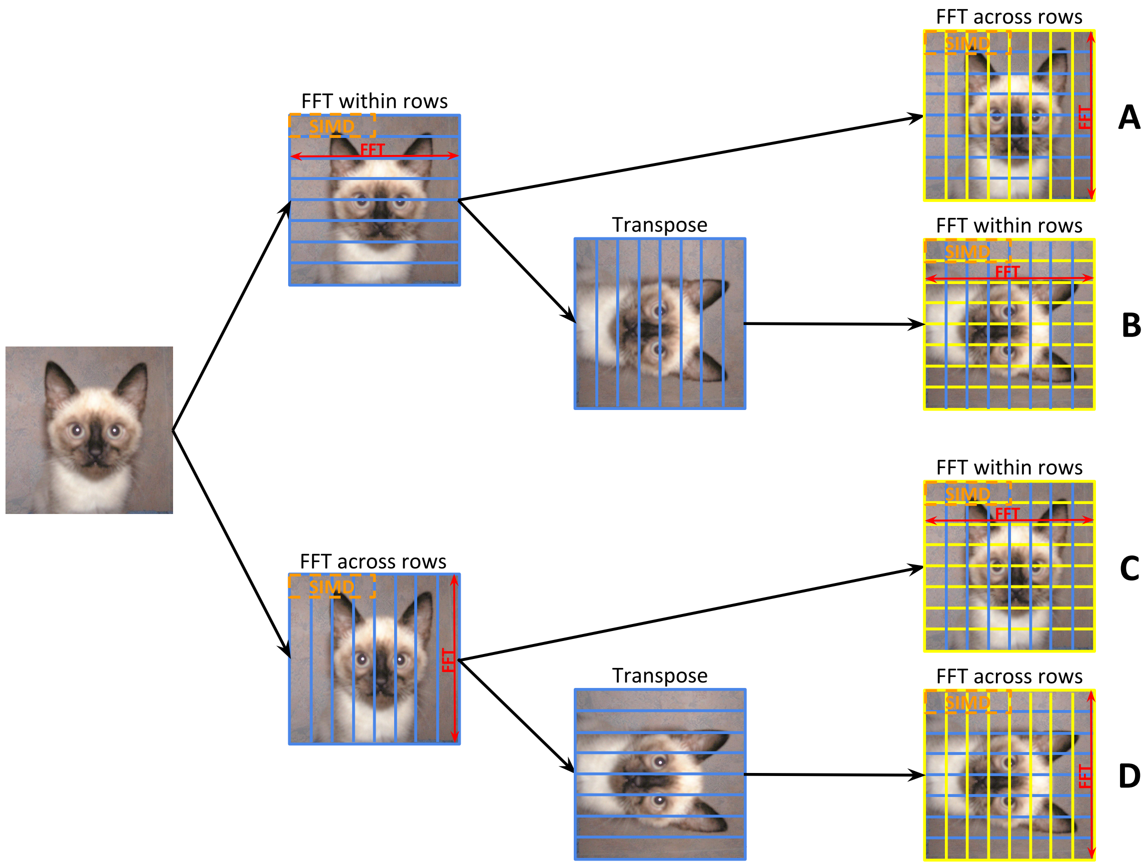


Figure 24: Implementation options for 2D FFT

the optimal option from a theoretical analysis.

The main concern in the 2D FFT implementation is to keep the ratio of SIMD shuffles to floating-point instructions as low as possible, to avoid bottlenecks on shuffle units. The 1D FFT within rows and the 1D FFT across rows uses different numbers of shuffle instructions. The within-rows variant does butterflies between elements of the same SIMD register, which requires shuffling the register. The across-rows variant does not need any shuffles. In addition to the 1D FFTs, options B and D include transposition of the image tile. A SIMD implementation of transposition consists mostly of SIMD shuffles, and it uses the same number of shuffle instructions as the within-row FFT of a tile. Now we can subsequently eliminate most options:

- We can eliminate option B, because it consists of two 1D FFTs within rows and a transposition and needs a redundant number of shuffle instructions.

- Options A and C differ only in the order of within-rows and across-rows FFTs. I suggest that it is better to first do FFT across rows, because the first transformation is a real-to-complex FFT, and it exposes additional symmetries, which are easy to use in an across-rows algorithm but require additional shuffles in within-rows variant. This observation eliminates option A in favor of option C.

- Options C and D use the same number of shuffles (transposition needs as many shuffles as within-row FFT of a tile), but in Option D they are all concentrated in the transposition operation, and in Option C they are interleaved with floating-point computations. Therefore, option C is less likely to create a pipeline bubble of shuffle instruction, and should therefore be preferred over option D.

3.2.3 Fused Butterfly Operations

The butterfly is one of the two computational components of the FFT. Given two elements, the bufferfly operation computes their sum and difference:

$$a_{\text{out}}, b_{\text{out}} \coloneqq a_{\text{in}} + b_{\text{in}}, a_{\text{in}} - b_{\text{in}}$$

Some operations can be fused into butterfly at no cost:

- Negation of b_{in}: $a_{\text{out}}, b_{\text{out}} \coloneqq a_{\text{in}} - b_{\text{in}}, a_{\text{in}} + b_{\text{in}}$.

- Negation of b_{out}: $a_{\text{out}}, b_{\text{out}} \coloneqq b_{\text{in}} + a_{\text{in}}, b_{\text{in}} - a_{\text{in}}$.

On modern processors the FMA operation often has the same performance characteristics as floating-point addition/subtraction. It makes it possible to fuse scaling of either a_{in} or b_{in} by an arbitrary factor, e.g., butterfly$(a_{\text{in}}, b_{\text{in}} \cdot c)$ can be computed with:

$$a_{\text{out}}, b_{\text{out}} := \underbrace{b_{\text{in}} \cdot c + a_{\text{in}}}_{\text{fma}(b,c,a)}, \underbrace{-b_{\text{in}} \cdot c + a_{\text{in}}}_{\text{fnma}(b,c,a)}$$

No explicit product $b_{\text{in}} \cdot c$ is formed, and the resulting code has the same cost as a regular butterfly.

I use the fusion technique in the FFT across rows, with the following benefits:

- Eliminate all multiplications by twiddle factors in a 4-sample complex FFT (and an 8-sample real FFT)
- Replace all multiplications by twiddle factors in an 8-sample complex FFT (and a 16-sample real FFT) with only two floating-point additions.
- Eliminate final scaling by $\frac{1}{N}$ in the inverse FFT.
- Eliminate several negations and multiplications in the real-to-complex 1D FFT

3.3 Winograd Transforms

I follow the specification of $F(6 \times 6, 3 \times 3)$ – the largest transformation described by Lavin [58] – which extends 3×3 kernels and 8×8 input tiles into 8×8 blocks of transformed coefficients. The convolution of the original tile with the kernel is equivalent to elementwise real product of transform coefficients. The output transform produces a 6×6 tile of output samples. Thus, the $F(6 \times 6, 3 \times 3)$ Winograd transform behaves similarly to an 8×8 Fourier transform on 3×3 kernels. Unlike the Fourier transform, the Winograd transform depends on the kernel size: my implementation following Lavin [58] handles only 3×3 kernels; support for other sizes is possible, but would require a separate implementation.

The 1D Winograd transform is a linear transformation and can be defined by a transformation matrix. The transformation matrix has many zeroes and additional structure, which I exploit in the implementation. I apply a 1D transform across rows of data block, then transpose it, and apply the same transformation again. Overall, both in the number of operations and in the structure of algorithm, the 2D Winograd transform is simpler than the 2D FFT.

3.4 Matrix Multiplication

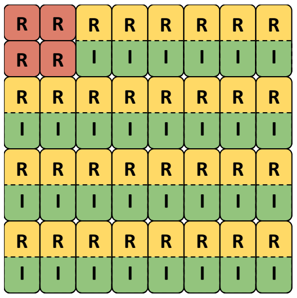


Figure 25: Layout of an 8×8 block of FFT coefficients. Red blocks indicate real coefficients, and yellow and green blocks indicate real and imaginary components of complex coefficients.

The blocks of coefficients from the Fourier or the Winograd transforms need to be reduced through a matrix multiplication-like operation. The FFT blocks have special structure, with few real and many complex elements, as depicted in Fig. 25 (blocks of Winograd transform coefficients contain only real numbers). A naive way to perform the reduction would be to arrange different elements of blocks into different matrices, and then call SGEMM or CGEMM for these matrices. However, two issues make such implementations undesirable.

First, to form the matrices for GEMM calls, each coefficient would need to be scattered to a distinct memory location. The scatter operation would add a significant cost to the transformation that produced the block of coefficients. To reduce the number of distinct memory writes, I scatter not individual elements of transformed blocks but SIMD tuples of elements. For example for the 8×8 block on Fig. 25 the first tuple would consist of the first two rows: they would be stored into a contiguous location, and the next two rows would be stored into a different contiguous location. Matrix multiplication on such tuples can be thought of as operating on interleaved matrices. I implement a custom GEMM operation, which operates on tuples and includes special handling for the real elements in the upper left corner of the block, following the classical Goto algorithm [33] and its modern extension to multi-core architectures [87]. Interleaving of matrices lowers the FLOPS-per-byte ratio in the inner kernel of the matrix multiplication and raises concern about whether such as implementation can be competitive with normal CGEMM. I address this concern in the next section.

The second issue with rearranging block elements for GEMM is that it involves double repacking of data. In the Goto algorithm [33], implemented by modern high-performance BLAS libraries, the input matrices are internally repacked, so that the data accessed by the inner kernel of a GEMM implementation is contiguously stored in memory. The parameters of this packing are usually hidden behind the GEMM interface. However, as one implements a custom matrix multiplication for tuples, one can repack the tuples on-the-fly in the transformation code. In my implementation, the FFT or Winograd transform that computes transformed blocks stores tuples of elements in the order that would be later optimal for the matrix multiplication. This optimization saves the extra repacking in the matrix multiplication implementation and partially offsets the performance issues due to lower compute intensity in the inner kernel.

3.5 Performance

I benchmarked my implementations on the convolutional layers of neural network models that won the ImageNet large scale visual recognition challenge [19] in 2012 [57], 2013 [83] and 2014 [85]. Table 7 presents details on the benchmarked convolutional layers. I excluded from the benchmark the first layers of AlexNet and Overfeat (fast

Model	Layer	Channels	Input	Padding	Kernel	8×8	16×16
VGG-A	conv1	$3 \rightarrow 64$	224×224	1	3×3	1444	256
VGG-A	conv2	$64 \rightarrow 128$	112×112	1	3×3	361	64
VGG-A	conv3.1	$128 \rightarrow 256$	56×56	1	3×3	100	16
VGG-A	conv3.2	$256 \rightarrow 256$	56×56	1	3×3	100	16
VGG-A	conv4.1	$256 \rightarrow 512$	28×28	1	3×3	25	4
VGG-A	conv4.2	$512 \rightarrow 512$	28×28	1	3×3	25	4
VGG-A	conv5	$512 \rightarrow 512$	14×14	1	3×3	9	1
AlexNet	conv2	$64 \rightarrow 192$	27×27	2	5×5	49	9
AlexNet	conv3	$192 \rightarrow 384$	13×13	1	3×3	9	1
AlexNet	conv4	$384 \rightarrow 256$	13×13	1	3×3	9	1
AlexNet	conv5	$256 \rightarrow 256$	13×13	1	3×3	9	1
Overfeat	conv2	$96 \rightarrow 256$	24×24	0	5×5	25	4
Overfeat	conv3	$256 \rightarrow 512$	12×12	1	3×3	4	1
Overfeat	conv4	$512 \rightarrow 1024$	12×12	1	3×3	4	1
Overfeat	conv5	$1024 \rightarrow 1024$	12×12	1	3×3	4	1

Table 7: Configurations of convolutional layers in performance evaluation

model) because they use convolutions with strides while my implementation only supports unit stride.

Fig. 26 illustrates the speedup of my implementations relative to the Caffe on an Intel Core i7 6700K system with 4 cores and 8 hyperthreads. I disabled Turbo Boost and dynamic frequency scaling so that the processor ran at a stable 4.0 GHz during the tests. Caffe was built with an upstream version of OpenBLAS; I also considered Caffe configuration with Intel MKL, but an OpenBLAS version delivered better performance[1]. The performance of my implementation was measured in standalone benchmarks with cached data evicted between runs. Both Caffe and my implementations were benchmarked in multi-threaded mode with 8 threads.

On VGG model A and AlexNet, convolution based on a 16×16 FFT delivers the best performance. On the layers of Overfeat with a 3×3 kernel, the algorithm based on a Winograd transform delivers the best performance. In all cases, an 8×8 FFT performs worse than 8×8 Winograd, which is reasonable: the FFT is more

[1]This is probably a performance bug in Caffe. In my standalone SGEMM benchmarks Intel MKL is generally faster than OpenBLAS.

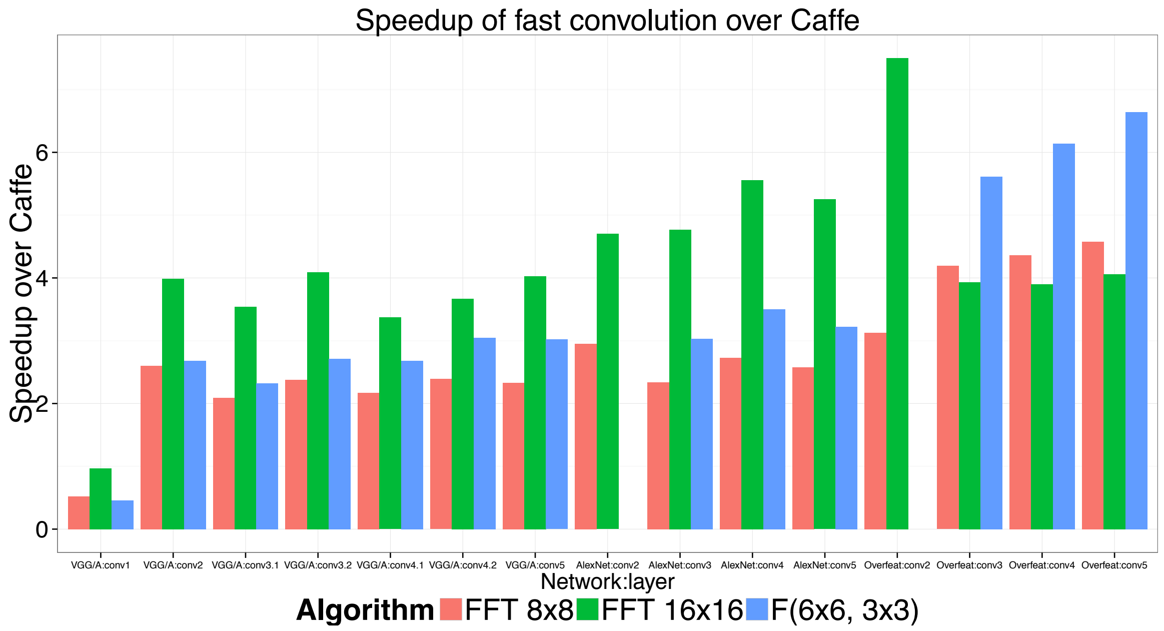


Figure 26: Speedup of my implementations against Caffe

computationally expensive than a Winograd transform and requires complex matrix multiplication to reduce transformed coefficients, which is up to twice slower than real matrix multiplication after Winograd transform. The key predictor of whether 16×16 FFT or $F(6 \times 6, 3 \times 3)$ Winograd transform would deliver better performance is the number of tiles required to cover the input image. The number of tiles can be computed as

$$n = \left\lceil \frac{H_\mathrm{i} + 2P_\mathrm{i} - H_\mathrm{k} + 1}{H_\mathrm{t} - H_\mathrm{k} + 1} \cdot \frac{W_\mathrm{i} + 2P_\mathrm{i} - W_\mathrm{k} + 1}{W_\mathrm{t} - W_\mathrm{k} + 1} \right\rceil$$

The last two columns of Table 7 specify the number of 8×8 and 16×16 tiles for the layers. A single 16×16 tile has the same number of elements as four 8×8 tiles. When the number of 8×8 tiles is less than $4\times$ the number of 16×16 tiles, the Winograd transform and the 16×16 FFT produce the same number of output elements, and the Winograd transform-based convolution wins due to doing a real matrix-matrix multiplication instead of a complex matrix-matrix multiplication for the FFT coefficients.

One may question if the efficiency of the Winograd transform can be further improved by using larger tiles. While it is mathematically possible to derive Winograd transforms for larger tiles, Lavin [58] argues that this rapidly increases the magnitude of coefficients in the transformation matrix, resulting in the accuracy loss of the convolution. I observed that convolutions with $F(6 \times 6, 3 \times 3)$ lose about half of their significant bits, and it is substantially less accurate than fast convolution based on Fourier transforms.

Besides benchmarking performance of the whole convolution, I also present performance results for the two components of the fast convolution algorithm: my 2D FFT and 2D Winograd transform implementations, and matrix multiplication.

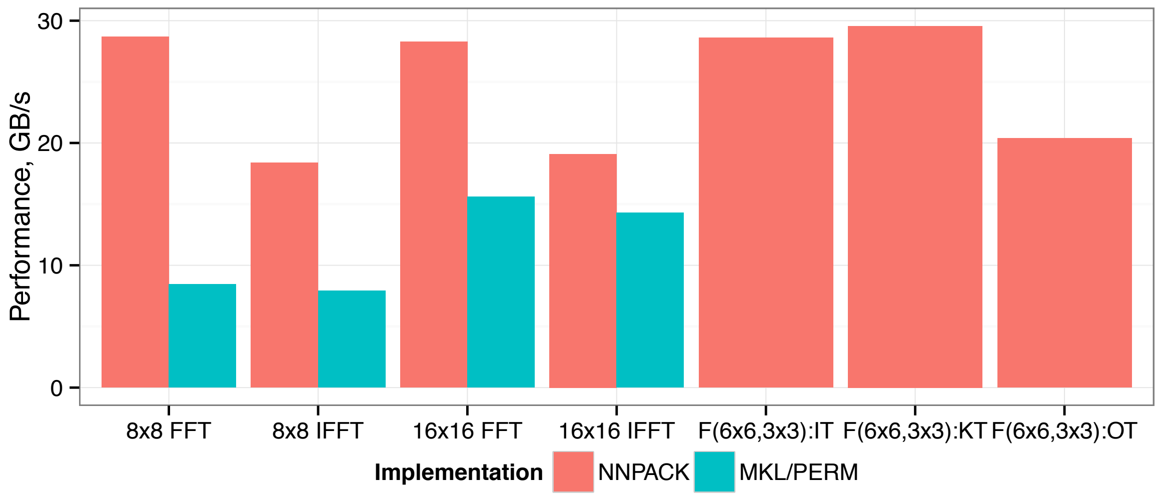


Figure 27: Performance of batch transforms. The custom 2D FFT implementation specialized for 8×8 and 16×16 blocks substantially outperform the more generic implementations in Intel MKL. Both the 2D FFT and the 2D Winograd transforms demonstrate performance close to the memory bandwidth peak of the system.

Fig. 27 presents performance of multi-threaded batch 2D FFT and 2D Winograd transforms with my implementation and Intel MKL 11.3 Update 1. I display performance in gigabytes per second to underline that my implementation saturates memory throughput: both 8×8 and 16×16 FFT reach approximately the same levels

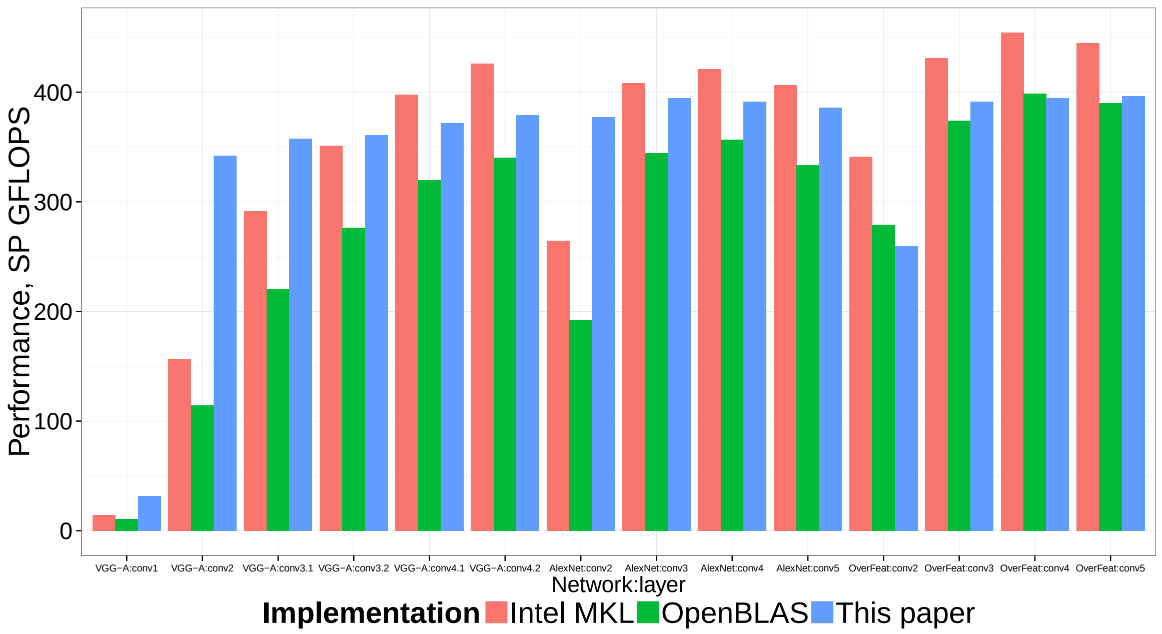


Figure 28: Performance of complex matrix multiplication. For my implementation I report performance of reduction of blocks of 8×8 FFT coefficients. For MKL and OpenBLAS with report CGEMM performance with $M = B, N = C_o, K = C_i$

of memory bandwidth. My implementation outperforms Intel MKL by $1.8 - 3.4\times$ on forward 2D FFT and by $1.3 - 2.3\times$ on inverse FFT. Lower performance of my implementation on inverse transforms is due to inability to use streaming stores: I use masked store to save a subset of output samples, resulting in excessive memory traffic. Winograd transform saturates memory channels just like 2D FFT, albeit delivers marginally better performance.

Fig. 28 illustrates multi-core performance of reduction of blocks of FFT coefficients. For my implementation, the performance is measured directly. For Intel MKL and OpenBLAS libraries, I measure performance of CGEMM calls with the same parameters as could be used for reduction of the FFT blocks. The figure suggests that despite the fact that my implementation operates on interleaved matrices with resulting lower arithmetic intensity, it can attain performance competitive with these highly optimized libraries.

3.6 Current Limitations and Future Work

One limitation of my implementation is its lack of support for strided convolutions. Strided convolutions could be enabled with a minor modification of inverse transform function: it could write out only the valid elements. Such implementation would be quite inefficient, because it would throw out most computed outputs, but since strided convolutions are normally used with very large kernel sizes (e.g. 11×11 in AlexNet), it could still outperform the naive implementation. A better approach would be to use the efficient method of computing strided convolutions with Fast Fourier Transform suggest by Brosch and Roger [8].

Table 8: Transformation costs, measured through FP_ARITH_INST_RETIRED hardware counter

Transformation	FLOPs
Forward FFT 8×8	912
Forward FFT 16×16	4056
Inverse FFT 8×8	992
Inverse FFT 16×16	4480
Input $F(6 \times 6, 3 \times 3)$	736
Kernel $F(6 \times 6, 3 \times 3)$	304
Output $F(6 \times 6, 3 \times 3)$	480

3.7 Conclusion

I demonstrated that training convolutional layers of Imagenet-winning models on CPUs can be accelerated up to 7.5× through a combination of fast convolution algorithms and highly tuned implementations. These substantial speedups are achieved by a combination of several techniques:

- High-performance 2D FFT, based on both decimation-in-time and decimation-in-frequency radix-2 Cooley-Tukey algorithm, complex-to-complex, real-to-complex, and dual-sequence real-to-complex 1D FFT. The resulting 2D FFT implementation outperforms Intel MKL by $1.3-3.4$ times and saturates memory bandwidth on Intel Skylake system.

- High-performance 2D Winograd transform, which delivers even higher throughput than the 2D FFT.

- Custom SGEMM- and CGEMM-like kernels, which tightly integrate with transformation code: the transformation routines repack outputs for better GEMM efficiency, and GEMM work with interleaved matrices to make memory access pattern in the transformation routines friendlier to memory subsystem.

CHAPTER IV

PRIMITIVES FOR LOW-INTENSITY CONVOLUTIONS

4.1 Introduction

Early neural network architectures for computer vision utilized convolutions with large kernel sizes as the primarly building block. 3x3 convolutions were particularly popular: VGG and ResNet feature extractors were a common building block for image classification, object detection, and style transfer, while U-Net was the high-accuracy architecture for segmentation, and all of these architecture spend the majority of FLOPs in 3x3 convolutions. For 3x3 convolutions, algorithmic acceleration with Winograd- and Fourier-transform-based algorithms is the key to efficient inference. However, in recent years, new types of convolutions emerge as common building blocks, and require different methods.

Depthwise separable convolutions, popularized by Xception and MobileNet architectures, gain popularity as the basic building blocks of computer vision architectures. Depthwise separable convolutions can be thought of as factorizations of traditional convolution with large kernel size into two convolutions: pixelwise convolution (convolution with 1x1 kernel), followed by a depthwise convolution. Whereas traditional convolution recombines data both from different channels and from neighbouring spatial locations, depthwise separable convolution factorize this transformation into separate recombination across different channels (in the 1x1 convolution) and recombination across neighbouring spatial locations (in the depthwise convolution).

[0]This chapter is based on the single-author workshop paper [23] with minor modifications: "Marat Dukhan. The Indirect Convolution Algorithm. Presented on the Efficient Deep Learning for Computer Vision workshop, June 2019."

The components of depthwise separable convolutions exhibit much lower arithmetic intensity, and are less susceptible to acceleration through fast convolution algorithms. For pixelwise convolutions, fast convolution algorithms bring nothing at all, and for depthwise convolutions the advantage of convolution in Fourier or Winograd space is eroded by inability to amortize the cost of transformations across either input or output channels. These types of convolutions, which I denote low-intensity convolutions, require a different approach to optimization. As they stress the memory subsystem more than computational units, algorithmic optimizations have to focus on minimizing memory operations, in particular the memory layout transformations in the im2col and col2im patch-building algorithms.

In this section I present a novel type of algorithm for Convolution computation, named the **Indirect Convolution algorithm**. The Indirect Convolution algorithm is a modification of GEMM-based algorithms, and like GEMM-based algorithms it can efficiently support arbitrary Convolution parameters, and leverage the vast trove of research on high-performance GEMM implementation. Additionally, the Indirect Convolution algorithm has two major advantages over GEMM-based algorithms:

- The Indirect Convolution algorithm **eliminates expensive and memory-intensive im2col transformations**. Elimination of im2col transformations improves performance by up to 62% compared to GEMM-based algorithms. The performance improvement is particularly prominent on Convolutions with small number of output channels, when im2col comprises a large share of Convolution runtime.
- The Indirect Convolution algorithm allows to **replace the im2col buffer with a much smaller indirection buffer**. The size of im2col buffer scales linearly with the number of input channels, but the size of indirection buffer does not depend on the number of input channels. Thus, the memory footprint advantage of Indirect Convolution algorithm is the greatest for Convolutions with many

input channels.

4.1.1 Limitations

The high efficiency of the Indirect Convolution algorithm is contingent on certain conditions:

- The algorithm is optimized for the **NHWC layout**, supported by TensorFlow [1], TensorFlow Lite, and Caffe2 frameworks. While the algorithm can be adapted to work in the NCHW layout (native to PyTorch [76] and Caffe [53] frameworks), I don't expect it would be competitive with the state-of-the-art patch-building algorithms of Andersen et al [100] due to its strided memory access in the NCHW layout.
- The algorithm is optimized for the forward pass of the Convolution operator, and has **limited applicability to the backward pass of Convolution operator and to the Transposed Convolution operator**. The Indirect Convolution algorithm presented in this paper replaces the GEMM-based algorithms using im2col or im2row transformations. However, for the backward pass of a strided Convolution operator and for the strided Transposed Convolution operator, col2im and row2im-based algorithms are more optimal due to smaller number of arithmetic operations.
- Similarly to the GEMM-based convolution, the **Indirect Convolution algorithm is not efficient for depthwise convolutions**. Depthwise convolutions [14] independently convolve each channel with its own set of filters.

4.2 The Indirect Convolution algorithm

The Indirect Convolution algorithm can be represented as a modification of GEMM-based algorithms. Where GEMM-based algorithms reshuffle data to fit it into the GEMM interface, the Indirect Convolution Algorithm instead modifies the GEMM

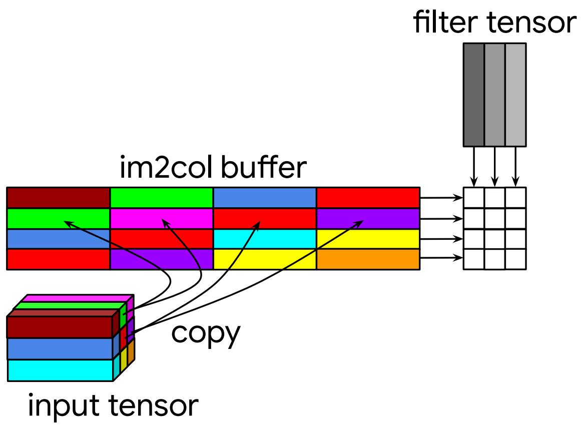


Figure 29: GEMM operation as a component of GEMM-based convolution algorithm. im2col buffer represents matrix A, filter tensor - matrix B, and their product constitutes the output tensor.

primitive to adopt it to the original data layout. The modified GEMM primitive, denoted the **Indirect GEMM** in this paper, has a similar computational structure as the standard GEMM, and can reuse the same optimizations.

4.2.1 GEMM Primitive

For an $M \times K$ matrix A (optionally transposed), $K \times N$ matrix B (optionally transposed), and $M \times N$ matrix C, and scalar constants α and β, the GEMM primitive computes

$$C \leftarrow \alpha A \times B + \beta C$$

In the context of the forward pass of a Convolution operator, A contains input tensor data, B the constant filter data, and C represents output tensor data. In the traditional im2col+GEMM algorithm, $\alpha = 1$, and $\beta = 0$, albeit newer low-memory GEMM-based algorithms [5] make use of $\beta = 1$ case as well. Fig. 29 illustrates the role of GEMM primitive in GEMM-based convolution algorithm, and Listing 4.1 demonstrates the basic building block of a GEMM primitive – a GEMM micro-kernel that produce 2 rows and 2 columns of matrix C.

Listing 4.1: Implementation of GEMM micro-kernel in C

```
void uGEMM(
 int k, const float* pw,
 const float* pa, int lda,
 float* pc, int ldc)
{
 const float* pa0 = pa;
 const float* pa1 = pa + lda;
 float c00 = 0, c01 = 0, c10 = 0, c11 = 0;
 do {
  const float a0 = *pa0++, a1 = *pa1++;
  const float b0 = *pw++;
  c00 += a0 * b0;
  c10 += a1 * b0;
  const float b1 = *pw++;
  c01 += a0 * b1;
  c11 += a1 * b1;
 } while (--k != 0);
 pc[0] = c00; pc[1] = c01;
 pc += ldc;
 pc[0] = c10; pc[1] = c11;
}
```

4.2.2 From GEMM to Indirect GEMM

I suggest two modifications that jointly make the GEMM primitive directly suitable for the convolution implementation:

1. Removing the assumption that rows of matrix A are separated in memory by a constant stride. Instead, pointers to rows of matrix A are loaded from an array of pointers provided by the caller and denoted **indirection buffer**. In

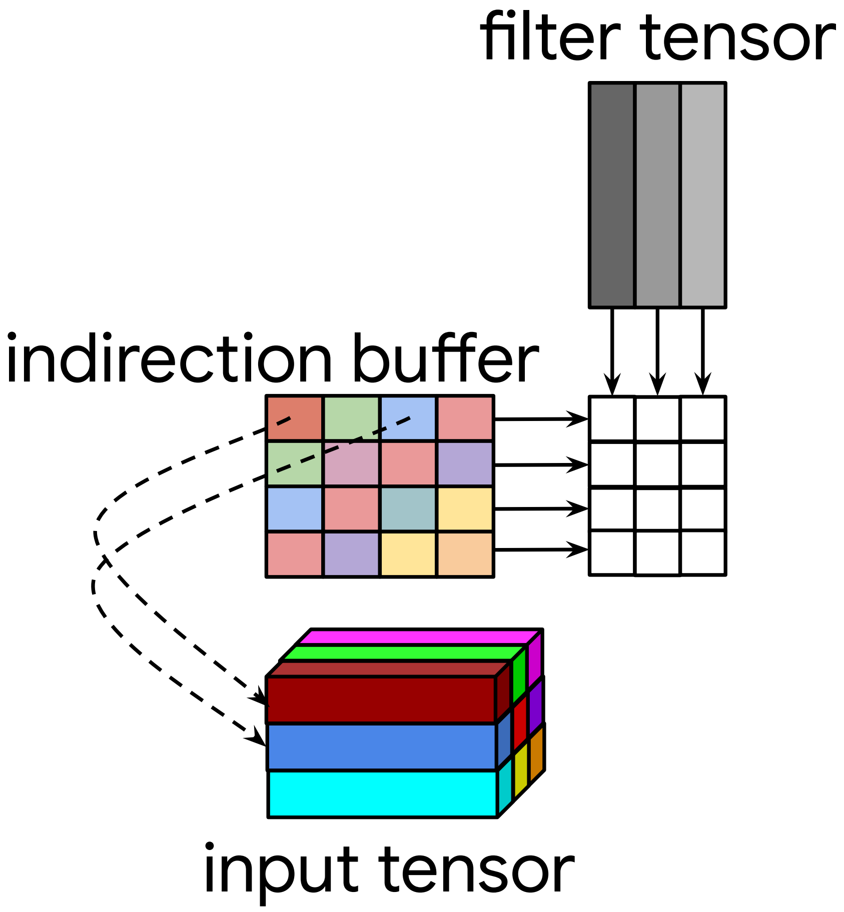


Figure 30: Indirect GEMM operation as a component of Indirect Convolution algorithm. The indirection buffer contains only pointers to rows of the input tensor, and the Indirect GEMM operation reads rows of data directly from the input tensor.

the context of a 2D convolution, the indirection buffer specifies the address of a row of pixels in the input tensor that contribute to the computation of the output pixel.

2. Secondly, I add an extra loop over elements of the kernel. For each iteration of this loop, the modified GEMM primitive loads new pointers to input rows from the indirection buffer, computes dot products of K elements specified by these pointers with the filter data, and accumulates the results of the dot product with results of the previous loop iterations.

These modifications enable Indirect Convolution algorithm to implement a fused im2col + GEMM operation, but without ever storing the result of im2col operation in memory. Fig. 30 illustrates the data flow in Indirect GEMM primitive and Listing 4.2 provides an example of an Indirect GEMM micro-kernel.

Listing 4.2: Implementation of Indirect GEMM micro-kernel in C

```c
void uIndirectGEMM(
 int n, int k, const float* pw,
 const float** ppa, int lda,
 float* pc, int ldc)
{
 float c00 = 0, c01 = 0, c10 = 0, c11 = 0;
 do {
  const float *pa0 = *ppa++;
  const float *pa1 = *ppa++;
  int kk = k;
  do {
   const float a0 = *pa0++, a1 = *pa1++;
   const float b0 = *pw++;
   c00 += a0 * b0;
   c10 += a1 * b0;
   const float b1 = *pw++;
   c01 += a0 * b1;
   c11 += a1 * b1;
  } while (--kk != 0);
 } while (--n != 0);
 pc[0] = c00; pc[1] = c01;
 pc += ldc;
 pc[0] = c10; pc[1] = c11;
}
```

4.2.3 Indirection Buffer

The Indirection buffer is a buffer of pointers to rows of input pixels. Each row has C pixels, and the rows can optionally be strided. For each output pixel position and for each kernel element the indirection buffer contains a pointer to a row of input pixels that would be convolved with a row of the filter weights for the corresponding kernel element to produce the corresponding output pixel.

It is common to use an implicit padding for convolutions with non-unit kernels. In convolutions with an implicit padding, the input tensor is implicitly padded with zeros along the spatial dimensions before computing convolution. To handle the padded convolution, the Indirect Convolution algorithm requires an explicit zero vector - a constant vector with C elements initialized to zeros. The explicit zero vector does not need to be contigious with the input tensor, and can even be shared between multiple convolution operators. During an initializing of the indirection buffer, pointers to input rows which fall outside of the input tensor range are replaced with pointers to the explicit zero vector.

The Indirection buffer depends on several parameters: shapes of input, output, and filter tensors, convolution stride, dilation, and implicit padding, and pointers to input tensor and explicit zero tensor, and stride of pixel rows in the input tensors. These parameters can be categorized into several groups, according to the frequency of their change and implications of their change on indirection buffer:

- Convolution stride, dilation, kernel size, implicit padding, number of input channels, and output channels are parameters of a neural network model and once the model is instantiated, they are practically immutable.
- Changes in the height and width of input or output tensors require a complete reinitialization of indirection buffer. However, for most types of models, and in particular in the production environment, such changes are rare.

- Changes in the batch size require a partial reinitialization of the indirection buffer only for batch indices that were not previously initialized.
- Changes in pointers to the input tensor or the explicit zero vector require a complete reinitialization of indirection buffer. To avoid the cost, a high-level framework implementing the convolution can guarantee that in the absence of shape changes, tensors have persistent location.

4.3 Experimental Evaluation

Four factors affect performance of the Indirect Convolution compared to the GEMM-based convolution algorithms:

1. Elimination of the *im2col* transformation for non-unit convolutions.
2. Improved caching of the input rows for convolutions with large kernels as Indirect GEMM reads input rows contributing to different output pixels from the same location while GEMM would read these input rows from different locations in the im2col buffer.
3. Overhead of loading pointers to rows of the input data from the indirection buffer compared to computing them under a constant stride assumption.
4. Potentially lower efficiency of two nested loops with $R \times S$ and C iterations in the Indirect GEMM operation compared to a single loop with $R \times S \times C$ iterations in the GEMM operation.

These factors are closely coupled together, but I can separately benchmark the effect of the first two (which positively affect the Indirect Convolution algorithm performance) and the last two (with negative effect on performance) by benchmarking three variants of convolution implementation:

- The Indirect Convolution algorithm

Table 9: Characteristics of mobile devices in performance evaluation. Microarchitecture (uArch) is specified for the big cores.

Device	Chipset	uArch
Samsung Galaxy S8	Exynos 8895	Exynos-M2
Google Pixel 2 XL	Snapdragon 835	Cortex-A73
Google Pixel 3	Snapdragon 845	Cortex-A75

- The traditional GEMM-based Algorithm. Unless the convolution uses 1x1 kernel and unit stride, this algorithm involves the im2col transformation.
- The GEMM part of the traditional GEMM-based algorithm. This benchmark excludes the im2col transformation, and therefore does not produce the correct result. I include it only as a way to separate the effect of different factors on performance.

4.3.1 Experimental Setup

Platforms I evaluated performance on three ARM64 Android devices with characteristics listed in Table 9. The processors in these mobile devices include two types of cores: high-performance (big) cores and low-power (little) cores. In my experiments, all benchmarks were run in single-threaded mode with thread pinning to a single big core.

Implementation I use highly optimized implementations of GEMM and the Indirect GEMM micro-kernels in ARM64 assembly with software pipelining for out-of-order cores. Both micro-kernels produce 4x8 output tile (i.e. 4 output pixels with 8 output channels each), and use exactly the same instruction sequence in the inner loop of the Indirect GEMM micro-kernel and the main loop of the GEMM micro-kernel.

Unlike many other GEMM implementations which use the Goto algorithm [33], I do not repack panels of matrix A accessed in a micro-kernel into a contiguous memory region. Goto and Van de Geijn [33] suggested repacking as a way to overcome limited cache associativity. In contrast, I find that with GEMM matrices that typically occur in neural network architectures, limited cache associativity is not a concern because whole panels of the A and B matrices read by the micro-kernel fit into level-1 cache. Note that avoiding repacking of matrix A in GEMM and Indirect GEMM primitives is my implementation detail, and both GEMM-based algorithm and the Indirect Convolution algorithm can be implemented either with or without repacking of inputs. However, my implementations of GEMM and Indirect GEMM micro-kernels assume that matrix B, which contains filter weights, is repacked into a contigious memory region, because filter weights never change at inference time, and such repacking can be done only once with no additional run-time cost.

Protocol I implement all micro-benchmarks on top of the Google Benchmark framework, which takes care of estimating sustained performance for the micro-benchmark. On top of it, each micro-benchmark is repeated 25 times. To bring measurements with different convolution parameters to a common scale, I compute resulting performance (in GFLOPS), and report median metric of the 25 runs, as well as 20% and 80% quantiles.

For each run of micro-benchmark I simulate the cache state during neural network inference: filter, bias, and output tensors, and indirection buffer are cleaned from cache, input tensor is prefetched into L1 cache, and the im2col buffer stays in cache between convolution invocations to represent the same im2col buffer space re-used between different convolution operators. The indirection buffer is initialized only once (outside of the benchmarked snippet), and reused across invocations of the Indirect Convolution algorithm.

Models I choose to evaluate performance on the convolution parameters of ResNet-18 [41] and SqueezeNet 1.0 [49] models. Unlike more recent mobile-optimized models like MobileNet v2 [82] and ShuffleNet v2 [64], which almost exclusively use 1x1 and depthwise convolutions, ResNet and SqueezeNet models employ a variety of convolution parameters, and provide a more balanced experimental workload. Both models

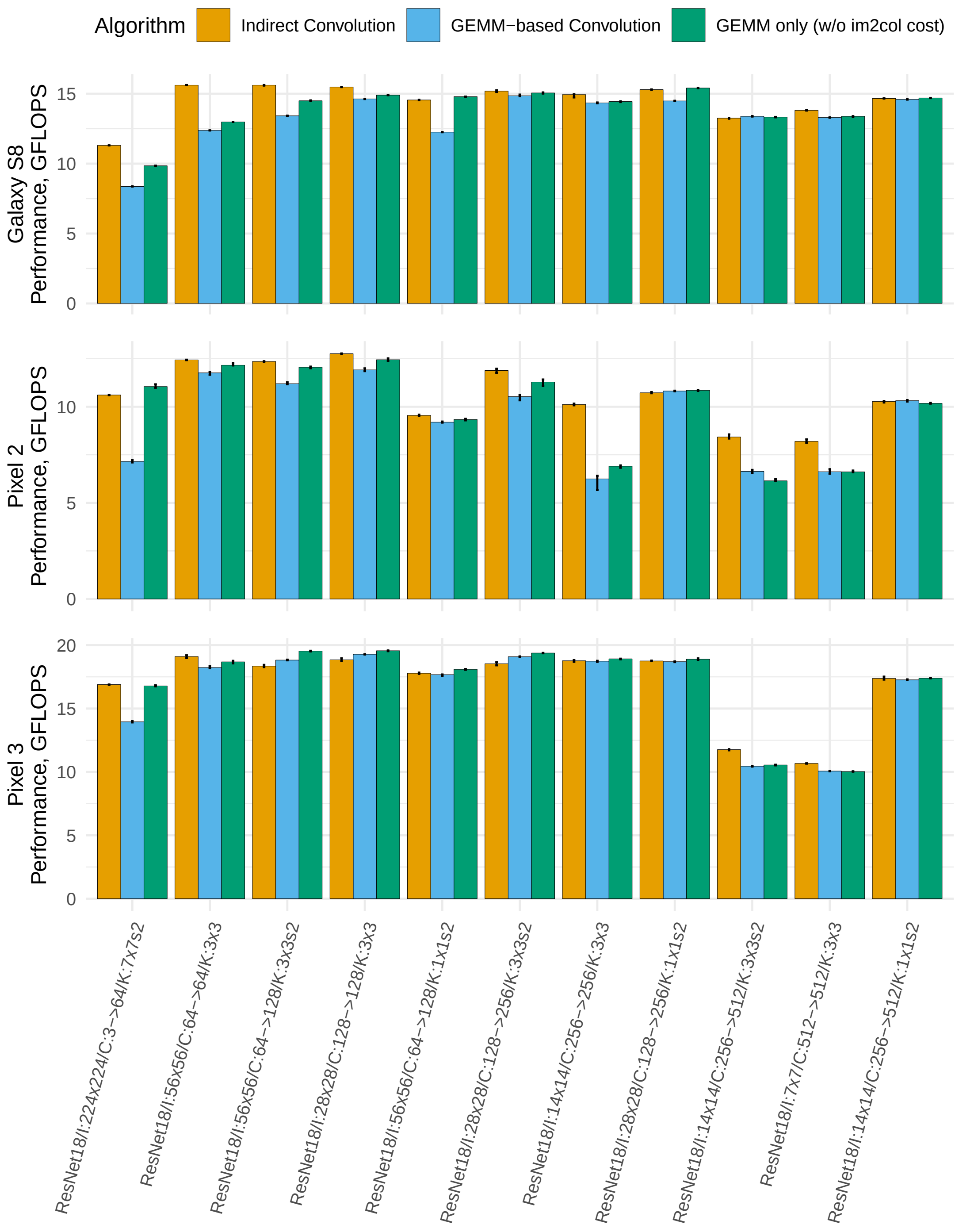


Figure 31: Performance of the Indirect Convolution algorithm and GEMM-based Algorithm on convolution operators of the ResNet-18 model. Opaque bars represent median performance across 25 runs. Error bars represent 20% and 80% quantiles.

start with a 7x7 stride-2 convolution, and then include 3x3 stride-1 convolutions. SqueezeNet additionally features 1x1 stride-1 convolutions and ResNet-18 makes use

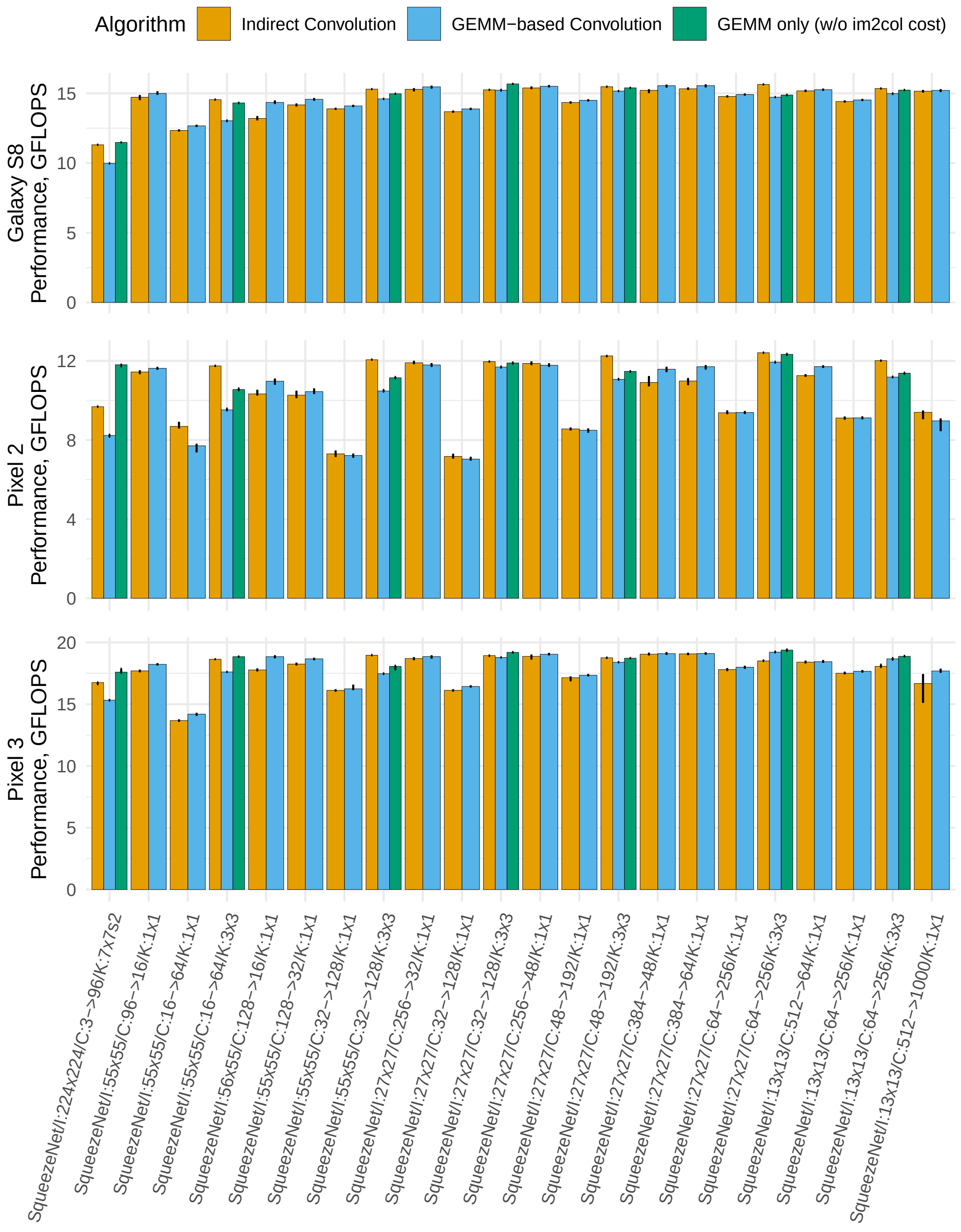


Figure 32: Performance of the Indirect Convolution algorithm and GEMM-based Algorithm on convolution operators of the SqueezeNet 1.0 model. Opaque bars represent median performance across 25 runs. Error bars represent 20% and 80% quantiles.

of 1x1 stride-2 and 3x3 stride-2 convolutions. Table 10 summarized the composition of the convolution parameters in both models.

Table 10: Types and count of Convolution operators in SqueezeNet 1.0 and ResNet-18 models. Convolutions with identical parameters are counted only once.

Convolution	ResNet-18	SqueezeNet 1.0
7x7 stride-2	1	1
3x3 stride-2	3	0
3x3 stride-1	4	6
1x1 stride-2	3	0
1x1 stride-1	0	15

Table 11: Geomean performance of modified GEMM primitive relative to standard GEMM primtive on 1x1 and non-1x1 Convolutions in ResNet-18 model.

Device	non-1x1	1x1 stride-2
Samsung Galaxy S8	+10.97%	+8.02%
Google Pixel 2 XL	+23.26%	+0.84%
Google Pixel 3	+4.31%	+0.51%

4.3.2 Experimental Results

Figs. 31 and 32 illustrate performance of the Indirect Convolution algorithm, the GEMM-based algorithm, and just the GEMM part of the GEMM-based algorithm on the ResNet-18 and SqueezeNet 1.0 models respectively. In 1x1 stride-1 Convolutions in the SqueezeNet model the GEMM-based algorithm directly call into GEMM primitive without using im2col transformation; for this reason, I do not separately show GEMM-only performance for these Convolutions.

These plots reveal that in most cases the Indirect GEMM has similar performance to the GEMM primitive post-im2col transformation; however, the addition of

Table 12: Geomean performance of modified GEMM primitive relative to standard GEMM primtive on 1x1 and non-1x1 Convolutions in SqueezeNet 1.0 model.

Device	non-1x1	1x1 stride 1
Samsung Galaxy S8	+5.70%	-1.84%
Google Pixel 2 XL	+11.29%	-0.25%
Google Pixel 3	+2.67%	-1.91%

the im2col transformation makes the Indirect Convolution algorithm outperform the GEMM-based Convolution on all Convolutions that involve im2col transformation. Tables 11 and 12 quantify this impact by types of convolution layers in the ResNet-18 and SqueezeNet 1.0 models. The Indirect Convolution algorithm has varying impact depending on the convolution parameters:

- Convolutions with larger than 1x1 kernels see the biggest impact, with major performance improvement in the $2.7 - 23.3\%$ range.
- 1x1 stride-2 convolutions, which similarly need im2col transformation in GEMM-based algorithms, but don't benefit from improved cache locality in the Indirect GEMM, see smaller improvements in the $0.5 - 8.0\%$ range.
- 1x1 stride-1 convolutions, where GEMM-based algorithms incur no im2col overhead, demonstrate a minor performance regression in the $0.3 - 1.9\%$ range, due to the extra complexity of the Indirect GEMM primitive compared to the tranditional GEMM.

These results suggest that the Indirect Convolution algorithm can provide substantial improvement for convolutions which involve im2col transformation in GEMM-based algorithms. However, the GEMM primitive has a small edge on 1x1 stride-1 convolutions, which do not involve the im2col transformation, and for best overall performance it is beneficial to switch between the GEMM or Indirect GEMM primitives depending on convolution parameters.

4.4 Analysis

The roofline model [96] provides a convenient analysis tool for predicting which parameters affect the relative performance of the Indirect Convolution algorithm and GEMM-based algorithms. For the analysis, I denote output height and width as H_{out} and W_{out}, input and output channels as C and K, and kernel height and width as R and S.

With the above notation, both GEMM and the Indirect GEMM algorithms involve $K \times C \times H_out \times W_out \times R \times S$ compute-bound operations (FLOPs)[1]. GEMM-based convolution with patch-building transformation additionally incurs $2 \times C \times H_{out} \times W_{out} \times R \times S$ memory operations. Assuming a system's arithmetic intensity (ratio of FLOPs to memory loads in balanced code) λ, then one does $(K + 2\lambda) \times C \times H_{out} \times W_{out} \times R \times S$ FLOPs-equivalent operations in GEMM-based convolution and the speedup of Indirect Convolution algorithm is $1 + \frac{2\lambda}{K}$. Thus, the Indirect Convolution algorithm is the most beneficial when the number of output channels is small. The upper bound on speedup is $1 + 2\lambda$, and suggests that as the systems' arithmetic intensity continues to grow with each generation, so will the advantage of the Indirect Convolution algorithm.

4.5 Conclusion

The Indirect Convolution algorithm is a modification of GEMM-based Convolution algorithms where the GEMM operation reads addresses of rows in the input tensor from indirection buffer. Experiments revealed that this modified GEMM-like operation has similar performance as the traditional GEMM operation, and suggested that the major differences between the two types of algorithms stem from the difference between the im2col buffer in GEMM-based algorithms and the indirection buffer in the Indirect Convolution Algorithm. Unlike im2col buffer in GEMM-based algorithms, the Indirection buffer is constant in the number of input channels, and can persist between convolution invocations.

The Indirect Convolution algorithm offers the universality of the GEMM-based algorithm, but with a smaller memory footprint and the elimination of the im2col transformation cost. These characteristics make the Indirect Convolution algorithm a viable option for default implementation of the convolution operator.

[1] I can exclude memory operations in GEMM and Indirect GEMM from the analysis, as these operations are almost always compute-bound.

The Indirect Convolution algorithm potentially has interesting performance characteristics beyond the scope of this paper. In particular, the algorithm may have additional performance advantage over GEMM-based algorithms during multi-threaded convolution invocation. The modified GEMM operation is compute-bound, and should scale linearly with the number of cores, while im2col-component of GEMM-based convolution would be saturated by memory or cache bandwidth, which has sublinear scaling in number of cores. Exploring these other dimensions of performance is a topic for further research.

CHAPTER V

PRIMITIVES FOR SPARSE CONVOLUTIONS

Neural networks are overparametrized machine learning models and have substantial redundancy in their trained parameters. Reducing such redundancy poses an opportunity to reduce the size of neural network models and improve their inference performance, and is long-standing research subject in the deep learning community. Some of the methods proposed for addressing redundancy include low-rank decompositions [20], low-precision weights representations, including fixed-point quantization [52], ternary [102] and binary networks [48, 77], pruning [40] and bucketing [38] of weights. Pruning, or sparsification, of weights works by either shrinking low-magnitude weights to zero and modifying the training procedure to guarantee that the weights stay at zero during further weight updates or by adding L1 loss on the weights. Unlike other methods that eliminate redundancy in neural network models, moderate degrees of pruning do not sacrifice model accuracy, and can even improve it [39]. Importantly, this result holds not only on obviously overparametrized large architectures like ResNet but also on smaller mobile-optimized architectures [49].

Most research to date has focused on using sparsity in weights to reduce model size [38], and application of sparsity to inference acceleration has been limited to specialized hardware [37], the case of highly structured sparsity patterns [94], or the relatively simple case of the Fully Connected layers [38]. In this Chapter I demonstrate that unstructured sparsity is practically useful for accelerating state-of-the-art convolutional neural networks for computer vision on general-purpose processors and

[0]This chapter is loosely based on the conference paper [25] with major revisions: "Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast Sparse Convnets. Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pages 14629–14638, 2020. " Only the parts where I am the main contributor are included in this Chapter.

discuss the primitives that make performance improvement possible.

5.1 *Sparse Inference*

I focus on the case of CNN architectures based on depthwise separarable convolutions, i.e., a combination of pixelwise convolutions (convolutions with a 1x1 kernel) and depthwise convolutions, which are mainstream in efficiency-optimized architectures [46, 64, 89]. Pixelwise convolutions account for nearly all computations in such architectures and directly map to the GEMM primitive from BLAS libraries in the traditional inference implementation. The key idea of sparse inference is to induce a large number of zeroes in the pixelwise convolution weights during training and replace dense matrix-matrix multiplication (GEMM) with sparse matrix-dense matrix multiplication (SpMM). There are several differences between the sparse matrix-dense matrix multiplication in sparse pixelwise convolution and that of traditional scientific computing applications:

- While many sparse matrices resulting from scientific computing problems have identifiable high-level sparsity structure that specialized SpMM implementations can detect and optimize for, sparse matrices resulting from neural network training have no apparent sparsity structure and look completely stochastic.
- Sparse matrices in scientific computing represent the constraints of real-world data or the scientific algorithm, and while it is possible to rearrange rows and columns for better efficiency, the mathematical pattern of sparsity structure is fixed. By constrast, sparse matrices resulting from neural network inference training are learned and can be modified by adjusting the training procedure. In particular, it is possible to induce block-sparse structure, but doing so comes at a cost: a neural network model with block-sparse weights achieves lower accuracy than the same-scale model with unstructured sparsity. To recover this accuracy loss one needs to train a larger model. Thus, there is a trade-off

between structured sparsity patterns and matrix sizes in sparse neural network inference.

- The sparse weights in the pixelwise convolutions never change during inference, and can be repacked into an optimal layout for SpMM computation with a one-time cost.

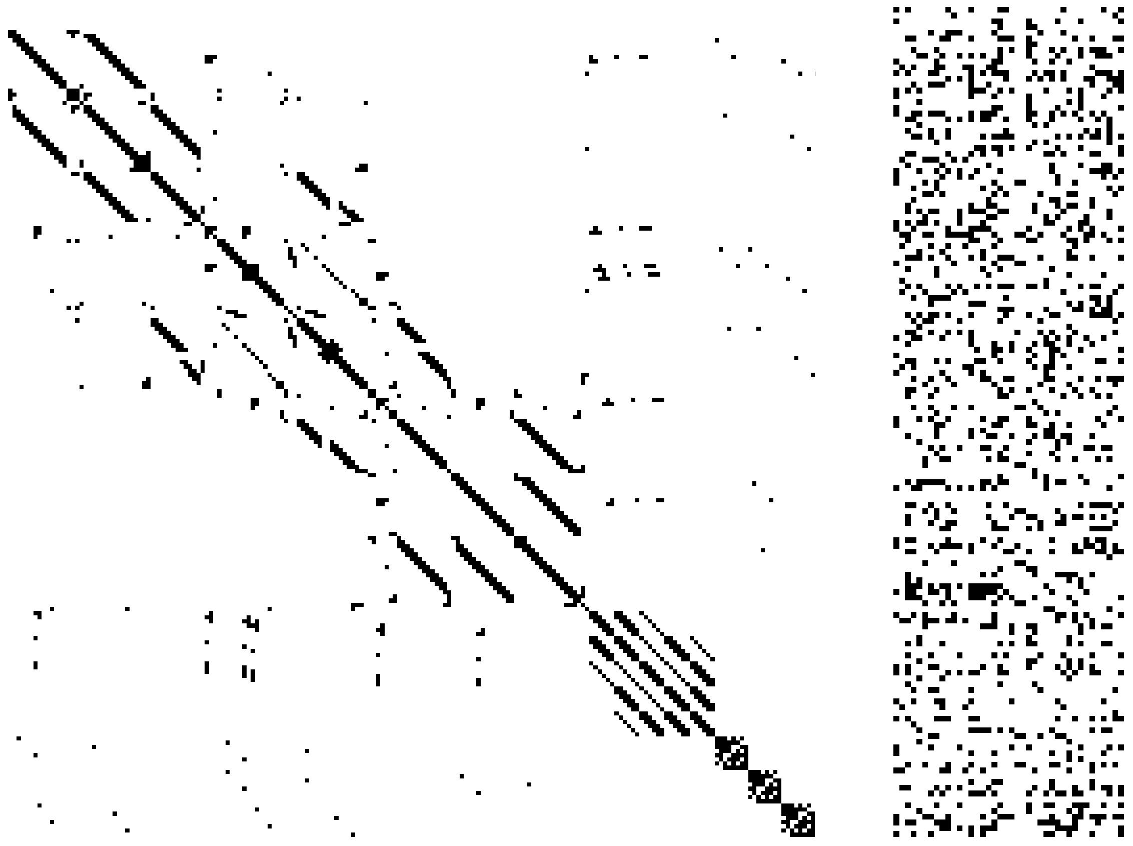


Figure 33: Examples of sparsity structure in a scientific computing problem and trained neural network weights. Left: matrix dwt_209 from SuiteSparse matrix collection [18]). Right: filter tensor of a pixelwise convolution in a sparse EfficientNet B0 model.

In a pixelwise convolution the filter tensor has only two non-degenerate dimensions corresponding to input channels and output channels. A coefficient in the filter tensor quantifies how much the input channel affects the output channel, and applies to every pixel: if the coefficient is non-zero, all input pixels of the corresponding input channel are multiplied by it and accumulated to all output pixels of the corresponding output channel, and if the coefficient is zero, the implementation can skip all pixels related to this pair of input and output channels. To make operations on groups of pixels efficient, particularly on wide SIMD architectures, the pixels within the same channel should be stored contiguously in memory. Among the standard tensor layouts only the NCHW layout stores pixels contiguously, but most ML systems operate with NHWC layout as it is more efficient for dense inference with low-intensity neural network models (Chapter IV).

The dilemma of choice between NCHW layout for efficient sparse inference and NHWC layout for compatibility with existing machine learning systems can be solved through a combination of two techniques: **dynamic layout rewriting** and **mixed-layout convolutions**. Dynamic layout rewriting modifies the operator graph of a machine learning model by replacing NHWC-layout operators with NCHW-layout operators when the latter are expected to be more efficient, e.g., for sparse inference. However, when done naively, such layout rewriting would modify the layout of model inputs and outputs as well, causing it to become incompatible with expected inputs and outputs. Mixed-layout convolutions solve this problem by consuming tensors of one layout (e.g. NHWC) on input and producing tensors of another layout (e.g., NCHW) on output, thereby enabling implicit layout conversion at no cost.

Table 13: Parameters of the first convolution in mobile-optimized computer vision models.

Model	Kernel	Stride	Padding	Channels
MobileNet v1 [46]	3×3	2	Same	$3 \rightarrow 32$
MobileNet v2 [82]	3×3	2	Same	$3 \rightarrow 32$
MobileNet v3 Large [45]	3×3	2	Same	$3 \rightarrow 16$
ShuffleNet [101]	3×3	2	Same	$3 \rightarrow 24$
ShuffleNet v2 [64]	3×3	2	Same	$3 \rightarrow 24$
EfficientNet B0 [89]	3×3	2	Same	$3 \rightarrow 32$
ESPNet [69]	3×3	2	Same	$3 \rightarrow 16$
ESPNet v2 [70]	3×3	2	Same	$3 \rightarrow 32$

In theory, mixed-layout convolutions present a tremendous expansion in the number of combinations of convolution parameters, input and output tensor shapes, and input and output tensor layouts. In practice, however, for many types of computer vision models it suffices to have only the very first convolution to be mixed-layout to convert from the input NHWC tensor to an output NCHW tensor, and this very first convolution is surprisingly stable across many machine learning models, as seen in Table 13.

5.2 Constraints on Computer Vision models

Convolutional neural network models utilizing sparse inference are thus subject to a number of constraints:

- The model starts with a mixed-layout 3×3 stride-2 convolution that consumes input in NHWC layout and produces output in NCHW layout.
- The sparse part of the model operates in the NHWC layout.

- The last operation in the sparse part of the model either produces output in NCHW layouts, or aggregates across all pixels so that the output layout (NC) does not have (H) height and (W) width dimensions.

While the above restrictions are not compatible with all computer vision models, the main types of computer vision problems can be implemented within these constraints.

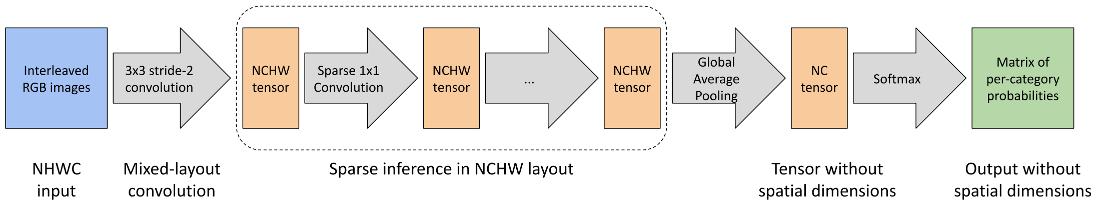


Figure 34: Illustration of sparse inference for image classification tasks.

Fig. 34 illustrates the architecture of a sparse image classification model. The image features produced by the last operation in the sparse part of the model are aggregated through global average pooling, which removes the distinction between NCHW and NHWC layouts, and normalized through a softmax operation to produce a matrix of per-class probabilities for each image in the batch.

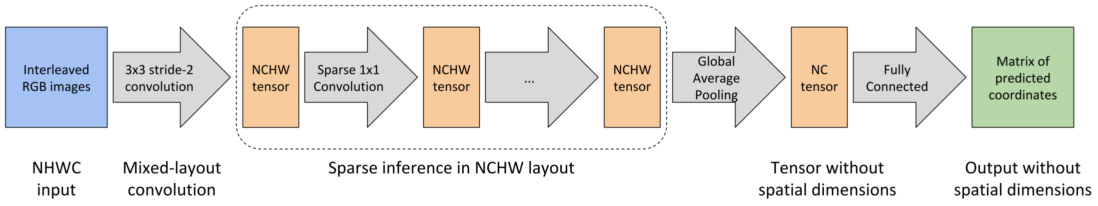


Figure 35: Illustration of sparse inference for object detection and other regression tasks.

Fig. 35 similarly illustrates the architecture of a sparse object detection model. The image features produced by the last operation in the sparse part of the model are aggregated through global average pooling and refined through a series of fully connected operators to produce a matrix of coordinates for the bounding boxes or keypoints. Importantly, as the output coordinates depend on all pixels of the input, this model works best for refining a crop of a single input object rather than the raw image with multiple objects in it.

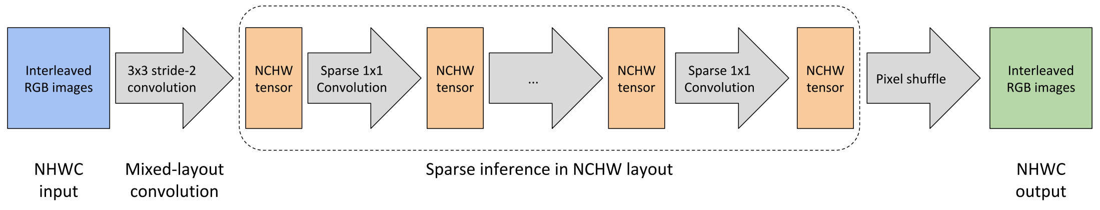


Figure 36: Illustration of sparse inference for segmentation and other dense prediction tasks.

Dense prediction tasks, such as segmentation, depth prediction, or generative adversarial networks (GANs), produce a 2D tensor of pixel values and present an additional challenge: unlike classification and regression tasks, there is no part of the network where the spatial dimensions are flattened and the tensor layout can be treated simultaneously as NCHW and NHWC. Instead, dense prediction models must produce output image in NHWC layout while internally processing data in NCHW for efficient sparse inference. The discrepancy between the internal NCHW and external NHWC layouts can be resolved in several ways:

- Explicitly transpose tensor data at the end of the sparse inference part of the model. However, this would add non-trivial overhead as dense prediction tasks operate with large feature maps in all parts of the model.

- Use mixed-layout deconvolution operation that consumes NCHW input and produce NHWC output. While this solution is possible in principle, it is challenging in practice, as different models use deconvolution with different parameters, and each would require a separate implementation.

- Use a mixed-layout resize operator and process an upscaled final image with a convolution operator [72]. However, computing the convolution operator on the upscaled image is significantly more expensive than upscaling image as a part of a deconvolution operator.

- Use mixed-layout subpixel convolutions [84], implemented via a pixelwise convolution followed by a mixed-layout pixel shuffle operation. This option is the most efficient for sparse inference, as pixelwise convolution can use sparse computations, and only the pixel shuffle operation needs to support mixed layout with NCHW input and NHWC output. As the pixel shuffle is intrinsically a transposition primitive, combining it with layout-changing transposition results in just another transposition operation.

Fig. 36 illustrates the architecture of a sparse segmentation model. The image features produced by the last pixelwise operation in the sparse part of the model are supplied to the mixed-layout pixel shuffle operation, which produces the final model's output in NHWC layout.

5.3 SpMM Primitive

SpMM, or sparse matrix-dense matrix multiplication, is the key primitive in sparse convolutional neural network inference. SpMM multiplies a dense matrix (input activations) by a sparse matrix (static weights) and produces a dense matrix with output activations. Unlike dense matrix-matrix multiplication (GEMM), SpMM organizes the computations in such a way that all zero elements of the sparse matrix are ignored and the runtime is proportional only to the number of non-zero weights. This feat is achieved by using special data structures for the sparse matrix, which store only non-zero values and sufficient metadata to recover the positions of these non-zero elements. In sparse pixelwise convolutions these data are non-zero weight elements and the corresponding input channel and output channel numbers.

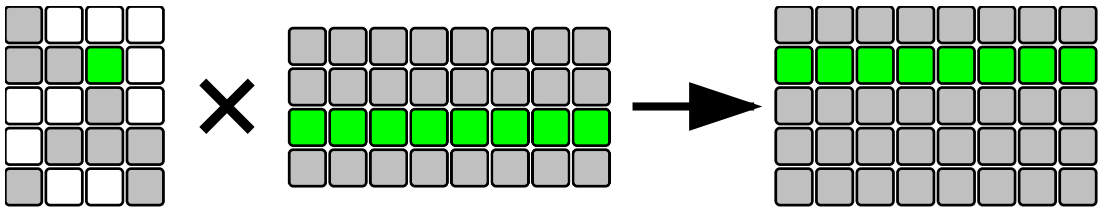


Figure 37: SpMM Primitive implemented using Read-Modify-Write microkernel.

The naive way to organize computations in the SpMM microkernel is to process each non-zero weight element one-by-one. This type of microkernel, illustrated in Fig. 37 and denoted as the Read-Modify-Write microkernel, loads all pixels from the input channel corresponding to a non-zero weight, multiplies them by the weight value, and accumulates the result to all pixels of the output channel corresponding to the weight. The Read-Modify-Write microkernel has the benefit of doing only contiguous memory loads and stores, at the cost of two reads and one write for every multiply-add operation. This low arithmetic intensity[1] makes this type of microkernel inefficient on most processor architectures.

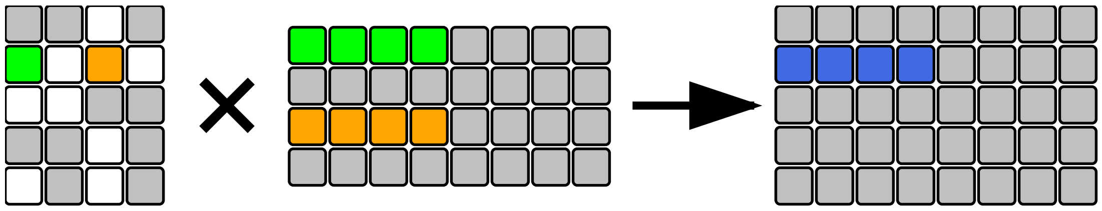


Figure 38: SpMM Primitive implemented using Read-Accumulate microkernel.

Fig. 38 illustrates a different type of microkernel, denoted a Read-Accumulate microkernel. This microkernel uses an in-register buffer to accumulate products of non-zero weights elements by the pixels of the correponding input channels before writing out the accumulated results. The inner loop of the Read-Accumulate microkernel contains only memory read and multiply-add operations. This instruction mix is similar to the inner kernel of dense matrix-matrix multiplication [33], which delivers

[1]high ratio of memory operations to arithmetic operations

near-peak performance on most modern processors. However, the Read-Accumulate microkernel comes with two drawbacks. First, the in-register accumulation buffer has a fixed size, so the microkernel processes only a fixed-size block of pixels in the inner loop and loads blocks of pixels corresponding to different input channels, resulting in non-contiguous memory access across the blocks of pixels, unlike the Read-Modify-Write microkernel where all pixels are read and updated contiguously. Secondly, the Read-Accumulate microkernel has an approximately 1:1 ratio of memory loads to multiply-accumulate operations, which makes it susceptible to the performance limits of the memory subsystem, especially when the loaded pixels are streamed from higher levels of the cache hierarchy.

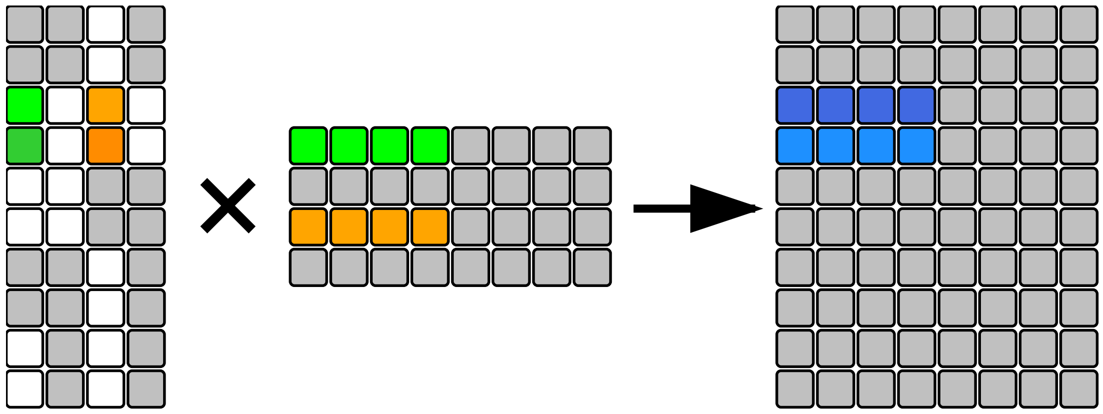


Figure 39: Block-SpMM Primitive implemented using Read-Accumulate microkernel.

Block-Sparse Matrix-Dense Matrix multiplication (Block-SpMM) is a modification of SpMM primitive that assumes blocked sparsity structure: non-zero elements are arranged into regular fixed-sized blocks, where all elements within a block are non-zero. In the case of sparse pixelwise convolutions, blocking in the output channels dimension is the most efficient as it enables reuse of loaded pixels for each input channel for updating multiple output channels. By comparison, blocking in the input channels dimension would reduce the amount of metadata to load for each non-zero weight, but would not enable any reuse of pixel data. Fig. 39 illustrates Read-Accumulate microkernel for Block-SpMM primitive with block size 2 in the output channels dimension. Each loaded input pixel is used to update two output pixels in adjacent output channels, resulting in an approximately 1:2 ratio of memory loads to multiply-accumulate operations. Blocked SpMM Read-Accumulate microkernels are even closer to GEMM microkernels in their instruction mix and can usually achieve near-peak performance.

5.4 Experimental Evaluation

5.4.1 Platform

I evaluate the performance impact of my sparse inference primitives on a Google Pixel 4a 5G mobile phone with Qualcomm Snapdragon 765G SoC. All experiments were performed for single-threaded execution, as using multiple cores tends to exceed the power constraints of the mobile device and leads to throttling and high performance variability.

5.4.2 Protocol

The performance benchmarks rely on the Google Benchmark framework to estimate sustained performance. I set the minimum run-time for each measurement to 1 second for SpMM benchmarks and 5 seconds for end-to-end model benchmarks. In the SpMM benchmarks I additionally repeat each measurement 25 times and record the median of the 25 runs, and simulate the cache state during neural network inference: the output vector is evicted from the cache before each iteration, but the input tensor stays in cache as long as it fits.

5.4.3 Pixelwise Convolutions

Figure 40 presents the performance of the GEMM (dense matrix-matrix multiplication), SpMM (sparse matrix-dense matrix multiplication), and BSpMM (SpMM with block sparsity structure illustrated in Fig. 39) on the matrix-matrix multiplication problems corresponding to pixelwise convolutions of the MobileNet v1 architecture [46]. The horizontal axis represents the sparsity, i.e., the share of zeroes in the weights matrix. The vertical axis represents effective GFLOPS: for dense matrix-matrix multiplication it match the actual GFLOPS metric achieved on this computation, while for sparse matrix-dense matrix multiplications effective GFLOPS represents how many GFLOPS a dense matrix-matrix multiplication would need to deliver to match the observed timings. For both sparse and dense matrix multiplications, effective GFLOPS are inversely proportional to wallclock runtime of the operation.

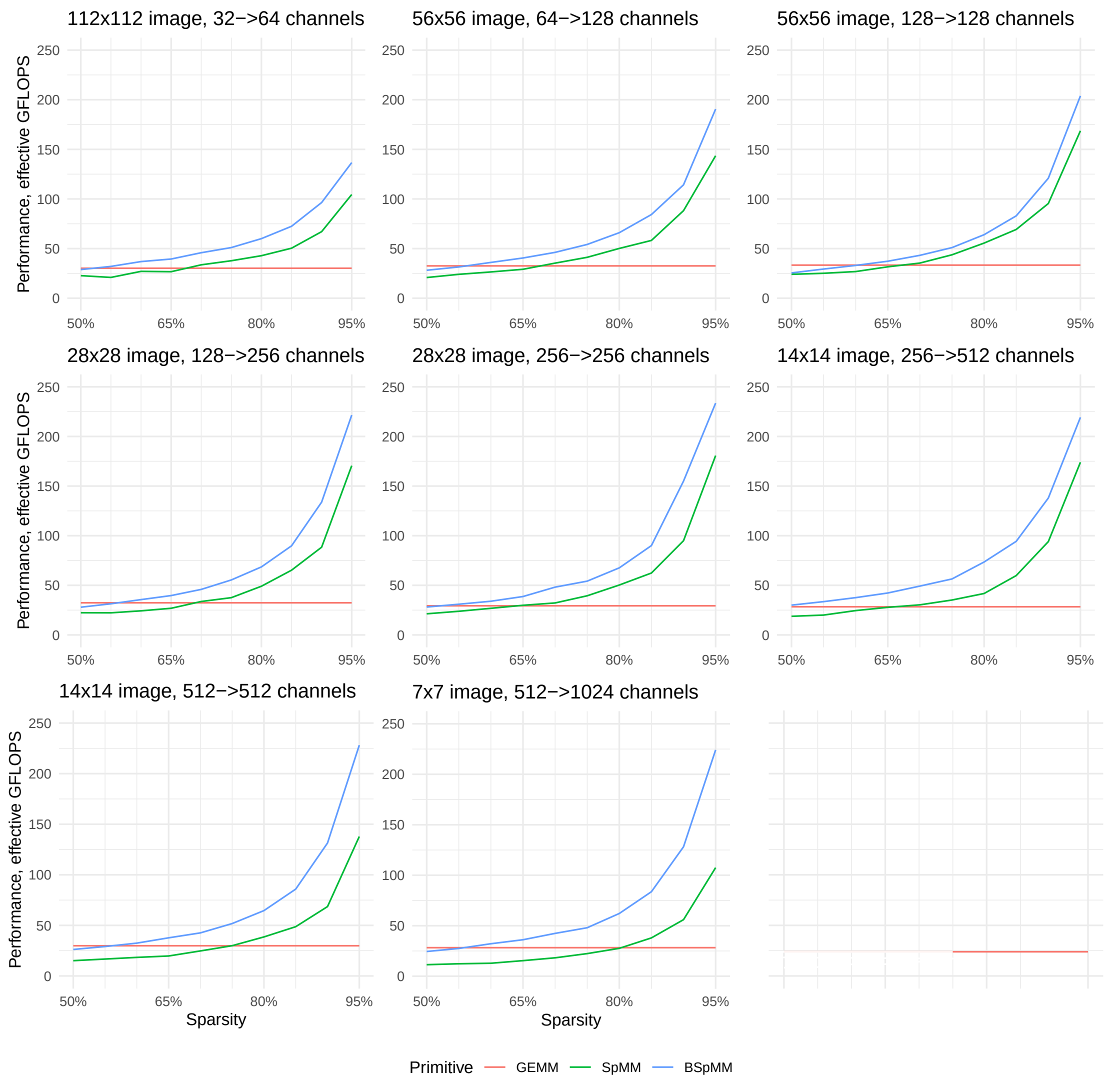


Figure 40: Performance of matrix multiplication problems corresponding to the pixelwise convolutions in the MobileNet v1 architecture.

The performance plots show that with minimal block sparsity structure, SpMM has comparable efficiency to GEMM at as little as 50% sparsity, and with unstructured sparsity SpMM breaks even with GEMM at $70 - 80\%$ sparsity. From the break-even point the advantage of SpMM grows inversely proportionally to the number of non-zeroes and at 95% sparsity reach over 6X for the the balanced matrix multiplication problems in the middle of MobileNet architecture and over 3X for the less balanced matrix multiplication problems in the beginning and in the end.

The advantage of the SpMM primitive over GEMM illustrated in Fig. 40 reinforces the idea of sparse pointwise convolutions as a means to accelerate inference in convolutional neural networks. However, it is by itself insufficient to fully validate this idea, as sparse inference also requires swapping other CNN operator for NCHW-layout or mixed-layout variants.

5.4.4 Randomized Models

To further validate the idea of acceleration through sparse neural network inference, I compare the latency of dense and unstructured sparse inference in MobileNet v1 [46], v2 [82], and v3 [45] architectures with varying sparsity in neural network weights. The values of the weights and sparsity patterns were randomized, but to my understanding it doesn't affect inference latency compared to the latency of models with trained weights and sparsity patterns. Both sparse and dense networks operate with inputs in NHWC layout and outputs in NC layout. However, dense inference uses NHWC layout internally while sparse inference employs a mixed-layout first convolution to convert the layout and do inference in the inner operators in NCHW layout, which favors efficient SpMM implementation.

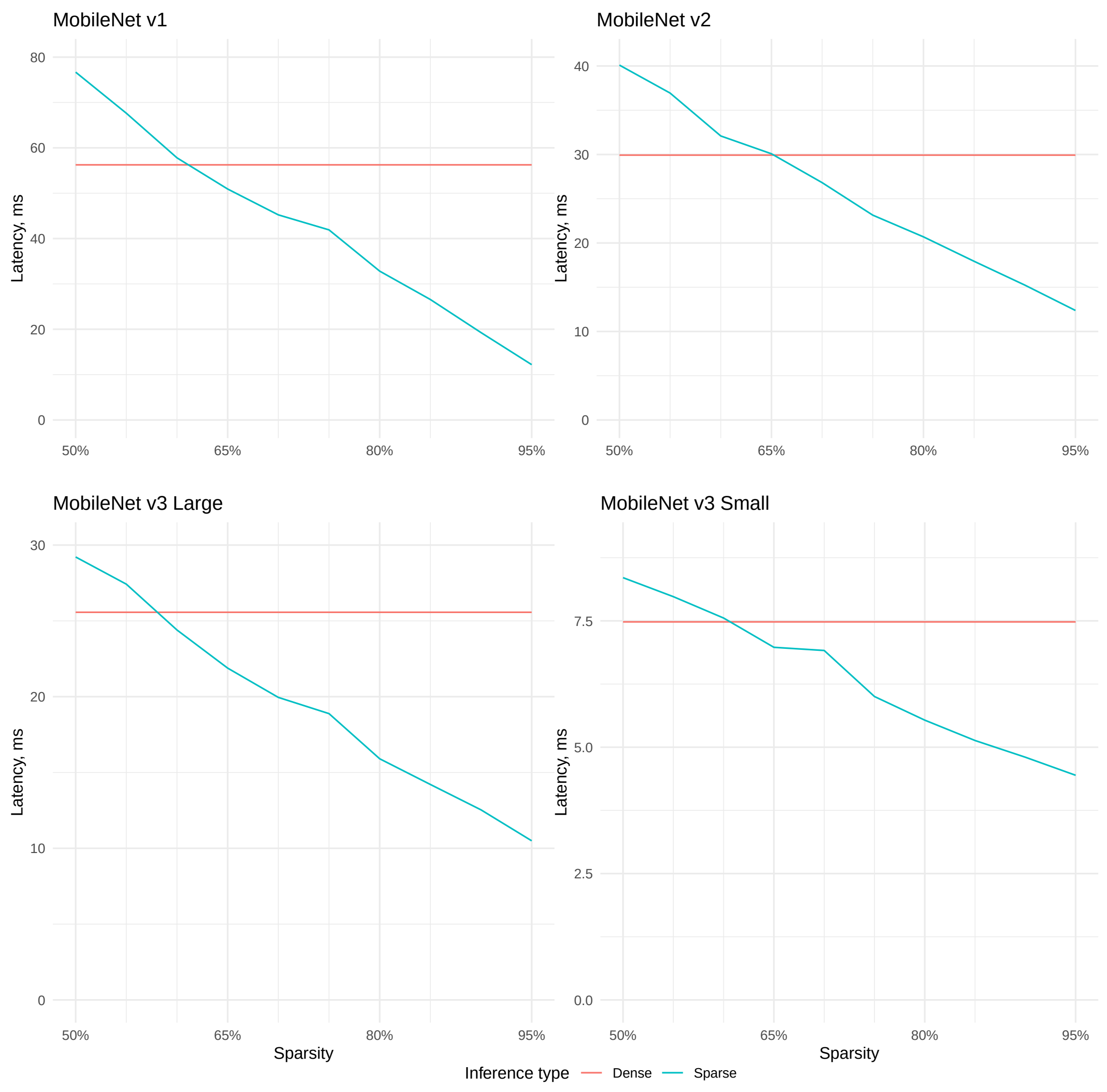


Figure 41: Dense and sparse inference latency in the three generations of MobileNet architectures.

Fig. 41 presents the latency of sparse and dense inference at different levels of weights sparsity. At unstructured 50% sparsity dense inference is more efficient, but as we look at increasing sparsity levels, the two types of inference break-even between $55-65\%$ sparsity, and sparse inference becomes more efficient, peaking at over 2X speedup in a 95% sparse network. Notably, large networks benefit more from sparse inference compared to small ones, as can be seen by comparing MobileNet v3 Large and MobileNet v3 Small architectures. This effect arises as sparse inference accelerates only pointwise convolutions, and pointwise convolutions occupy a larger share of total runtime in large networks since the complexity of pointwise convolutions grows quadratically with the number of channels while complexity of most other neural network operators, including depthwise convolutions, is linear in the number of channels.

The results in Fig. 41 further confirm the efficiency of sparse neural network inference, but miss one important detail: in practice high degrees of sparsity result in degradation in end-to-end accuracy, and thus highly sparse models need to be slightly larger than dense models to achieve equivalent end-to-end accuracy as discussed below.

5.4.5 End-To-End Models

	Model	Scale	Top-1	MFLOPs	Pixel 2	Pixel 3a	Pixel 4a 5G
MNv1	Dense	1.0	70.9	1120	125	106	58
	Sparse	1.4	**72.0**	**268**	**58**	**64**	**34**
MNv1	Dense	0.75	68.4	636	73	64	33
	Sparse	1.0	68.4	**146**	**31**	**34**	**19**
MNv1	Dense	0.5	63.3	290	36	33	18
	Sparse	0.75	**64.4**	**90**	**21**	**21**	**12**
MNv2	Dense	1.4	**75.0**	1110	150	129	64
	Sparse	2.0	74.5	**406**	**93**	**91**	**48**
	Sparse*	1.8	**74.9**	411	102	108	59
MNv2	Dense	1.0	71.8	580	83	74	35
	Sparse	1.4	72.0	**220**	**54**	**53**	**31**
	Dense	0.75	69.8	375	64	57	24
MNv2	Sparse	1.15	**70.2**	165	40	39	**21**
MNv2	Dense	0.5	65.4	182	33	30	**12**
	Sparse	0.8	65.2	**90**	**26**	**24**	13

Table 14: Comparison of dense and sparse model sizes, flops, and inference latencies. All times are in milliseconds.
*This model uses 80% unstructured sparsity in all pointwise convolutions.

For the ultimate validation of the sparse inference acceleration, I evaluate end-to-end latency of dense and sparse MobileNet v1 [46] and MobileNet v2 [82] models trained to approximately equivalent end-to-end accuracy. As expected, the sparse models require larger scale to achieve equivalent accuracy with smaller dense models. All models were trained with input sizes of 224×224, sparse MNv1 models are 90% sparse in every layer, and sparse MNv2 models are 85% sparse. In sparse MobileNet v1 models, pointwise convolution 12 uses blocking along the output channels dimension with block size of 4. In sparse MobileNet v2 models, pointwise convolutions 11 and onward use blocking along the output channels dimension with block size of 2.

Table 14 summarizes the performance evaluation and shows that sparse inference delivers end-to-end speedup compared to dense models of equivalent accuracy. The speedup can range from 25% to over $2X$, with larger models typically being more favorable to sparse inference. Notably, this improvement in efficiency is completely transparent to the users of a machine learning framework, as the inputs and outputs of the machine learning model have the same layout and meaning as they do in dense inference.

5.5 Conclusion

I presented the key ingredients for efficient sparse inference in convolutional neural networks: sparsification of pointwise convolutions with replacement of dense matrix-matrix multiplication with sparse matrix-dense matrix multiplication as the driver for acceleration, dynamic graph rewriting into NCHW layout for efficient sparse matrix-dense matrix multiplication, and Read-Accumulate microkernels for sparse matrix-dense matrix multiplication with GEMM-like instruction mix and low-level efficiency. Combined together, these ingredients deliver up to 2X improvement in inference latency on common convolutional neural network architectures for computer vision while maintaining the input/output interface standard in dense inference machine learning frameworks.

The sparse inference code presented in this Chapter is available as open source software in the XNNPACK library and its integration into TensorFlow Lite machine learning framework. While the results in this chapter validated efficiency improvements on mobile devices, the implementation in XNNPACK goes beyond mobile and powers commercial products using WebAssembly inference.

CHAPTER VI

PRIMITIVES FOR SOFTMAX

The *softmax* (also called *softargmax*) function is a smooth approximation to the *argmax* function. The softmax function $\sigma(x)$ operates on a vector of real-valued scores x_i and normalizes them into a probability distribution

$$p_i = \sigma(x)_i = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

where $p_i \geq 0$ and $\sum_i p_i = 1$.

The softmax function is commonly used in machine learning to assign a probabilistic interpretation to real-valued scores, such as the outputs of classification models [6]. Classification models output a probability for every possible object class, and the number of classes in modern datasets can reach millions. For example, natural language processing models may predict probability distribution over each possible word in a vocabulary, and recommendation systems may model the probability distribution over users, products, web pages, or their interactions. Table 15 summarizes the number of classes on several public classification datasets.

[0]This chapter is based on the workshop paper [22] with minor modifications. ©2020 IEEE. Reprinted, with permission, from "Marat Dukhan, and Artsiom Ablavatski. The Two-Pass Softmax Algorithm. 2020 IEEE International Parallel and Distributed Processing Symposium Workshops, pages 386–395, May 2020." I am the main contributor to the algorithm development, implementation, analysis, and experimental evaluation.

Table 15: Characteristics of several public machine learning datasets.

Dataset	Class description	Class Count
ImageNet [19]	Image category	22 thousand
One Billion Word [9]	Unique Words	0.8 million
Wikilinks [86]	Wikipedia pages	3 million
DepCC [73]	Web documents	365 million

Hierarchical Softmax [32] (HSM) and its modifications [36] are common techniques to scale classification models to large number of classes. HSM models jointly consider the softmax function and the matrix-matrix multiplication that produced its input, and replace them by a low-rank approximation. Thus, HSM methods improve performance by reducing the matrix-matrix multiplication cost, and do so by approximating the original machine learning model.

Unlike previous research, this work focus on the softmax function in the context of inference using a pre-trained model. This situation commonly arises in machine learning frameworks, such as TensorFlow [1] or PyTorch [76], when the training dataset or metaparameters needed to devise an accurate approximation to the model are not available. In this case, the model must be computed exactly according to the specification and unsafe approximations are not possible.

My contributions in this work are as follows:

- I demonstrate that a well-optimized implementation of the softmax function can be memory-bound even in single-threaded execution. This result emphasizes the importance of eliminating memory operations for further improvements in the performance of the softmax function.

- I introduce a novel algorithm for computing the softmax function. The new algorithm employs an exotic representation for intermediate values, where each value is represented as a pair of floating-point numbers: one representing the "mantissa" and another representing the "exponent." Thanks to the special representation of intermediate results, the new algorithm needs only two passes over an input vector versus three passes for traditional algorithms.

- I present and evaluate high-performance implementations of the new Two-Pass softmax algorithms for the x86-64 processors with AVX2 and AVX512F SIMD extensions. The experimental study demonstrates speedups of up to 28% on an Intel Skylake-X system.

The proposed improvements to the softmax implementation are orthogonal to matrix-matrix multiplication optimizations, and can be combined with sparsification [61, 74], low-rank decomposition [20], low-precision arithmetic [52, 75, 90], or hardware acceleration [37, 55] for the matrix-matrix multiplication that produces the softmax input.

6.1 The Three-Pass Algorithm

Direct calculation of the softmax function according to the formula $\sigma(x)_i = \frac{e^{x_i}}{\sum_k e^{x_k}}$ is fraught with numerical issues. A single-precision e^x function overflows[1] for $x > 89$ and underflows[2] for $x < -104$, and, in turn, causes NaN[3] outputs in naïve implementations. In practice, the parts of the machine learning models that produce input to the softmax function are rarely bounded, and thus an implementation can't assume that the input would fall into such narrow range.

[1]Produce floating-point infinity because result is too large to be represented as a finite single-precision floating-point number

[2]Produce 0 because result is too small to be distinguishable from zero in the single-precision floating-point format

[3]Not a Number: a special floating-point value defined by IEEE 754 standard indicating an invalid result

To overcome numerical instability issues machine learning frameworks adapt a workaround by using the equivalence [30]:

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_k e^{x_k}} = \frac{e^{(x_i - c)}}{\sum_k e^{(x_k - c)}}$$

which holds for any c value. In particular, if $c = \max_i x_i$, then:

- No inputs to e^x function exceed zero
- There is at least one zero input to the e^x function, and thus the denominator of the fraction is non-zero.[4]

Together, these properties result in good numerical behavior of the computation and lead to Algorithm 1.

Algorithm 1 The Three-Pass algorithm with re-computation of exponential function

```
function SoftmaxThreePassRecompute($X$, $Y$)
    $N \leftarrow$ Length($X$)
    $\mu \leftarrow \max_{1 \le i \le N} X_i$                          ▷ Pass 1: read X
    $\sigma \leftarrow \sum_{1 \le i \le N} \text{Exp}(X_i - \mu)$       ▷ Pass 2: read X
    $\lambda \leftarrow 1/\sigma$
    for all $1 \le i \le N$ do
        $Y_i \leftarrow \lambda \cdot \text{Exp}(X_i - \mu)$             ▷ Pass 3: read X, write Y
    end for
end function
```

Both of Pass 2 and Pass 3 in Algorithm 1 compute $e^{(x_i - \mu)}$ with the same x_i and μ values. This observation hints at a potential optimization: if computing e^x is expensive, one could save the computed $e^{(x_i - \mu)}$ values to avoid recomputing them in the Pass 3. Such a modification is presented in Algorithm 2.

[4]Mathematically, the denominator of the fraction is always non-zero regardeless of the choice of c, but in floating-point arithmetics it can turn into numerical zero due to underflow when all inputs to the e^x function are large and negative.

Algorithm 2 The Three-Pass algorithm with re-loading of exponential computations

```
function SoftmaxThreePassReload($X$, $Y$)
    $N \leftarrow \textsc{Length}(X)$
    $\mu \leftarrow \max_{1 \le i \le N} X_i$                ▷ Pass 1: read X
    $\sigma \leftarrow 0$
    for all $1 \le i \le N$ do
        $Y_i \leftarrow \textsc{Exp}(X_i - \mu)$              ▷ Pass 2: read X, write Y
        $\sigma \leftarrow \sigma + Y_i$
    end for
    $\lambda \leftarrow 1/\sigma$
    for all $1 \le i \le N$ do
        $Y_i \leftarrow \lambda \cdot Y_i$                    ▷ Pass 3: read Y, write Y
    end for
end function
```

Algorithm 2 computes $e^{(x_i - \mu)}$ values only once, but this reduction in the number of computations comes at a cost: the second pass of Algorithm 2 does both a read and a write for each element, unlike Algorithm 1 where the second pass does only reads.

6.2 The Two-Pass Algorithm

The Three-Pass Algorithms 1 and 2 avoid numerical issues by normalizing inputs relative to their maximum value, but then require an additional memory pass to find the maximum value. In this section I suggest that it is possible to get the numerical stability without the extra memory pass, and present a practical Two-Pass algorithm for softmax computation.

The immediate reason for the numerical instability of a naïve softmax implementation is the saturation of e^x for inputs outside of the narrow range of $[-104, 89]$.

Therefore, one has to look inside the e^x function for a solution.

The e^x function can be implemented in many ways, but practical implementations [28, 50, 66, 24] in IEEE floating-point arithmetic follow the traditional structure of elementary function implementation [71], and include three steps:

1. **Range reduction**, where the problem of approximating e^x on the infinite input domain $x \in (-\infty, +\infty)$ is reduced to a problem of approximating e^x on a small finite range. For e^x, a natural range reduction derives from the equivalence

$$e^x = e^{\overbrace{x - \log 2 \cdot \lfloor x \cdot \log_2 e \rceil}^{t \in \left[-\frac{\log 2}{2}, \frac{\log 2}{2}\right]}} \cdot 2^{\overbrace{\lfloor x \cdot \log_2 e \rceil}^{n \in \mathbb{Z}}}$$

   which decomposes the approximation of e^x on $x \in (-\infty, +\infty)$ into approximating e^x on $t \in \left[-\frac{\log 2}{2}, \frac{\log 2}{2}\right]$ and multiplying the result by 2^n where n is an integer.

2. **Approximation** of the function on the reduced range, i.e. on $\left[-\frac{\log 2}{2}, \frac{\log 2}{2}\right]$ for e^x. This step is achieved through a polynomial or rational approximation, often in combination with table look-ups.

3. **Reconstruction** of the final e^x value from the approximation on the reduced range. For e^x, the reconstruction step consists of multiplication of e^t by 2^n, and can be achieved at low cost by manipulating the exponent field of a binary floating-point number $m := e^t$. **It is this step where the underflow and overflow situations arise**: $e^t \in \left[\frac{\sqrt{2}}{2}, \sqrt{2}\right]$ and thus always fits into a single-precision floating-point number, but $n = \lfloor x \cdot \log_2 e \rceil$ can exceed the range of the exponent field, causing underflow or overflow.

The key idea that enables the Two-Pass Softmax algorithm is to remove the reconstruction step, and instead keep the result of e^x as a pair of floating-point values (m, n), where $m = e^t$. Mathematically, $e^x = m \cdot 2^n$, but in general this expression can not be evaluated in floating-point arithmetic without overflowing or underflowing the result. The representation of n as a *floating-point* number is important: although n is by design always an integer, it can have a very large magnitude and fall outside of the range of standard integer formats. Therefore, the result e^x must be represented as two, rather than one, floating-point numbers. Using multiple floating-point numbers to represent the real-valued result has a long history in double-double, triple-double, quad-double [43] representations and Error-Free Transformations [35]. However, these representations use multiple floating-point numbers to improve precision of floating-point arithmetic, whereas I suggest using two floating-point numbers to extend its dynamic range.

Algorithm 3 The Two-Pass softmax algorithm. ExtExp denotes an exponential function that produce a pair (m, n) of floating-point values.

```
function SoftmaxTwoPass($X$, $Y$)
    $N \leftarrow \textsc{Length}(X)$
    $m_{sum} \leftarrow 0$
    $n_{sum} \leftarrow -\infty$
    for all $1 \leq i \leq N$ do
        $m_i, n_i \leftarrow \textsc{ExtExp}(X_i)$                              $\triangleright$ Pass 1: read X
        $n_{max} \leftarrow \textsc{Max}(n_i, n_{sum})$
        $m_{sum} \leftarrow m_i \cdot 2^{n_i - n_{max}} + m_{sum} \cdot 2^{n_{sum} - n_{max}}$
        $n_{sum} \leftarrow n_{max}$
    end for
    $\lambda_{sum} \leftarrow {}^{1}/_{m_{sum}}$
    for all $1 \leq i \leq N$ do
        $m_i, n_i \leftarrow \textsc{ExtExp}(X_i)$                              $\triangleright$ Pass 2: read X, write Y
        $Y_i \leftarrow m_i \cdot \lambda_{sum} \cdot 2^{n_i - n_{sum}}$
    end for
end function
```

Algorithm 3 presents the softmax computation in just two passes by implementing addition for the (m, n) representation. The reduction pass tracks the running maximum n value among all elements and accumulates the scaled m values to the running sum. It avoids the floating-point overflow by scaling m values by the difference between the corresponding n values and maximum of n values. As this difference is never positive, m values are never scaled up, which ensures the absence of the floating-point overflow.

Algorithm	Memory reads	Memory writes	Bandwidth cost
Three-Pass (Recompute) 1	3N	**1N**	4N
Three-Pass (Reload) 2	3N	2N	5N
Two-Pass 3	**2N**	**1N**	**3N**

Table 16: Theoretical analysis of memory complexity and bandwidth costs of the three softmax algorithms.

6.3 Theoretical Analysis

While the number of memory passes in the presented softmax algorithms is evident from the names, the number of actual memory operations is more nuanced. Every pass of the Three-Pass algorithm with Recomputing reads the input array, while the last pass also writes the output array. The Three-Pass algorithm with Reloading just reads the input array in the first pass, reads the input array and writes the output array in the second pass, and reads-modifies-writes the output array in the last pass. The Two-Pass algorithm reads the input array in both passes, and also writes the output array in the second pass. Thus, the memory bandwidth requirements of the Two-Pass algorithm are similar to just the last two passes of the Three-Pass algorithm with Recomputing.

Table 16 summarizes the number of memory reads, memory writes, and the memory bandwidth cost for the three algorithms on arrays of N elements. Per Table 16, the Two-Pass algorithm has a memory bandwidth advantage of 33% over the Three-Pass algorithm with Recomputing and 67% over the Three-Pass algorithm with Reloading. In practice, I should treat these numbers as upper bounds, because higher computational complexity of the Two-Pass algorithm cuts into gains from bandwidth reduction.

6.4 Experimental Evaluation

6.4.1 Platform

I evaluate the performance of the three softmax algorithms on the Intel Xeon W-2135 processor based on Skylake-X microarchitecture and with the characteristics listed in Table 17. To improve performance stability I disabled dynamic frequency scaling in the processor for the duration of my experiments.

Table 17: Characteristics of the Intel Xeon W-2135 processor used for experimental evaluation of the softmax algorithms.

Characteristic	Value
Microarchitecture	Skylake-X
Number of cores	6
Number of hyperthreads	12
Base frequency	3.7 GHz
L1 cache size (per core)	32 KB
L2 cache size (per core)	1 MB
L3 cache size (shared by all cores)	8.25 MB
AVX2 / AVX512 FMA throughput	2 / cycle
AVX2 / AVX512 FMA latency	4 cycles

Additionally, in Sec. 6.4.8 I replicate a subset of the experiments on a Broadwell-based Intel Xeon E5-2696 v4 processor and on an AMD Ryzen 9 3900X processor with Zen 2 microarchitecture.

6.4.2 Protocol

I use the Google Benchmark framework to estimate sustained performance of the softmax implementations. I set the minimum run-time for each measurement to 5 seconds, repeat each measurement 25 times and record the median of the 25 runs. In each benchmark run I simulate the cache state during neural network inference: output vector is evicted from the cache before each iteration, but the input tensor stays in cache as long as it fits.

6.4.3 Implementation

I developed highly optimized implementations of the Three-Pass Algorithms 1 and 2, and the Two-Pass Algorithm 3 in C. For all three algorithms, I did two templated implementations targeting AVX2 and AVX512 instruction sets. I expressed high-level optimization parameters, such as unroll factor for the loops and the number of accumulator variables in reduction functions, as meta-parameters of the templated implementations, and employed auto-tuning to discover their optimal values.

An efficient implementation of vectorized e^x function is a key component of all softmax variants. For my implementation, I adapted the throughput-optimized methods of Dukhan and Vuduc [24] to single-precision floating-point evaluation. Algorithm 4 presents the resulting table-free, branch-free, and division-free algorithm.

Algorithm 4 Calculation of e^x in the Three-pass softmax algorithms

```
function EXP($x$)
    $n \leftarrow \lfloor x \cdot \log_2 e \rceil$
    $t \leftarrow x - n \cdot \log 2$                                        ▷ Cody-Waite range reduction
    $p \leftarrow 1 + t(c_1 + t(c_2 + t(c_3 + t(c_4 + t \cdot c_5)))))$      ▷ Polynomial approximation
    $y \leftarrow p \cdot 2^n$                                               ▷ Reconstruction
    return $y$
end function
```

Algorithm 4 follows the traditional structure of elementary function implementation [71], described in Sec. 6.2. It starts with a range reduction to reduce approximation on the infinite input domain to approximation on a small and finite range.

The calculation of the reduced argument t in Algorithm 4 uses Cody-Waite range reduction [15]: $\log 2$ is represented as a sum of two single-precision constants, $\log_{hi} 2$ and $\log_{lo} 2$, to improve the accuracy of this step. Range reduction results in a reduced argument t in the $[-\frac{\log 2}{2}, \frac{\log 2}{2}]$ range and a reduced integer argument n. Next, e^t is approximated on $[-\frac{\log 2}{2}, \frac{\log 2}{2}]$ with a degree-5 polynomial. The polynomial coefficients are produced by the algorithm of Brisebarre and Chevillard [7] as implemented in the Sollya software package [13]. Following [24], I evaluate the approximation polynomial with a Horner scheme using Fused Multiply-Add instructions to minimize the number of floating-point instructions and maximize the throughput. In the last stage Algorithm 4 reconstructs the final output value of the function by multiplying the polynomial approximation result p by 2^n. In the AVX2 implementation, I do this multiplication by directly manipulating floating-point exponent to construct a scale number s:

$$s := \begin{cases} 2^n & n >= -1260 \\ 0 & n < 126 \leq x \end{cases}$$

and compute the final step as $y \leftarrow p \cdot s$. This reconstruction trick has two built-in assumptions: the argument x to e^x is negative,[5] and subnormal floating-point numbers can be flushed to zero without significant accuracy impact. The reconstruction step in the AVX512 implementation leverages the new VSCALEFPS instruction [17], which computes $p \cdot 2^n$ as a single hardware operation.

The resulting e^x implementation has a maximum error under 2 ULP, validated through exhaustive evaluation on all valid inputs. This accuracy is comparable to other vectorized elementary function implementations, e.g., SIMD functions in the GNU LibM library guarantee maximum error under 4 ULP.

Implementation of the ExtExp in the Two-Pass softmax algorithm is similar to Algorithm 4 with the reconstruction step removed. Thus, implementations of both

[5] Always the case for the e^x evaluation in the Three-Pass softmax algorithms.

the Three-Pass and the Two-Pass algorithms use exactly the same range reduction and approximating polynomials to compute the exponential function.

6.4.4 The Three-Pass Algorithms and Bandwidth Saturation

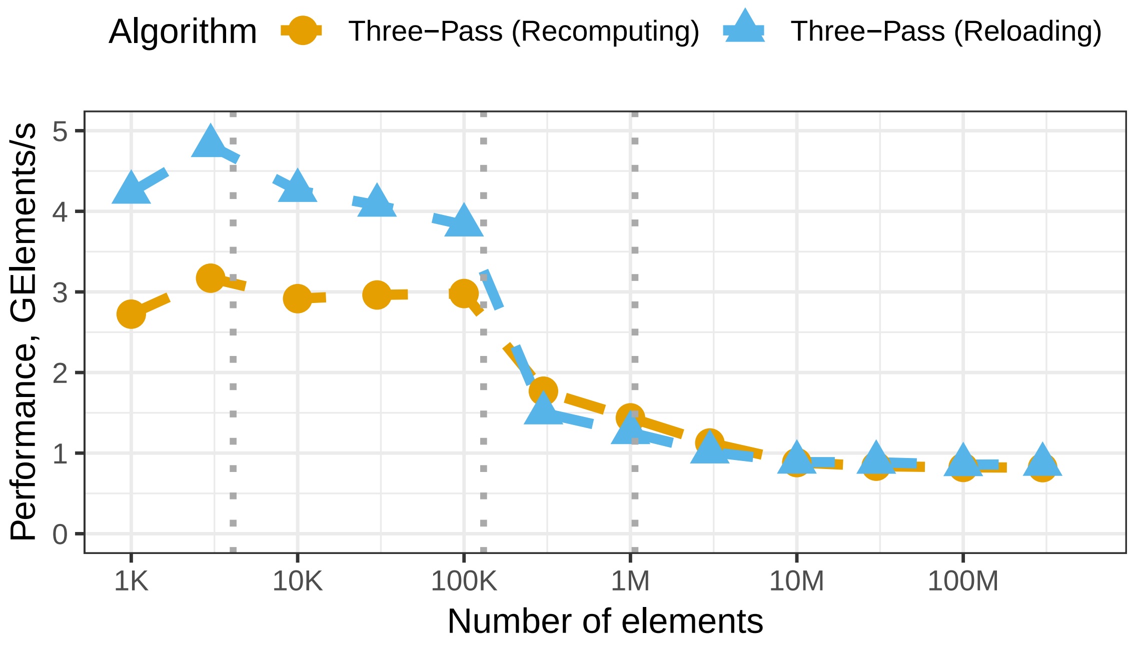


Figure 42: Single-threaded performance comparison of the Softmax algorithms 1 and 2 in the AVX512 implementations on the Skylake-X system. Gray dotted lines denote boundaries of level-1, level-2, and level-3 caches.

Fig. 42 presents the single-threaded performance of the Three-Pass softmax Algorithm 1 with recomputing of exponentials and the Three-Pass softmax Algorithm 2 with reloading of computed exponentials in the AVX512 implementations. Reloading of exponential computations delivers $30-55\%$ speedup when the data is small enough to fit into private L1 and L2 caches, but turns into 15% slowdown when operating on L3 cache, and eventually levels off at $4-6\%$ advantage when working set exceeds last-level cache.

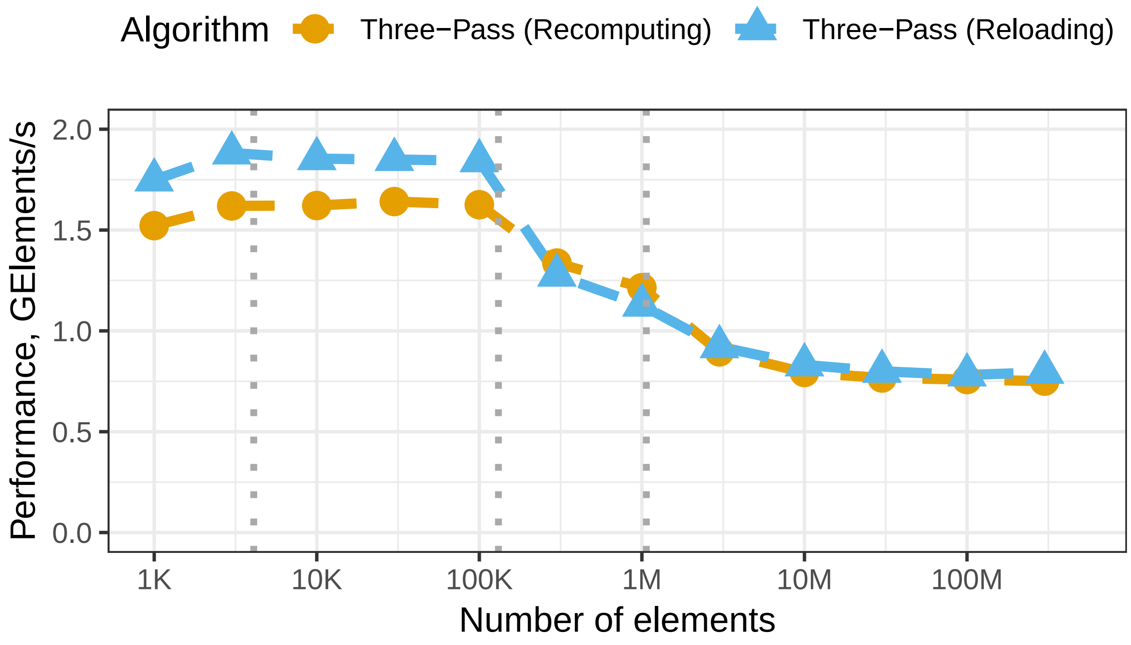


Figure 43: Single-threaded performance comparison of the Softmax algorithms 1 and 2 in the AVX2 implementations on the Skylake-X system. Gray dotted lines denote boundaries of level-1, level-2, and level-3 caches.

The AVX2 implementation of the same Three-Pass softmax Algorithm 1 and Algorithm 2 is illustrated in Fig. 43 and demonstrates similar trends. As the working set increases, the $13-16\%$ speedup from reloading of exponential computations goes down, and eventually levels off at $3-6\%$ for large arrays.

The small difference between recomputing and reloading of exponential computations on Fig. 42 and Fig. 43 suggests that despite the expensive exponential function, softmax might be memory-bound for large arrays. To directly test this hypothesis, I decompose Algorithms 1 and 2 into individual memory passes and compare measured bandwidth to STREAM benchmarks [68].

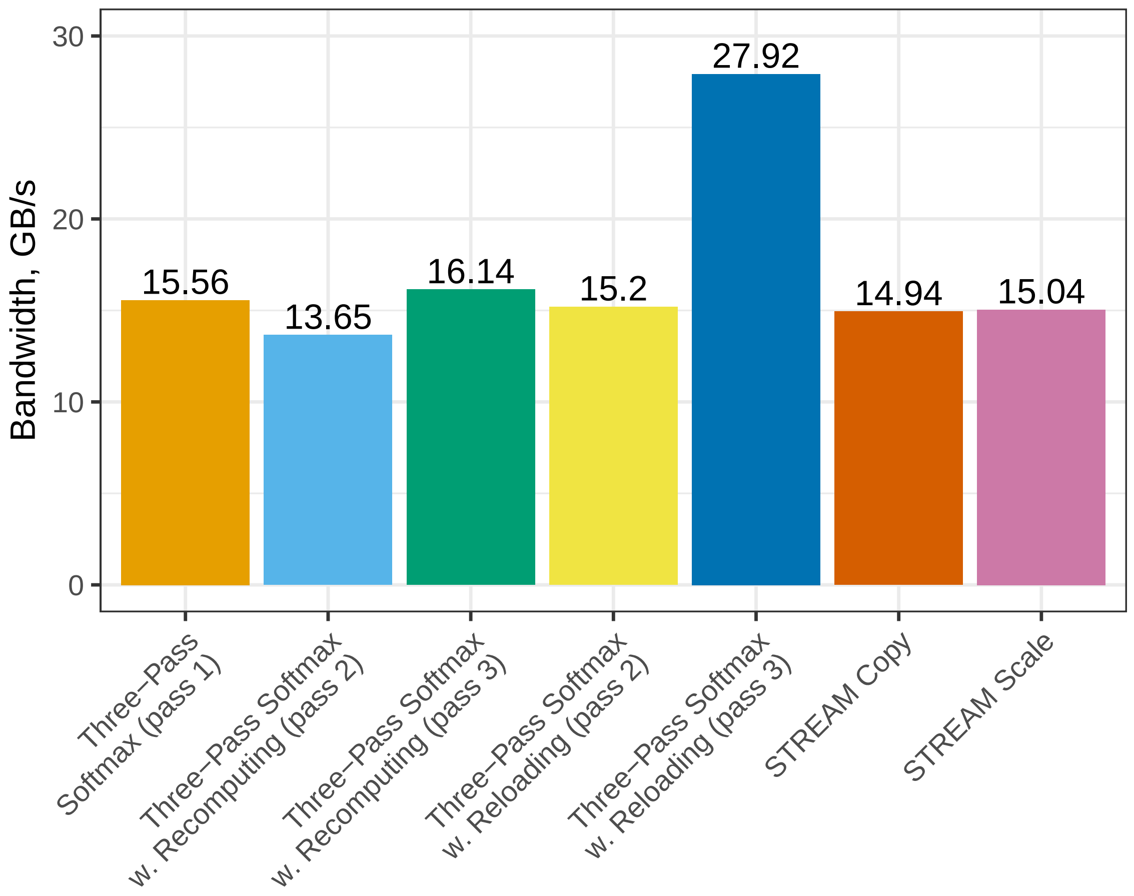


Figure 44: Measured single-threaded memory bandwidth on the Skylake-X system in the three passes of the Softmax algorithms 1 and 2, and in the STREAM benchmark. Both the softmax implementations and the STREAM benchmark use AVX512 instructions.

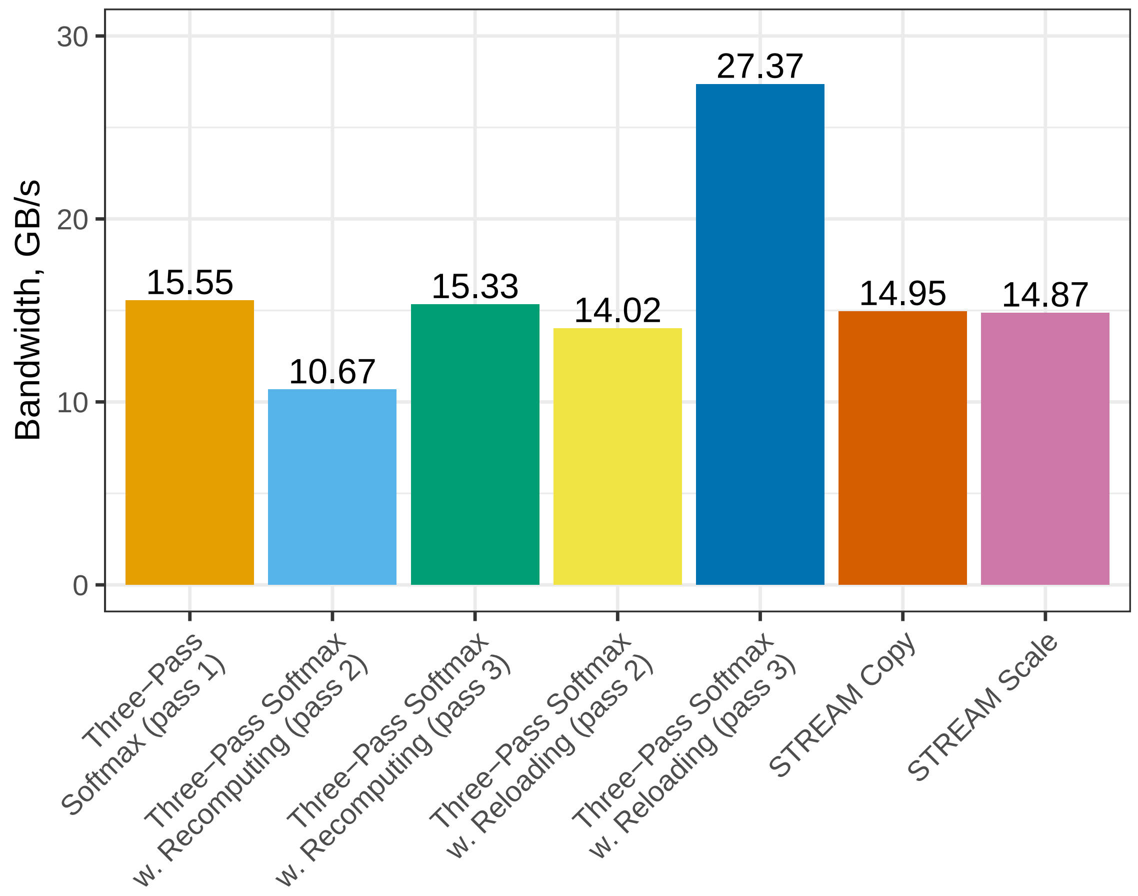


Figure 45: Measured single-threaded memory bandwidth on the Skylake-X system in the three passes of the Softmax algorithms 1 and 2, and in the STREAM benchmark. Both the softmax implementations and the STREAM benchmark use AVX2 instructions.

Fig. 44 and 45 illustrate the memory bandwidth in each pass of the Three-Pass softmax Algorithms 1 and 2, as well as Copy and Scale STREAM benchmarks [68]. As recommended in STREAM's documentation, I set the array size to four times the size of last-level cache ($8,650,752$ single-precision elements for Softmax, and $4,325,376$ double-precision elements for STREAM). The first softmax pass (max-reduction) is the same in both versions of the Three-Pass algorithm and thus presented only once. This pass reads one input array and doesn't have a direct equivalent in STREAM.

However, it achieves similar bandwidth to STREAM Copy and Scale benchmarks, which both read one array and write one array. The second pass in Algorithm 1 reads one array, computes exponentials on the inputs, and accumulates them. It achieves 91% of STREAM Copy bandwidth in AVX512 version and 71% in AVX2 version. The second pass Algorithm 2 is similar, but additionally stores computed exponents into the output array. Although it achieves higher bandwidth than the second pass in Algorithm 1, it takes substantially longer to complete; the higher bandwidth is due to doubling the number of transferred bytes with a less than proportional increase in run time. The third pass of Algorithm 1 reads one array, computes exponentials on the inputs, scales them, and writes results to another array. This pass does the same number of memory operations as STREAM Scale benchmark, but substantially more computational operations. Yet, my auto-tuned implementations exceed the performance of STREAM Scale benchmark in both the AVX512 and the AVX2 versions. The third pass of the Algorithm 2 is an in-place variant of STREAM Scale benchmark. The processor clearly favors in-place operation: it is 86% faster than STREAM Scale with AVX512, and 84% faster with AVX2.

To summarize, passes 1 and 3 of the Algorithm with Recomputing 1 demonstrate similar memory performance to STREAM benchmark, and pass 2 in AVX512 implementation is not far behind. Passes 1 and 2 of the Algorithm 2 with Reloading similarly perform close to STREAM bandwidth, and pass 3 is significantly faster than STREAM Scale benchmark. These results confirm that performance of Three-Pass softmax algorithms is limited by achievable memory bandwidth and suggest that **softmax computation can be further accelerated only through reducing the number of memory operations**.

6.4.5 The Two-Pass Algorithm

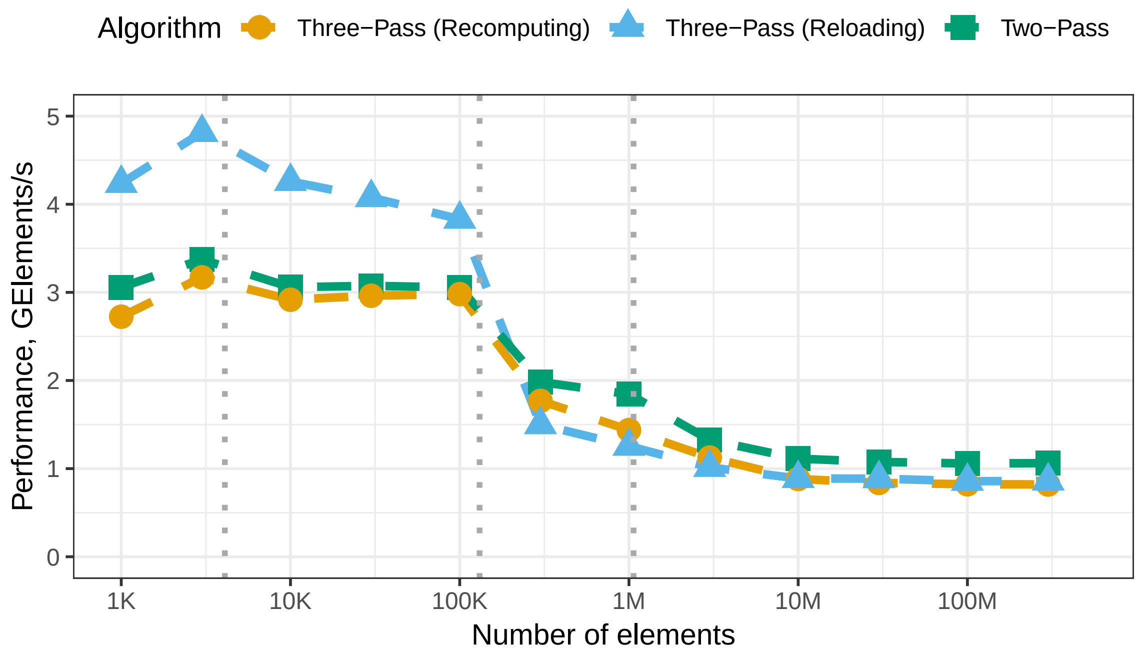


Figure 46: Single-threaded performance comparison of the Algorithms 1, 2, and 3 in the AVX512 implementations. Gray dotted lines denote boundaries of level-1, level-2, and level-3 caches.

On Fig. 46 I present the performance of the Two-Pass softmax algorithm in comparison with the two versions of the Three-Pass algorithm in AVX512 implementations. On out-of-cache working sets the proposed Two-Pass softmax algorithm outperforms Three-Pass algorithms by $18\% - 28\%$.

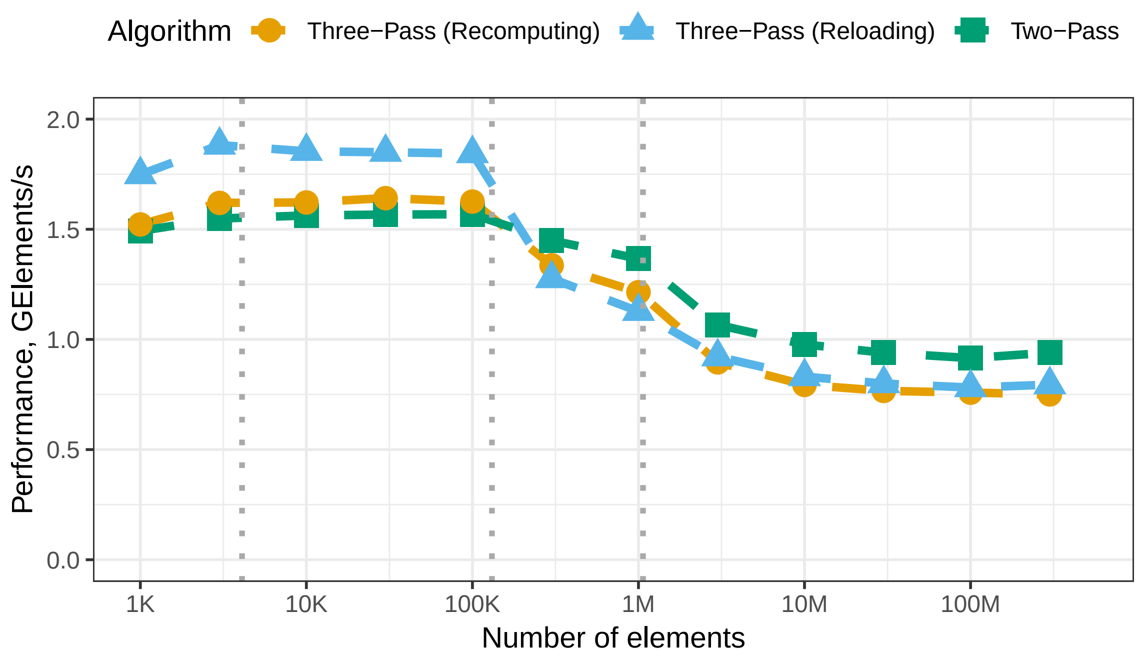


Figure 47: Single-threaded performance comparison of the Algorithms 1, 2, and 3 in the AVX2 implementations. Gray dotted lines denote boundaries of level-1, level-2, and level-3 caches.

Fig. 47 similarly compares performance of the Two-Pass and the Three-Pass algorithms in the AVX2 implementations. Here, the Two-Pass algorithm outperforms Three-Pass algorithm with Reloading of exponential computations by $16\% - 18\%$ on out-of-cache workloads. The smaller speedups, compared to AVX512 implementation, are explained by relatively higher cost of recomputing exponentials in AVX2 compared to AVX512. Compared to the Three-Pass Algorithm 1, which similarly recomputed exponentials, the Two-Pass algorithm wins by $19 - 25\%$.

In Fig.48 I decompose the absolute run-time for the three algorithms and two SIMD instruction sets into individual memory passes and offers insight into the origin of performance improvements with the Two-Pass algorithm. The two passes of the Two-Pass softmax algorithm have similar, but slightly higher absolute run-time to

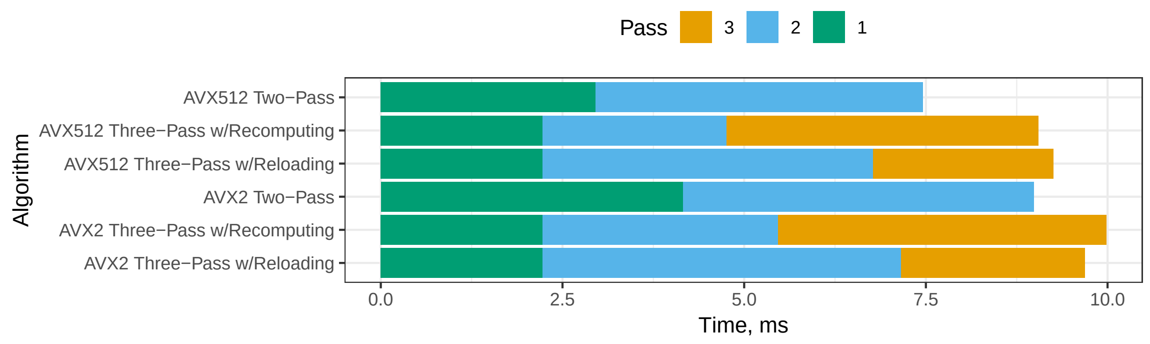


Figure 48: Absolute runtime of the passes in the Algorithms 1, 2, and 3 in both the AVX2 and the AVX512 implementations. The algorithms were evaluated on arrays of 8,650,752 single-precision elements on a single threaded of the Skylake-X system.

the last two passes of the Three-Pass softmax algorithm with recomputation of re-computation of exponential function, which share the same memory access pattern. The slightly higher run-time in the passes of the Two-Pass algorithm can be explained by larger number of operations needed for accumulation on the (m, n) representation compared to just accumulating scalar floating-point values.

6.4.6 Multi-Threaded Performance

The benchmarks in Sec. 6.4.4 and Sec. 6.4.5 presented performance of a single-threaded softmax computation and demonstrated that on HPC-class systems softmax saturates memory bandwidth even when running on a single core. Utilizing multiple cores increases available computational resources faster than achievable memory bandwidth, and therefore increases the advantage of the bandwidth-saving Two-Pass softmax algorithm. To quantify this advantage, I fix the size of the array at 4 times the last-level cache size [68] and vary the number of threads from 1 to 6 (number of cores in the system) to 12 (number of logical processors, including hyperthreads, in the system).

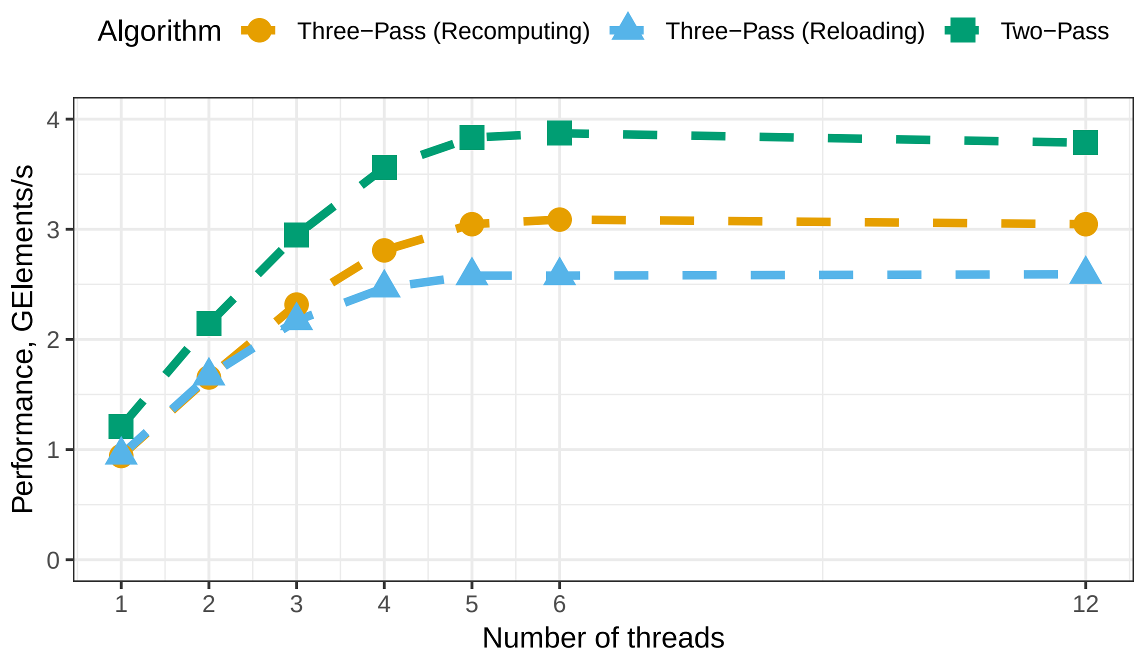


Figure 49: Weak scaling (the number of elements scales proportionally to the number of threads used) of the softmax algorithms in the AVX512 implementations on the Skylake-X system.

Fig. 49 illustrates weak multi-core scaling of the AVX512 implementations. As the number of threads grows, the advantage of the Two-Pass over Three-Pass algorithms remains unchanged at $25 - 28\%$. Interestingly, the reloading variant of the Three-Pass algorithm scales worse than the recomputing variant, and the recomputing Algorithm 1 outperforms the reloading Algorithm 2 when at least 3 cores are being utilized.

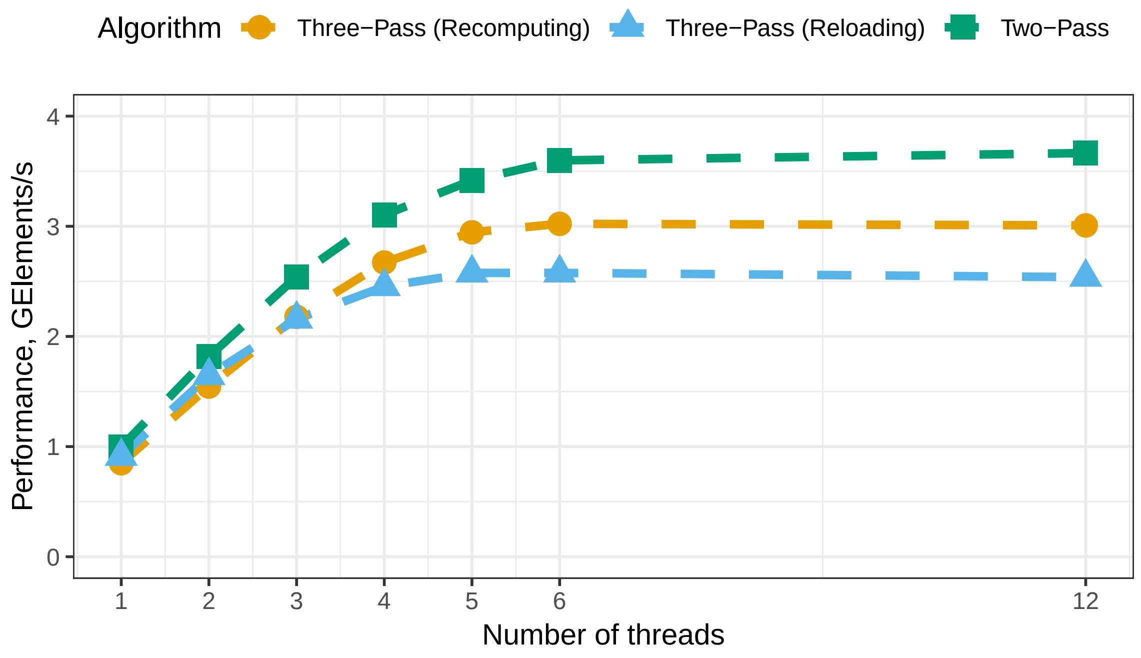


Figure 50: Weak scaling (the number of elements scales proportionally to the number of threads used) of the softmax algorithms in the AVX2 implementations on the Skylake-X system.

Fig. 50 similarly illustrates weak multi-core scaling of the AVX2 implementations. The advantage of the Two-Pass over Three-Pass algorithms grows from 9% on a single core to 19% when utilizing all 6 cores to 22% when also using hyperthreads.

6.4.7 Comparison with Intel DNNL

The results in Sec. 6.4.5-6.4.6 demonstrate that on out-of-cache inputs the Two-Pass softmax algorithm outperforms the Three-Pass softmax algorithms in my implementation, but leaves out the question of whether my implementations are competitive with the state-of-the-art. To demonstrate the absolute effectiveness of the Two-Pass algorithm, I compared my implementations of the three softmax algorithm to the softmax primitive in Intel Deep Neural Network Library (DNNL) version 1.1.1.

Intel DNNL implements the Three-Pass softmax Algorithm 2 with reloading of

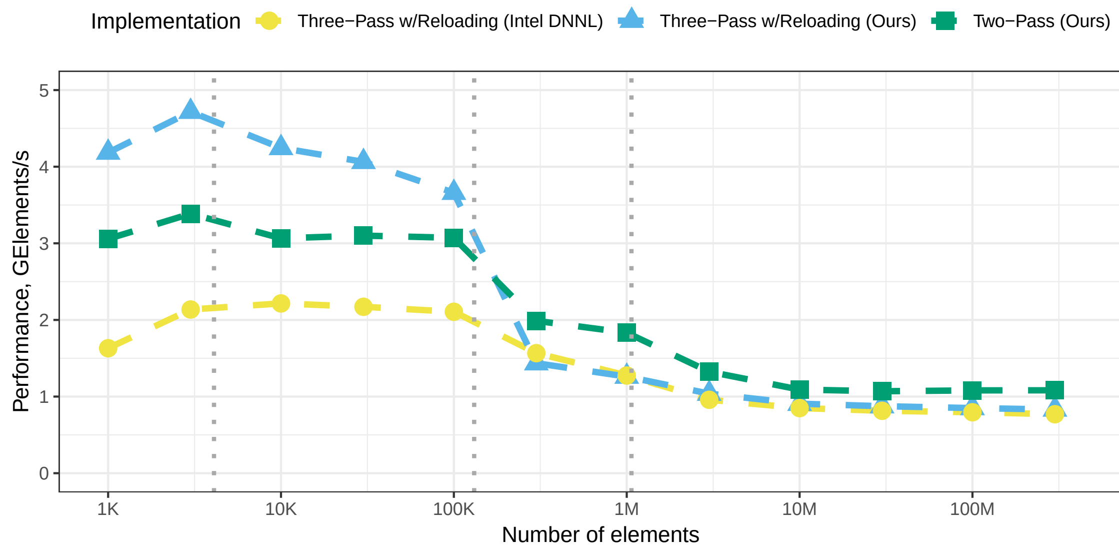


Figure 51: Performance comparison of my implementation of Algorithms 1, 2, and 3, with the softmax implementation in Intel DNNL library. Gray dotted lines denote boundaries of level-1, level-2, and level-3 caches.

computed exponentials. It includes implementations for SSE4.1, AVX, and AVX512 instruction sets, and automatically dispatches to an AVX512 implementation on the Skylake-X processor. Unlike my implementations, Intel DNNL generates its implementation at runtime using Just-in-Time (JIT) technology. JIT code generation potentially allows adaptation of the implementation to parameters of a particular softmax operator (e.g., the number of channels).

Fig. 51 presents the comparison between two implementations (Ours and DNNL) of the Three-Pass algorithm with reloading of exponentials, and the Two-Pass softmax algorithm in my implementation. For the Three-Pass algorithm with reloading, my implementation ourperforms Intel DNNL for the majority of problem sizes. Its advantage is particularly high – over 2X – when data fits into L1, diminish to $72 - 94\%$ within L2, and levels off at $7 - 8\%$ for out-of-cache problem sizes. As the implementation in Intel DNNL is less efficient than ours, my Two-Pass softmax algorithm outperforms DNNL softmax primitive on all problem sizes: it is $28 - 41\%$ faster on out-of-cache problem sizes, and up to 87% when input fits into L1 cache.

6.4.8 Validation on Alternative Processors

The results in Sec. 6.4.4-6.4.7 were all collected on the Xeon W-2135 processor with the Intel Skylake-X microarchitecture, which prompts a question as to whether the advantage of the Two-Pass softmax algorithm is restricted to a specific type of processor. To demonstrate that the Two-Pass algorithm generalizes to other types of processors, I replicated results of Sec. 6.4.5 on Xeon E5-2696 v4 processor with Intel Broadwell microarchitecture and Ryzen 9 3900X with AMD Zen 2 microarchitecture. Both of these processors support AVX2, but not AVX512, and have different cache hierarchy parameters than the Intel Skylake-X system.

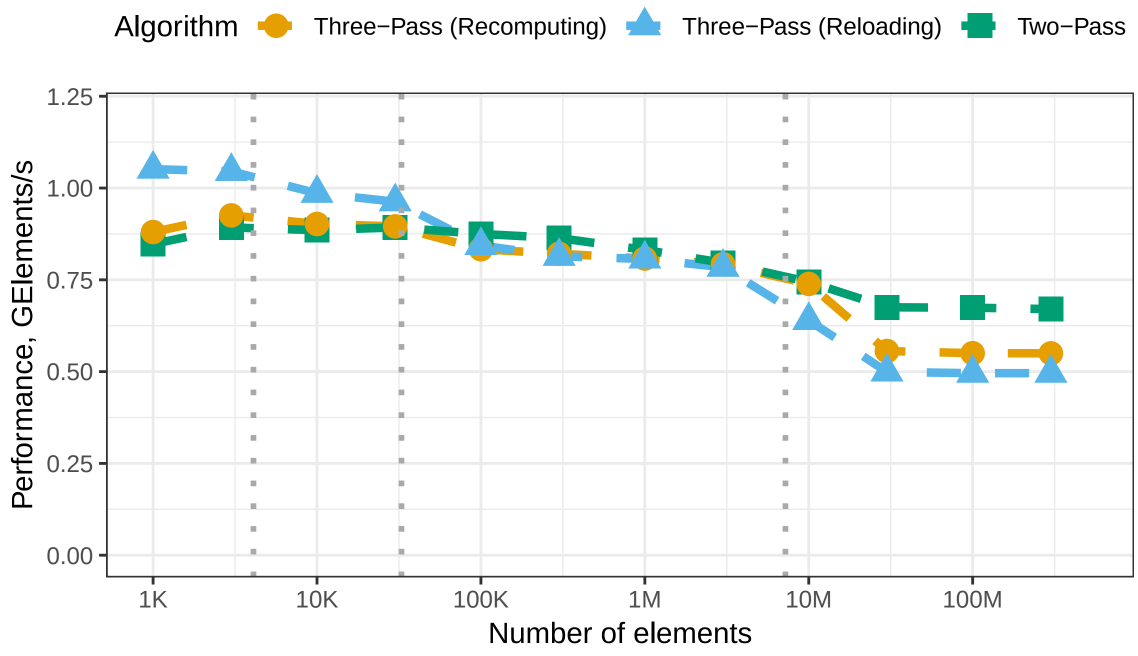


Figure 52: Performance comparison of Algorithms 1, 2, and 3 on an Intel Broadwell-based system. Gray dotted lines denote boundaries of level-1, level-2, and level-3 caches.

Fig. 52 presents performance of the three softmax algorithms on the Intel Broadwell system. Although the Two-Pass softmax algorithm demonstrates inferior performance on problems which fit into L2 cache, it is competitive with the Three-Pass softmax algorithms on L3-sizes problems, and outperforms them by $21 - 23\%$ on out-of-cache problems.

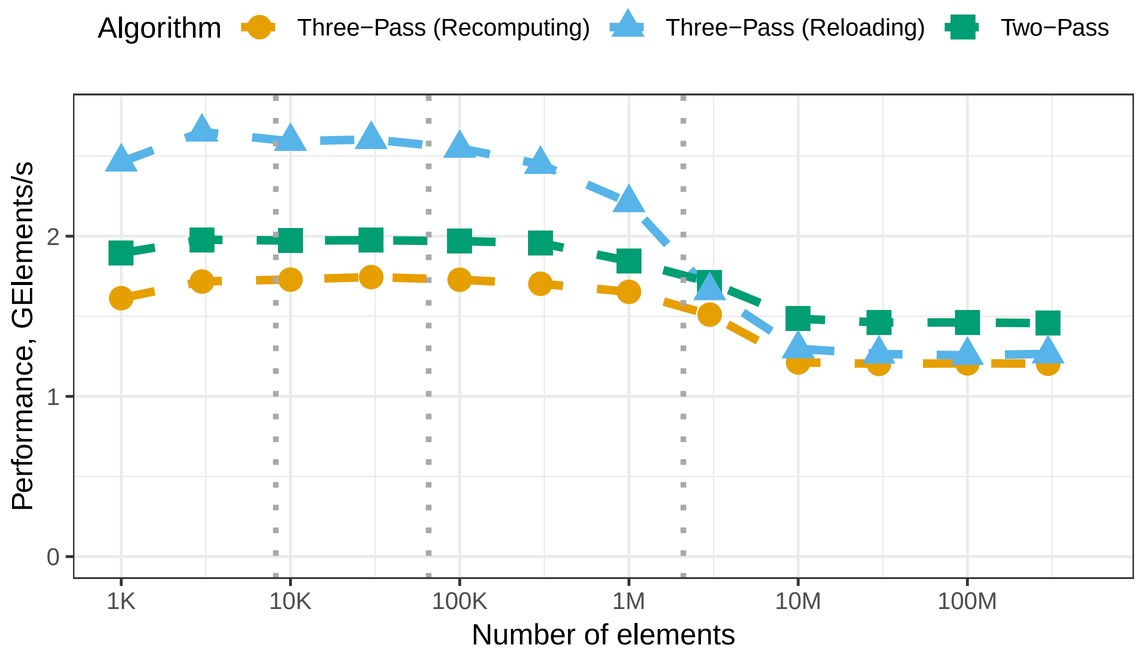


Figure 53: Performance comparison of Algorithms 1, 2, and 3 on a Ryzen 9 3900X system. Gray dotted lines denote boundaries of level-1, level-2, and level-3 caches.

Fig. 53 shows a similar picture on AMD Zen 2 microarchitecture. Here, the Three-Pass algorithms deliver superior performance as long as data fits into L3 cache, but lose $14 - 16\%$ to the Two-Pass algorithm when the data exceeds cache size.

6.5 Conclusion

I presented a novel Two-Pass algorithm for softmax computation and demonstrated that my new Two-Pass algorithm is up to 28% faster than the traditional Three-Pass algorithm on large input vectors. The algorithm, however, offers no advantage over a reloading variant of the Three-Pass algorithm when the data fits into the processor's cache.

This study focused on performance on a CPU, but the algorithm has great potential for GPUs and hardware AI accelerators. These platforms further shift the balance between compute and memory performance towards expensive memory and cheap floating-point operations, and would favor the reduced memory intensity of the presented Two-Pass softmax algorithm.

CHAPTER VII

CONCLUSION

In less than a decade neural networks transitioned from a curious novelty to a standard instrument of scientific research and engineering practice. Common performance library interfaces, such as BLAS, FFTW, and Intel MKL, predate the explosion in neural network adaption and incur inefficiencies when directly used for neural network inference. These inefficiencies primarily arise from insufficient flexibility of classical performance primitives, and require expensive memory layout transformations to map neural network inference computations to the performance primitives provided by existing libraries.

In this dissertation I demonstrated how the primitives from performance libraries dense matrix-matrix multiplication, sparse matrix-dense matrix multiplication, 2D Fourier transform, and vector mathematical functions can be modified to exceed state-of-the-art performance for neural network inference. Importantly, the modified primitives differ in their interfaces, yet exhibit similar structure to the classical well-studied variants from performance libraries, and this change has two implications. First, most research on improving matrix-matrix multiplication, Fourier transforms, and vector mathematical functions can be with small modifications adapted for the optimized neural network inference primitives, enabling AI optimizations to tap into a vast trove of prior art. Secondly, current hardware, extensively optimized over decades for classical scientific computing primitives, is by extension well suited for neural network inference provided that the software uses the most efficient algorithms.

The thesis leaves several questions related to the modified performance primitives unexplored. First, as the proposed primitives improve the efficiency of convolutions and softmax, elementwise operations such as addition, elementwise multiplication, and activation functions become responsible for a sizeable fraction of inference runtime. Elementwise operations are intrinsically memory-bound and become ideal candidates for fusion into the proposed performance primitives, e.g., Inverse Fourier transform (for FFT-based fast convolution), Output Winograd Transform (for Winograd-based fast convolution), Indirect GEMM (for low-intensity dense convolution), or SpMM (for sparse pixelwise convolution). Naturally, is it infeasible to provide every possible fused primitive as a part of a performance library, but it remains to be studied if a constrained set of fused primitives is sufficient to accelerate most widely used neural networks. Compilers may have a role in carrying out such fusion transformations. Secondly, the question of combining the proposed primitives with quantization is largely open. Indirect GEMM is clearly useful for quantized inference (and this was explored by the author in the QNNPACK library), but it remains to be seen if the primitives for fast convolution algorithms or sparse inference can be quantized without dramatic degradation of end-to-end accuracy. Both fast convolution algorithms and especially sparse inference involve some accuracy loss, and so does quantization; whether this accuracy loss is orthogonal to the accuracy loss from quantization is an open question. Adaptation of the Two-Pass Softmax algorithm to quantized inference is yet another open question: the original algorithm heavily relied on properties of floating-point numbers. Thirdly, two of the proposed primitives, Tuple-GEMM for fast convolution algorithms and Indirect GEMM for the indirect convolution algorithm, are both neccesiated by the constrained nature of the original GEMM primitive, but in different ways: for Tuple-GEMM the critical constraint was lack of efficient batching support in the GEMM interface, and for Indirect GEMM it was the only reduction dimension being insufficient to represent multi-dimensional reduction in the convolution operator. It is unclear if both generalizations could be combined in an efficient and practically useful manner.

The ideas on efficient neural network inference outlined in this dissertation were implemented in the NNPACK, QNNPACK, and XNNPACK libraries, which are widely disseminated in the machine learning community. In particular, among the three largest ecosystems of machine learning frameworks, the PyTorch ecosystem includes NNPACK, QNNPACK, and XNNPACK libraries, the TensorFlow ecosystem makes use of XNNPACK library, and the TVM ecosystem integrates NNPACK library. With NNPACK, QNNPACK, and XNNPACK libraries being integrated into the Android Open Source Project, Instagram, and Snapchat, the impact of these libraries extends beyond machine learning engineers and researchers, and enables AI-powered experiences for over two billion people.

REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X., "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (USA), p. 265–283, USENIX Association, 2016.

[2] Abdel-Hamid, O., Mohamed, A.-r., Jiang, H., Deng, L., Penn, G., and Yu, D., "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[3] Al-Rfou, R., Alain, G., Almahairi, A., Angermüller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Snyder, J. B., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P. L., Cho, K., Chorowski, J., Christiano, P. F., Cooijmans, T., Côté, M., Côté, M., Courville, A. C., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Ducoffe, M., Dumoulin, V., Kahou, S. E., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I. J., Graham, M., Gülçehre, Ç., Hamel, P., Harlouchet, I., Heng, J., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Korobov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrançois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P., Mastropietro, O., McGibbon, R., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C. J., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, É., Spieckermann, S., Subramanyam, S. R., Sygnowski, J., Tanguay, J., van Tulder, G., Turian, J. P., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., and Zhang, Y., "Theano: A python framework for fast computation of mathematical expressions," *CoRR*, vol. abs/1605.02688, 2016.

[4] Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen,

G., Chen, J., Chen, J., Chen, Z., Chrzanowski, M., Coates, A., Diamos, G., Ding, K., Du, N., Elsen, E., Engel, J., Fang, W., Fan, L., Fougner, C., Gao, L., Gong, C., Hannun, A., Han, T., Johannes, L. V., Jiang, B., Ju, C., Jun, B., LeGresley, P., Lin, L., Liu, J., Liu, Y., Li, W., Li, X., Ma, D., Narang, S., Ng, A., Ozair, S., Peng, Y., Prenger, R., Qian, S., Quan, Z., Raiman, J., Rao, V., Satheesh, S., Seetapun, D., Sengupta, S., Srinet, K., Sriram, A., Tang, H., Tang, L., Wang, C., Wang, J., Wang, K., Wang, Y., Wang, Z., Wang, Z., Wu, S., Wei, L., Xiao, B., Xie, W., Xie, Y., Yogatama, D., Yuan, B., Zhan, J., and Zhu, Z., "Deep Speech 2: End-to-end speech recognition in English and Mandarin," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, p. 173–182, JMLR.org, 2016.

[5] Anderson, A., Vasudevan, A., Keane, C., and Gregg, D., "High-performance low-memory lowering: GEMM-based algorithms for DNN convolution," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 99–106, 2020.

[6] Bridle, J. S., "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition," in *Neurocomputing* (Soulié, F. F. and Hérault, J., eds.), (Berlin, Heidelberg), pp. 227–236, Springer Berlin Heidelberg, 1990.

[7] Brisebarre, N. and Chevillard, S., "Efficient polynomial L-approximations," in *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*, pp. 169–176, 2007.

[8] Brosch, T. and Tam, R., "Efficient training of convolutional deep belief networks in the frequency domain for application to high-resolution 2D and 3D images," *Neural Comput.*, vol. 27, p. 211–227, Jan. 2015.

[9] Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T., "One billion word benchmark for measuring progress in statistical language modeling," tech. rep., Google, 2013.

[10] Chellapilla, K., Puri, S., and Simard, P., "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition* (Lorette, G., ed.), (La Baule (France)), Université de Rennes 1, Suvisoft, Oct. 2006. http://www.suvisoft.com.

[11] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z., "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[12] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A., "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, (USA), p. 579–594, USENIX Association, 2018.

[13] Chevillard, S., Joldeş, M., and Lauter, C., "Sollya: An environment for the development of numerical codes," in *Mathematical Software – ICMS 2010* (Fukuda, K., Hoeven, J. v. d., Joswig, M., and Takayama, N., eds.), (Berlin, Heidelberg), pp. 28–31, Springer Berlin Heidelberg, 2010.

[14] Chollet, F., "Xception: Deep learning with depthwise separable convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1800–1807, 2017.

[15] Cody, W. J., *Software Manual for the Elementary Functions (Prentice-Hall Series in Computational Mathematics)*. USA: Prentice-Hall, Inc., 1980.

[16] Collobert, R., Kavukcuoglu, K., and Farabet, C., "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[17] Cornea, M., "Intel AVX-512 instructions and their use in the implementation of math functions," *Intel Corporation*, 2015.

[18] Davis, T. A. and Hu, Y., "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[19] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L., "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[20] Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R., "Exploiting linear structure within convolutional networks for efficient evaluation," in *Advances in Neural Information Processing Systems* (Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K. Q., eds.), vol. 27, Curran Associates, Inc., 2014.

[21] Duchi, J., Hazan, E., and Singer, Y., "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, p. 2121–2159, July 2011.

[22] Dukhan, M. and Ablavatski, A., "Two-Pass Softmax algorithm," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (Los Alamitos, CA, USA), pp. 386–395, IEEE Computer Society, may 2020.

[23] Dukhan, M., "The Indirect Convolution algorithm," *CoRR*, vol. abs/1907.02129, 2019.

[24] Dukhan, M. and Vuduc, R., "Methods for high-throughput computation of elementary functions," in *Parallel Processing and Applied Mathematics* (Wyrzykowski, R., Dongarra, J., Karczewski, K., and Waśniewski, J., eds.), (Berlin, Heidelberg), pp. 86–95, Springer Berlin Heidelberg, 2014.

[25] Elsen, E., Dukhan, M., Gale, T., and Simonyan, K., "Fast sparse convnets," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, (Los Alamitos, CA, USA), pp. 14617–14626, IEEE Computer Society, jun 2020.

[26] Fog, A. and others, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA cpus," *Copenhagen University College of Engineering*, vol. 93, p. 110, 2011.

[27] Frigo, M. and Johnson, S., "FFTW: an adaptive software architecture for the FFT," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, vol. 3, pp. 1381–1384 vol.3, 1998.

[28] Gal, S., "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Trans. Math. Softw.*, vol. 17, p. 26–45, Mar. 1991.

[29] Gatys, L. A., Ecker, A. S., and Bethge, M., "A neural algorithm of artistic style," *CoRR*, vol. abs/1508.06576, 2015.

[30] Goodfellow, I., Bengio, Y., and Courville, A., *Deep Learning.* MIT Press, 2016.

[31] Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y., "Maxout networks," in *Proceedings of the 30th International Conference on Machine Learning* (Dasgupta, S. and McAllester, D., eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 1319–1327, PMLR, 17–19 Jun 2013.

[32] Goodman, J., "Classes for fast maximum entropy training," in *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, vol. 1, (Los Alamitos, CA, USA), pp. 561–564, IEEE Computer Society, may 2001.

[33] Goto, K. and van de Geijn, R. A., "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, May 2008.

[34] Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K., "Accurate, large minibatch SGD: training ImageNet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017.

[35] Graillat, S. and Ménissier-Morain, V., "Error-free transformations in real and complex floating point arithmetic," in *International Symposium on Nonlinear Theory and its Applications (NOLTA'07)*, (Vancouver, Canada), pp. 341–344, Sept. 2007.

[36] Grave, É., Joulin, A., Cissé, M., Grangier, D., and Jégou, H., "Efficient softmax approximation for GPUs," in *Proceedings of the 34th International Conference on Machine Learning* (Precup, D. and Teh, Y. W., eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 1302–1310, PMLR, 06–11 Aug 2017.

[37] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J., "EIE: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254, 2016.

[38] Han, S., Mao, H., and Dally, W. J., "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR, abs/1510.00149*, vol. 2, 2015.

[39] Han, S., Pool, J., Narang, S., Mao, H., Tang, S., Elsen, E., Catanzaro, B., Tran, J., and Dally, W. J., "DSD: regularizing deep neural networks with dense-sparse-dense training flow," *CoRR*, vol. abs/1607.04381, 2016.

[40] Han, S., Pool, J., Tran, J., and Dally, W., "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems* (Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., eds.), vol. 28, Curran Associates, Inc., 2015.

[41] He, K., Zhang, X., Ren, S., and Sun, J., "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[42] Heinecke, A., Henry, G., Hutchinson, M., and Pabst, H., "LIBXSMM: Accelerating small matrix multiplications by runtime code generation," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 981–991, 2016.

[43] Hida, Y., Li, X., and Bailey, D., "Algorithms for quad-double precision floating point arithmetic," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, pp. 155–162, 2001.

[44] Hinton, G., Srivastava, N., and Swersky, K., "Lecture 6a overview of mini–batch gradient descent,"

[45] Howard, A., Sandler, M., Chen, B., Wang, W., Chen, L.-C., Tan, M., Chu, G., Vasudevan, V., Zhu, Y., Pang, R., Adam, H., and Le, Q.,

"Searching for MobileNetV3," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 1314–1324, 2019.

[46] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H., "MobileNets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[47] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q., "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, 2017.

[48] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y., "Binarized neural networks," in *Advances in Neural Information Processing Systems* (Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., and Garnett, R., eds.), vol. 29, Curran Associates, Inc., 2016.

[49] Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K., "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[50] Iordache, C. and Tang, P., "An overview of floating-point support and math library on the Intel XScale architecture," in *Proceedings 2003 16th IEEE Symposium on Computer Arithmetic*, pp. 122–128, 2003.

[51] Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A., "Image-to-image translation with conditional adversarial networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5967–5976, 2017.

[52] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[53] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T., "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, (New York, NY, USA), p. 675–678, Association for Computing Machinery, 2014.

[54] Johnson, J., Alahi, A., and Fei-Fei, L., "Perceptual losses for real-time style transfer and super-resolution," in *European Conference on Computer Vision*, pp. 694–711, Springer, 2016.

[55] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau,

M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H., "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, p. 1–12, June 2017.

[56] Kingma, D. and Ba, J., "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[57] Krizhevsky, A., Sutskever, I., and Hinton, G. E., "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, p. 84–90, May 2017.

[58] Lavin, A., "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015.

[59] Leary, C. and Wang, T., "XLA: TensorFlow, compiled," *TensorFlow Dev Summit*, 2017.

[60] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P., "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[61] Liu, B., Wang, M., Foroosh, H., Tappen, M., and Penksy, M., "Sparse convolutional neural networks," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 806–814, 2015.

[62] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C., "SSD: Single shot multibox detector," in *Computer Vision – ECCV 2016* (Leibe, B., Matas, J., Sebe, N., and Welling, M., eds.), (Cham), pp. 21–37, Springer International Publishing, 2016.

[63] Loshchilov, I. and Hutter, F., "Fixing weight decay regularization in Adam," *CoRR*, vol. abs/1711.05101, 2017.

[64] Ma, N., Zhang, X., Zheng, H.-T., and Sun, J., "ShuffleNet v2: Practical guidelines for efficient CNN architecture design," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 116–131, September 2018.

[65] Mahajan, D., Girshick, R. B., Ramanathan, V., He, K., Paluri, M., Li, Y., Bharambe, A., and van der Maaten, L., “Exploring the limits of weakly supervised pretraining,” *CoRR*, vol. abs/1805.00932, 2018.

[66] Markstein, P., *IA-64 and elementary functions - speed and precision.* 2000.

[67] Mathieu, M., Henaff, M., and LeCun, Y., “Fast training of convolutional networks through FFTs,” *CoRR*, vol. abs/1312.5851, 2013.

[68] McCalpin, J. D., “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[69] Mehta, S., Rastegari, M., Caspi, A., Shapiro, L., and Hajishirzi, H., “ESPNet: Efficient spatial pyramid of dilated convolutions for semantic segmentation,” in *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[70] Mehta, S., Rastegari, M., Shapiro, L., and Hajishirzi, H., “ESPNetv2: A light-weight, power efficient, and general purpose convolutional neural network,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9182–9192, 2019.

[71] Muller, J.-M., Brisebarre, N., De Dinechin, F., Jeannerod, C.-P., Lefevre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S., and others, “Handbook of floating-point arithmetic,” 2010.

[72] Odena, A., Dumoulin, V., and Olah, C., “Deconvolution and checkerboard artifacts,” *Distill*, 2016.

[73] Panchenko, A., Ruppert, E., Faralli, S., Ponzetto, S. P., and Biemann, C., “Building a Web-scale dependency-parsed corpus from CommonCrawl,” in *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)* (chair), N. C. C., Choukri, K., Cieri, C., Declerck, T., Goggi, S., Hasida, K., Isahara, H., Maegaard, B., Mariani, J., Mazo, H., Moreno, A., Odijk, J., Piperidis, S., and Tokunaga, T., eds.), (Miyazaki, Japan), European Language Resources Association (ELRA), May 7-12, 2018 2018.

[74] Park, J., Li, S. R., Wen, W., Li, H., Chen, Y., and Dubey, P., “Holistic SparseCNN: Forging the trident of accuracy, speed, and size,” *arXiv preprint arXiv:1608.01409*, vol. 1, no. 2, 2016.

[75] Park, J., Naumov, M., Basu, P., Deng, S., Kalaiah, A., Khudia, D. S., Law, J., Malani, P., Malevich, A., Satish, N., Pino, J., Schatz, M., Sidorov, A., Sivakumar, V., Tulloch, A., Wang, X., Wu, Y., Yuen, H., Diril, U., Dzhulgakov, D., Hazelwood, K. M., Jia, B., Jia, Y., Qiao, L., Rao, V., Rotem, N., Yoo, S., and Smelyanskiy, M., “Deep

learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications," *CoRR*, vol. abs/1811.09886, 2018.

[76] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S., "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[77] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A., "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Computer Vision – ECCV 2016* (Leibe, B., Matas, J., Sebe, N., and Welling, M., eds.), (Cham), pp. 525–542, Springer International Publishing, 2016.

[78] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V., "Regularized evolution for image classifier architecture search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4780–4789, Jul. 2019.

[79] Redmon, J. and Farhadi, A., "Yolo9000: Better, faster, stronger," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6517–6525, 2017.

[80] Ren, S., He, K., Girshick, R., and Sun, J., "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems* (Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., eds.), vol. 28, Curran Associates, Inc., 2015.

[81] Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Satish, N., Olesen, J., Park, J., Rakhov, A., and Smelyanskiy, M., "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, 2018.

[82] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C., "MobileNetV2: Inverted residuals and linear bottlenecks," pp. 4510–4520, 06 2018.

[83] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y., "Overfeat: Integrated recognition, localization and detection using convolutional networks," *arXiv preprint arXiv:1312.6229*, 2013.

[84] Shi, W., Caballero, J., Huszár, F., Totz, J., Aitken, A., Bishop, R., Rueckert, D., and Wang, Z., "Real-time single image and video superresolution using an efficient sub-pixel convolutional neural network," 06 2016.

[85] Simonyan, K. and Zisserman, A., "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[86] Singh, S., Subramanya, A., Pereira, F., and McCallum, A., "Wikilinks: A large-scale cross-document coreference corpus labeled via links to Wikipedia," *University of Massachusetts, Amherst, Tech. Rep. UM-CS-2012*, vol. 15, 2012.

[87] Smith, T. M., Van De Geijn, R., Smelyanskiy, M., Hammond, J. R., and Van Zee, F. G., "Anatomy of high-performance many-threaded matrix multiplication," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1049–1059, IEEE, 2014.

[88] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A., "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[89] Tan, M. and Le, Q. V., "EfficientNet: Rethinking model scaling for convolutional neural networks," *CoRR*, vol. abs/1905.11946, 2019.

[90] Tulloch, A. and Jia, Y., "High performance ultra-low-precision convolutions on mobile devices," *arXiv preprint arXiv:1712.02427*, 2017.

[91] van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K., "WaveNet: A generative model for raw audio," in *9th ISCA Speech Synthesis Workshop*, pp. 125–125, 2016.

[92] Van Zee, F. G. and van de Geijn, R. A., "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, pp. 14:1–14:33, June 2015.

[93] Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y., "Fast convolutional nets with fbfft: A GPU performance evaluation," *arXiv preprint arXiv:1412.7580*, 2014.

[94] Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H., "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems* (Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., and Garnett, R., eds.), vol. 29, Curran Associates, Inc., 2016.

[95] Whaley, R. C. and Dongarra, J. J., "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp. 1–27, IEEE Computer Society, 1998.

[96] Williams, S. W., *Auto-tuning performance on multicore computers.* University of California, Berkeley Berkeley, CA, 2008.

[97] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Łukasz Kaiser, Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J., "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[98] Yu, D., Eversole, A., Seltzer, M., Yao, K., Kuchaiev, O., Zhang, Y., Seide, F., Huang, Z., Guenter, B., Wang, H., Droppo, J., Zweig, G., Rossbach, C., Gao, J., Stolcke, A., Currey, J., Slaney, M., Chen, G., Agarwal, A., Basoglu, C., Padmilac, M., Kamenev, A., Ivanov, V., Cypher, S., Parthasarathi, H., Mitra, B., Peng, B., and Huang, X., "An introduction to computational networks and the computational network toolkit," Tech. Rep. MSR-TR-2014-112, October 2014.

[99] Zeiler, M. D., "ADADELTA: an adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012.

[100] Zhang, J., Franchetti, F., and Low, T. M., "High performance zero-memory overhead direct convolutions," in *Proceedings of the 35th International Conference on Machine Learning* (Dy, J. and Krause, A., eds.), vol. 80 of *Proceedings of Machine Learning Research*, pp. 5776–5785, PMLR, 10–15 Jul 2018.

[101] Zhang, X., Zhou, X., Lin, M., and Sun, J., "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6848–6856, 2018.

[102] Zhu, C., Han, S., Mao, H., and Dally, W. J., "Trained ternary quantization," *CoRR*, vol. abs/1612.01064, 2016.

[103] Zlateski, A., Lee, K., and Seung, H. S., "ZNNi: Maximizing the inference throughput of 3D convolutional networks on CPUs and GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, IEEE Press, 2016.