Optimizing Convolutional Layers

User Guide | NVIDIA Docs

DU-09795-001_v001 | June 2021

Table of Contents

List of Figures

List of Tables

Chapter 1. Quick-Start Checklist

The following quick-start checklist provides specific tips for convolutional layers.

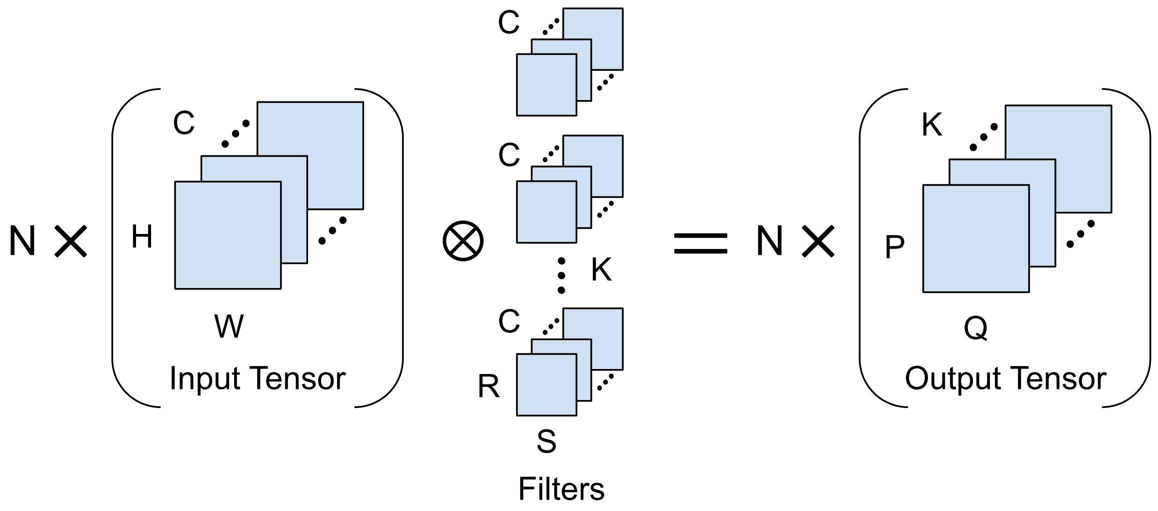
- Choose the number of input and output channels to be divisible by 8 (for FP16) or 4 (for TF32) to run efficiently on Tensor Cores. For the first convolutional layer in most CNNs where the input tensor consists of 3-channel images, padding to 4 channels is sufficient if a stride of 2 is used; see Channels In And Out.
- Choose parameters (batch size, number of input and output channels) to be divisible by at least 64 and ideally 256 to enable efficient tiling and reduce overhead; see Quantization Effects.
- Larger values for size-related parameters (batch size, input and output height and width, and the number of input and output channels) can improve parallelization. As with fully-connected layers, this speeds up an operation's efficiency, but does not reduce its absolute duration; see How Convolution Parameters Affect Performance and subsections.
- NVIDIA libraries offer a set of different convolution algorithms with different performance behaviors, dependent on the convolution's parameters. When the size of the input processed by the network is the same in each iteration, autotuning is an efficient method to ensure the selection of the ideal algorithm for each convolution in the network. For TensorFlow, autotuning is enabled by default. For PyTorch, enable autotuning by adding `torch.backends.cudnn.benchmark = True` to your code.
- Choose tensor layouts in memory to avoid transposing input and output data. There are two major conventions, each named for the order of dimensions: NHWC and NCHW. We recommend using the NHWC format where possible. Additional details, including framework support, can be found in Tensor Layouts In Memory: NCHW vs NHWC.

Chapter 2. Introduction

A convolution is defined by the sizes of the input and filter tensors and the behavior of the convolution, such as the padding type used.

Figure 1 illustrates the minimum parameter set required to define a convolution.

Figure 1. Convolution of an NCHW input tensor with a KCRS weight tensor, producing a NKPQ output.



In the remainder of this guide, we'll use the single-letter abbreviations for each parameter.

Note: N and K here are not related to the N and K parameters in a General Matrix Multiplication (GEMM); GEMM dimensions will be represented here as `M x N x K` to keep them distinct from the convolution's parameters. GEMMs and GEMM performance are explained in the Matrix Multiplication Background User Guide.

Table 1. Parameters defining a convolution

Parameter	Tensor	Meaning
`N`	N/A	Batch size

Parameter	Tensor	Meaning
`C`	Input	Number of channels
`H`		Height
`W`		Width
`K`	Output	Number of channels
`P`		Height (often derived from other parameters)
`Q`		Width (often derived from other parameters)
`R`	Filter	Height
`S`		Width
`U`		Vertical stride
`V`		Horizontal stride
`PadH`		Input padding in the vertical dimension
`PadW`		Input padding in the horizontal dimension
`DilH`		Dilation in the vertical dimension
`DilW`		Dilation in the horizontal dimension

Chapter 3. Convolution Algorithms

NVIDIA cuDNN library implements convolutions using two primary methods: implicit-GEMM-based and transform-based.

The implicit GEMM approach is a variant of direct convolution, and operates directly on the input weight and activation tensors. Alternatively, convolutions can be computed by transforming data and weights into another space, performing simpler operations (for example, pointwise multiplies), and then transforming back. The cuDNN library provides some convolution implementations using FFT and Winograd transforms.

3.1. Choosing A Convolution Algorithm With cuDNN

When running a convolution with cuDNN, for example with cudnnConvolutionForward(), you may specify which general algorithm is used.

The cuDNN API provides functions for estimating the relative performance of different algorithms. One set of functions, prefixed with `cudnnGet`, uses a set of heuristics to predict the relative performance of available algorithms. These functions evaluate quickly; however, although we're constantly improving our heuristics, the predictions may not always be accurate. Suboptimal algorithm choice may occasionally occur, and is more common for unusual types of convolutions and corner cases.

An alternative set of functions, prefixed with `cudnnFind`, tests and reports the performance of all available algorithms to determine the most efficient option for the given convolution operation. The benefit of using these functions is that the algorithm selected is the best choice. However, since actual performance tests are being run, these functions can be time and resource-intensive.

After an algorithm is chosen, our heuristics specify additional low-level details; for example, tile size, discussed at length in Quantization Effects and in the Dimension Quantization Effects section in the *Matrix Multiplication Background User Guide*.

As of cuDNN version 8, the cuDNN backend API grants more precise control over tile size and other parameters of the algorithm used.

Chapter 4. Tensor Core Usage And Performance Recommendations

The primary method to execute convolutions (without transforms) used by NVIDIA Tensor Core GPUs is called implicit GEMM. It performs exactly the same number of math operations as a direct convolution and hence is computationally equivalent.

Implicit GEMM operates natively on the convolution input tensors, converting the computation into a matrix multiply on the fly. It is important to note that corresponding matrices are never created in memory. Thus, to calculate arithmetic intensity, one can use the original tensor sizes.

To illustrate the concept of convolution as a matrix multiply let's first consider a single application of a convolution filter to input data. Say we are applying a 3x3 convolution to a 128-channel input tensor. To compute a single output value, we effectively compute a dot-product of two 1,152-element vectors. One is the weights for a filter (`3x3x128 = 1,152`). The other is composed of the data (activation) values that are multiplied with the weights to produce the output. Since not all 1,152 data values are contiguous in memory, the original tensor layout is read and on the fly is converted to the appropriate vector form. To compute all the outputs we perform multiple dot-products, which can be seen as a matrix multiply, but since the matrices are implicitly formed, rather than created in memory, this method is called *implicit* GEMM. To understand the performance of convolutions, however, it can be useful to understand the shapes and sizes of these "virtual" matrices.

4.1. Tensor Layouts In Memory: NCHW vs NHWC

Convolutions typically operate on four-dimensional tensors: a batch composed of N "images" of C channels of H x W feature maps.

Deep learning frameworks commonly use NCHW and NHWC layouts in memory (the acronym lists the dimensions from the slowest to the fastest varying in memory). Layout choice has an effect on performance, as convolutions implemented for Tensor Cores require NHWC layout and are fastest when input tensors are laid out in NHWC.

Note: NCHW layouts can still be operated on by Tensor Cores, but include some overhead due to automatic transpose operations, as shown in Figure 2. Transpose overhead tends to be more significant when input and output tensors are larger or the amount of calculation required is lower (such as when filter sizes are smaller). To maximize performance, we recommend using the NHWC tensor layout.

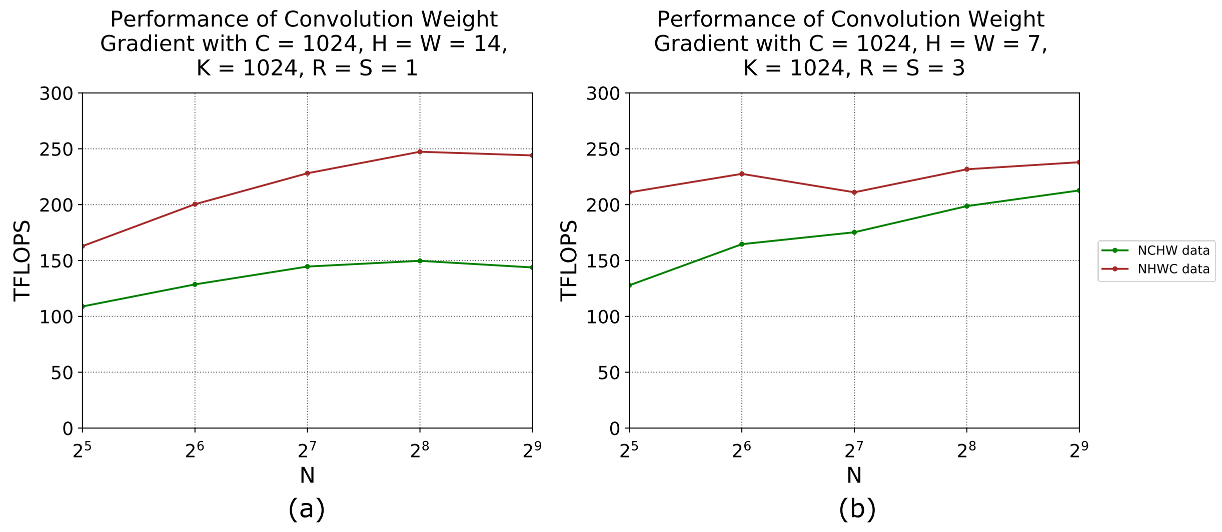


Figure 2. Kernels that do not require a transpose (NHWC) perform better than kernels that require one or more (NCHW). NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.

At the time of writing, NHWC layouts are available in frameworks including MxNet (via the `layout` argument, documentation page here), TensorFlow (via the `data_format` argument, documentation page here), and PyTorch (via the Channels Last memory format, documentation page here). Performance examples in this section can be assumed to use input and output data in the NHWC layout unless otherwise stated.

4.2. Implicit GEMM Dimensions

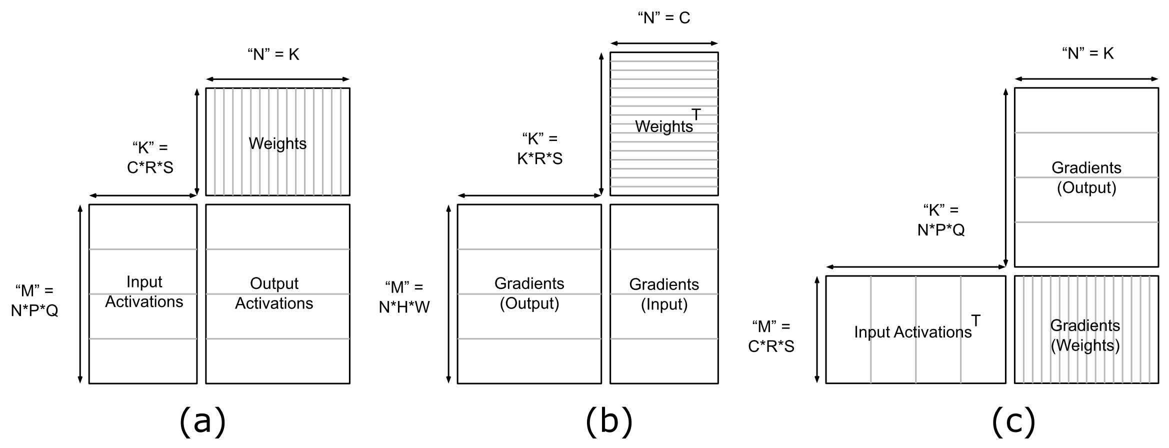
Let's now consider the dimensions of the matrices we encounter when performing forward convolution, calculating activation gradients, and calculating weight gradients.

Table 2. Translation of convolution parameters to corresponding GEMM parameters

Computation Phase	"M"	"N"	"K"
Forward Propagation	N*P*Q	K	C*R*S
Activation Gradient	N*H*W	C	K*R*S
Weight Gradient	C*R*S	K	N*P*Q

The composition of the "virtual" matrices is shown below. For each pass, there is one virtual matrix that, if explicitly constructed, would contain more values than its corresponding tensor. For example, during forward convolution, the A matrix (`N*P*Q x C*R*S`) is composed of input activations (a tensor with dimensions `N x H x W x C`). Each individual input activation appears in `R*S` places in the matrix, repeated with necessary offsets to cause multiplication of that input value with the overlaid values of the matching `R x S` filter channel. Similar conceptual expansions occur for output activation gradients when computing input activation gradients, and for input activations during weight gradient calculation.

Figure 3. Dimensions of equivalent GEMMs for (a) forward convolution, (b) activation gradient calculation, and (c) weight gradient calculation



It is important to reiterate that matrices of these sizes are not stored in memory; they are an abstraction to help explain the computation. The "repeated" values are not literally copied, and wasteful reads from memory are avoided. This is directly visible in the calculation of arithmetic intensity. A (forward) implicit GEMM reads two input tensors of size NCHW and

KCRS and produces one output tensor of size NKPQ using CRS multiplies and adds per element of the NKPQ output tensor, for a total of NKPQCRS multiply-accumulates. Therefore, the arithmetic intensity in FP16 with 2 bytes/element is:

$$\text{Arithmetic Intensity} = \frac{\#\ \text{ops}}{\#\ \text{bytes}} = \frac{2 \cdot (N \cdot K \cdot P \cdot Q) \cdot (C \cdot R \cdot S)}{2 \cdot (N \cdot C \cdot H \cdot W + K \cdot C \cdot R \cdot S + N \cdot K \cdot P \cdot Q)} = \frac{N \cdot K \cdot P \cdot Q \cdot C \cdot R \cdot}{N \cdot C \cdot H \cdot W + K \cdot C \cdot R \cdot S +}$$

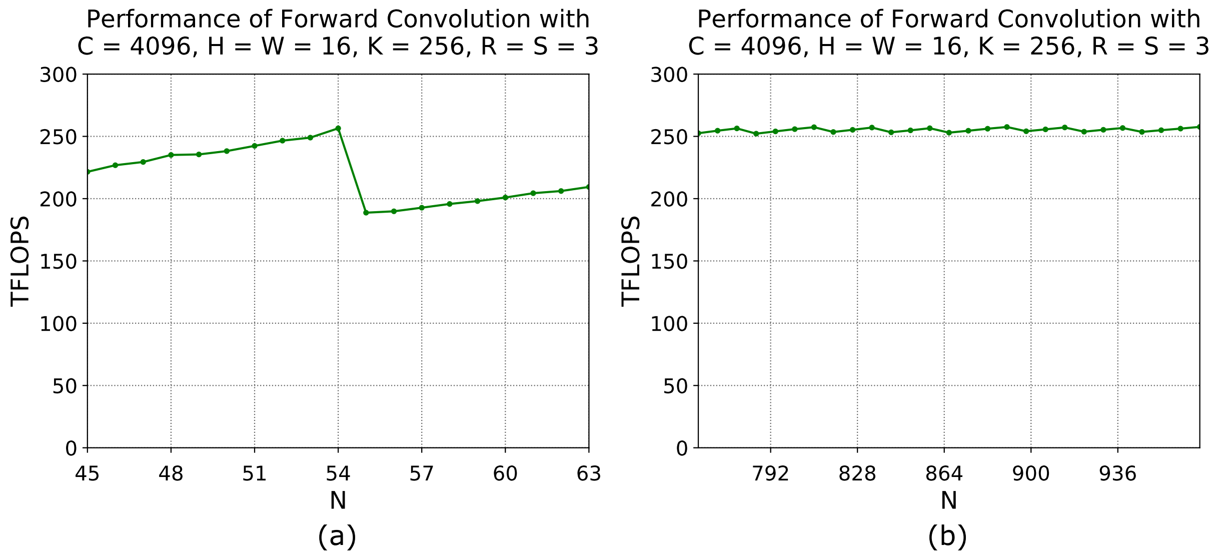
For example, computing a 3x3 convolution on a 256x56x56x64 input tensor, producing a 256x56x56x128 output, all in half-precision, has an arithmetic intensity of 383.8 FLOPS/byte.

4.3. Quantization Effects

Tile and wave quantization effects can be significant, especially for small problem sizes (background in Dimension Quantization Effects in the *Matrix Multiplication Background User Guide*). Just like for GEMMs, in implicit GEMM, the representation of the output matrix is divided into tiles of a chosen size, and that set of tiles is distributed across available multiprocessors.

Our testing GPU has 108 SMs. Each SM can handle a number of thread blocks in parallel that is dependent on the kernel being used; for best parallelization, an implicit GEMM should contain an integer multiple of 108 tiles.

Figure 4. Graphs showing the performance of convolution with filter size 3x3, input size 16x16, 4096 channels of input, and 256 channels of output. Best performance is observed when N is divisible by 108 (when a multiple of 216 tiles, two in parallel on each SM, are created). NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



As Figure 4 shows, convolutions that result in small equivalent GEMMs can exhibit significant quantization effects. When `N = 54`, 216 tiles are created; when `N = 55`, 220 tiles are created.

The former results in high Tensor Core utilization, while the latter will require an additional wave to process the remainder of 4 tiles, severely impacting performance (Figure 4 (a)). Once the convolutions are reasonably large, the effect is less pronounced (Figure 4 (b)).

It is worth noting that weight gradient quantization does not behave as implied by the GEMM dimensions in Figure 3; for quantization purposes, the height of the matrix to be tiled should be seen as C (rather than `C*R*S`). This is discussed in more detail at the end of the section on filter size.

4.4. How Convolution Parameters Affect Performance

In this section, we discuss the trends affecting performance. To keep things simple, padding is set such that `H = P` and `W = Q`, and both the stride and dilation are equal to one unless indicated otherwise.

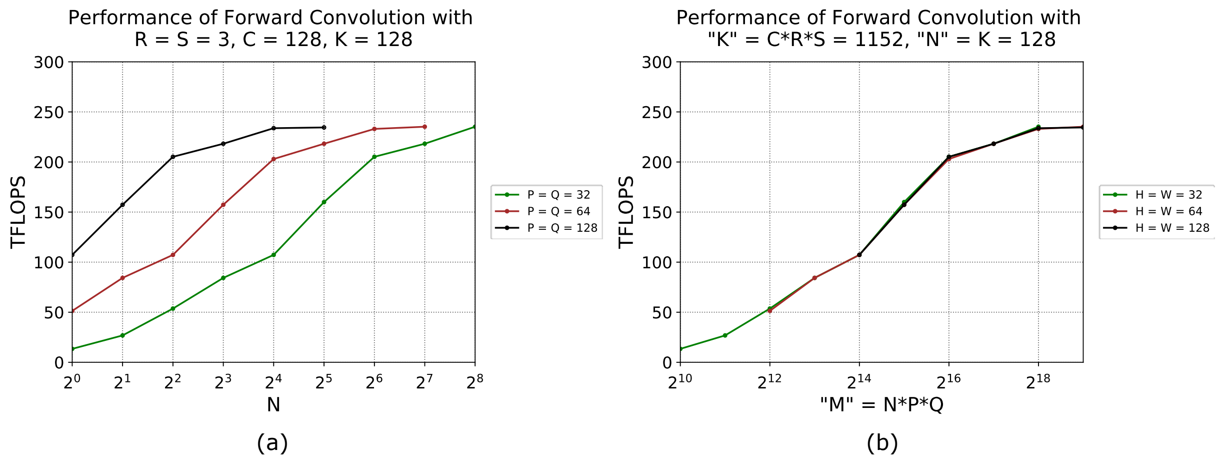
4.4.1. Batch Size, Height And Width

When representing a forward convolution as a GEMM, the product of batch size, output height, and output width (`N*P*Q`) is the "M" dimension of the unrolled input tensor (A matrix) as well as the output (C matrix).

The individual values of these parameters are not especially important to GEMM performance; only the final dimension, the product `N*P*Q`, is significant. Conveniently, in most applications, batch size may be changed more easily than the parameters contributing to the height and width of the output tensor.

Generally, efficiency improves as `N*P*Q` increases, with diminishing returns. From Figure 5, we can see that points with equivalent `N*P*Q` has roughly equivalent performance, as the corresponding GEMMs have the same dimensions.

Figure 5. Different perspectives on the same data. Performance improves with increased N*P*Q. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



When calculating the activation gradient, `N*H*W` is the "M" dimension of the equivalent GEMM (compared to `N*P*Q` in forward convolution). With a filter stride of 1, the performance of forward convolution and activation gradient calculation will be roughly the same.

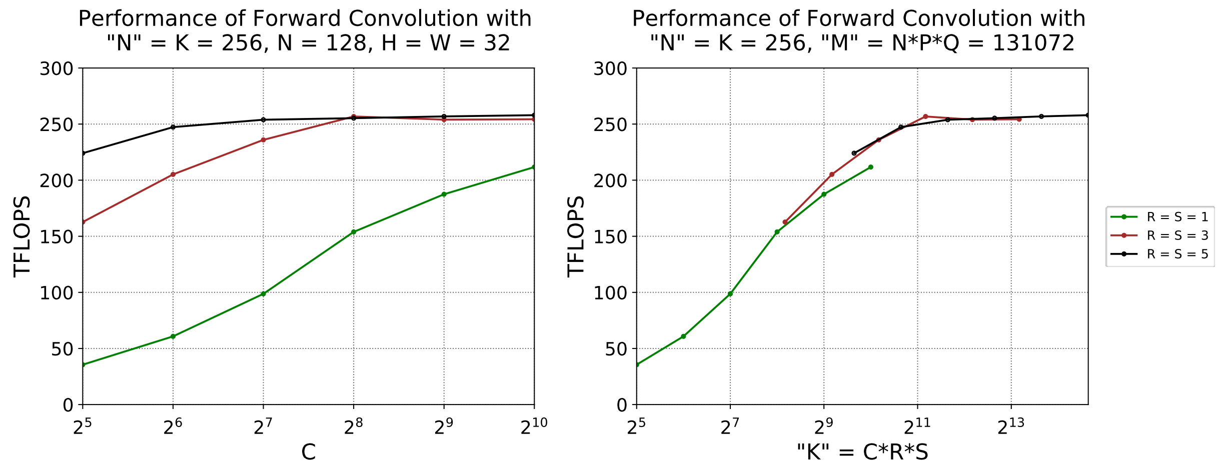
In contrast, for weight gradient calculation, `N*P*Q` becomes the accumulation ("K" in a GEMM) dimension. The performance impact of this dimension is not as straightforward, as it does not affect the tiling of the output matrix in any way. However, larger values of `N*P*Q` generally leads to more time spent multiplying and accumulating elements, rather than in setup and teardown overhead for the GEMM computation, improving the fraction of peak performance achieved for the whole operation. It is worth noting that cuDNN generally supports tiling in the `N*P*Q` dimension as well for weight gradients, as many common layers result in tiny (for example, 64x64 in the first block of a standard ResNet) output matrices that, by themselves, don't offer enough tile parallelism to keep the GPU occupied.

4.4.2. Filter Size

The equivalent GEMM for forward convolution has a "K" dimension of `C*R*S`. As mentioned previously, the "K" dimension does have an impact on performance, and this effect is most pronounced for small GEMMs.

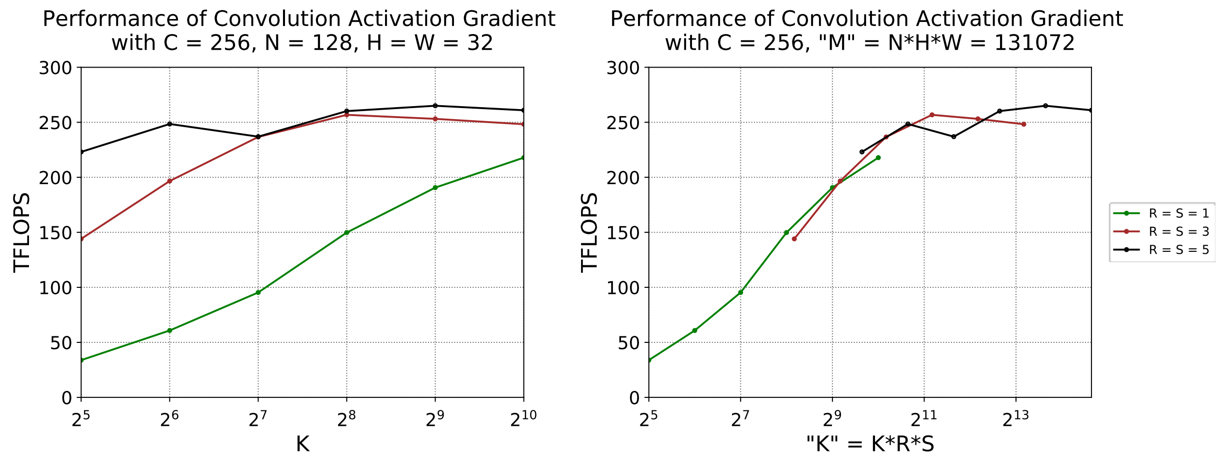
When using a 1x1 filter, layers with more input channels tend to perform better in forward convolution (Figure 6), as it is ultimately the `C*R*S` product that matters.

Figure 6. Convolutions with larger filters tend to perform better (C*R*S matters). NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



When calculating activation gradients of the convolution, K affects this dimension instead: `"K" = K*R*S`. There is a clear similarity between Figure 6 and Figure 7; in general, trends related to C for forward convolution are related to K for activation gradient calculation and vice versa.

Figure 7. Near-identical relations as in Figure 22, but involving K. For activation gradients, K*R*S matters. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



The weight gradient calculation has `"M" = C*R*S`, therefore the impact of filter size on performance is similar to that discussed for batch size, height, and width previously; larger values tend to perform better. When considering tile quantization, however, the weight gradient algorithm differs from the forward and data gradient ones; only the C dimension, not the full `C*R*S` dimension, quantized against the tile size (meaning, filter size parameters can be ignored). For example, when using 64x64 tiles, if `C = 32`, half of each tile (vertically) is wasted regardless of R and S; only the value of C matters.

4.4.3. Channels In And Out

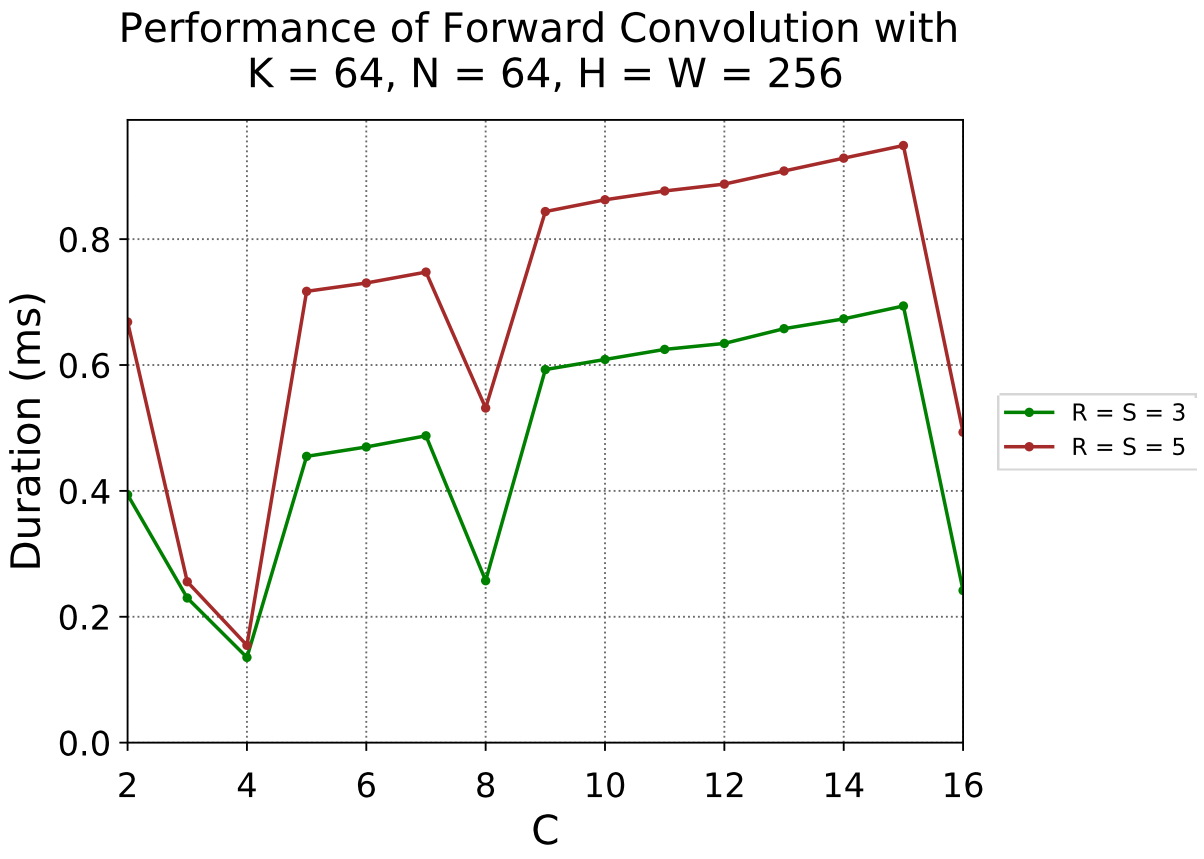
Requirements to enable Tensor Cores depend on the cuDNN version used. With cuDNN v7.6.3 and later, convolution dimensions will automatically be padded where necessary to leverage Tensor Cores. Earlier versions of cuDNN are stricter: using Tensor Cores with NHWC-packed data requires C and K to be aligned to multiples of 4 with TF32, 8 with FP16, or 16 with INT8. For NCHW-packed FP16 data, channels will be automatically padded to multiples of 8 such that Tensor Cores will be enabled. However, using NCHW data with Tensor Core enabled kernels involves some additional transpose costs, which are discussed in Tensor Layouts In Memory: NCHW vs NHWC.

On the other hand, with these earlier versions of cuDNN, automatic padding doesn't kick in for NHWC-packed data, so a less-efficient fallback kernel, which does not make use of Tensor Cores, is chosen. Convolutions with NHWC data do perform better than those with NCHW data given that C and K are divisible by 8. In other words, if a layer is already being used with NCHW data, automatic padding will occur; however, if NHWC data is being used, choosing or padding C and K to be a multiple of 8 improves performance.

For cuDNN v7.6.3 and higher, padding is automatic regardless of data format. Padding adds some time, although this cost is typically negligible compared to the gain in performance from enabling Tensor Cores. It is worth noting that choosing C and K to be multiples of 8 for FP16 or the equivalent for other data types gives the best efficiency: for these cases, padding is not required.

There are cases when channel counts are small and non-negotiable. For the first layer in a network, it is common to have a very small value of C (1 or 3 for grayscale and RGB or YCrCb images, respectively). Special-case convolution implementations are available to meet this need, specifically for `C = 4` and a stride of 2 (Figure 8). The data shown here was collected using cuDNN 8.1, so padding is automatic. Performance improvement from `C = 3` to `C = 4` is less drastic here than with versions earlier than 7.6.3, but choosing `C = 4` is still faster as no padding occurs.

Figure 8. Specialized kernels for C = 4 speed up common first layers in convolutional neural nets (NHWC data used). Choosing C = 4 or a multiple of 8 gives best performance. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.

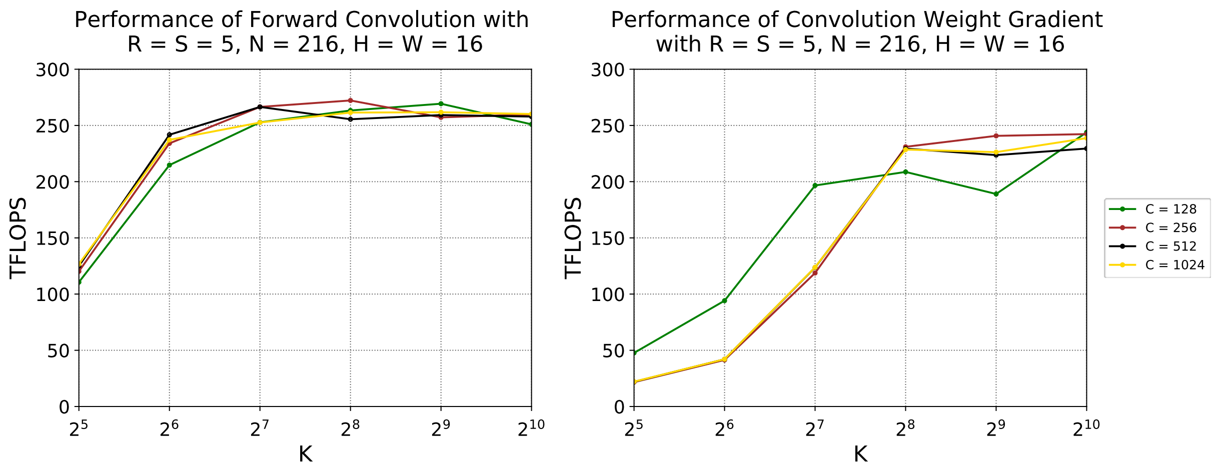


Forward convolution performance relative to C was previously discussed in Filter Size (`"K" = C*R*S`), as was activation gradient calculation relative to `K ("K" = K*R*S)`. The effect of C on weight update performance (with "M" = C*R*S) is mentioned in Batch Size, Height And Width. In short, larger values usually give higher efficiency, with diminishing returns.

The number of channels of input and output can have a more immediate impact on performance, however; the GEMM dimension "N" is equal to C or K for all of the forward convolution, activation gradient calculation, and weight gradient calculation. Thus, your choosing of these parameters can have a direct impact on performance.

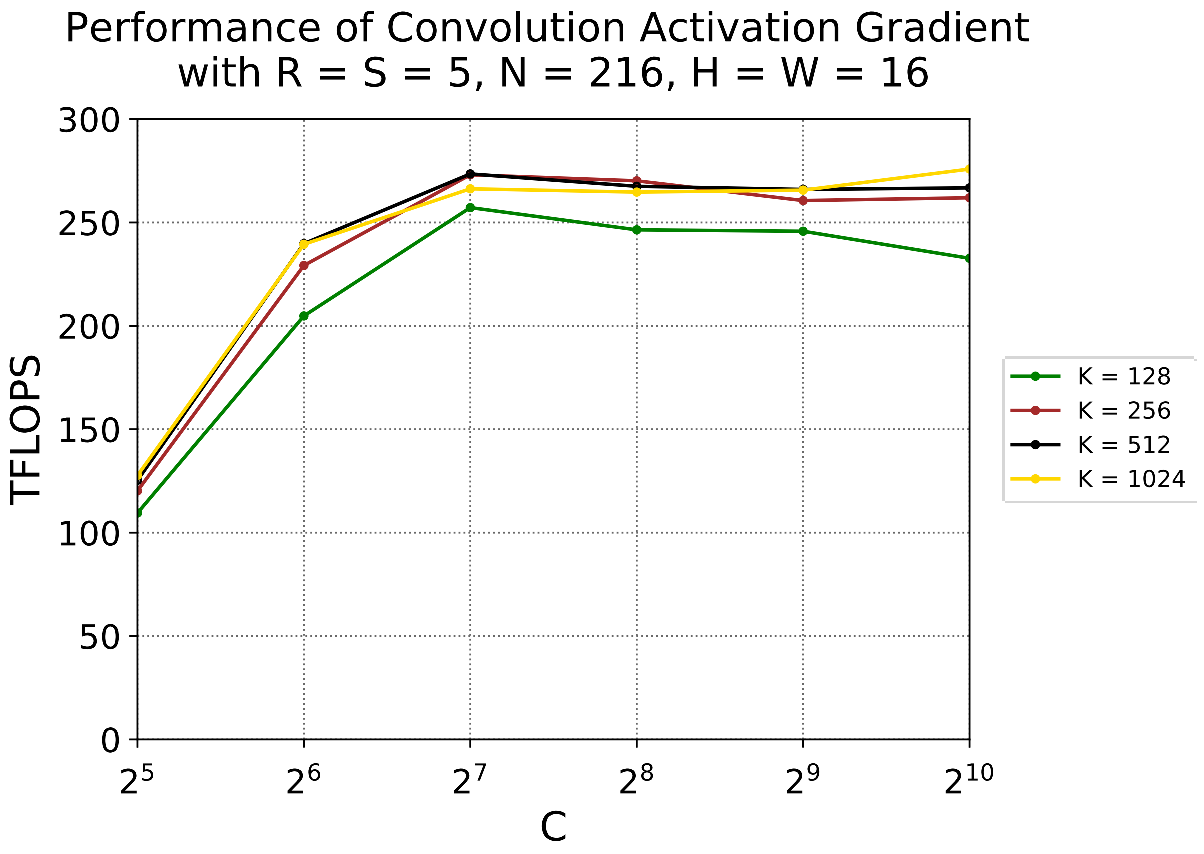
Similar behavior can be seen in forward convolution and weight gradient computation as varied across K (Figure 9). In both cases, `"N" = K`, resulting in a strong trend for small values of K and diminishing returns once K is larger than most tile sizes.

Figure 9. Forward convolution and weight gradient computation performance is much better for larger K, up to a point. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



The same effect is present for channels of input in activation gradient computation (`"N" = C`), as seen in Figure 10. As previously mentioned, the effect of C on activation gradient computation tends to match the effect of K on forward convolution.

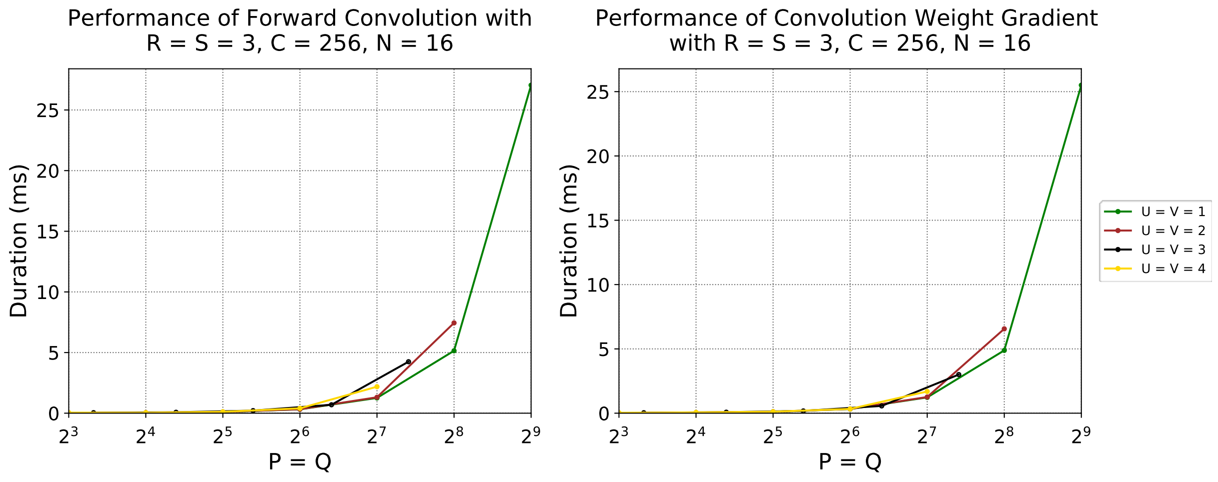
Figure 10. Activation gradient calculation performance improves as C increases, with diminishing returns. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



4.5. Strides

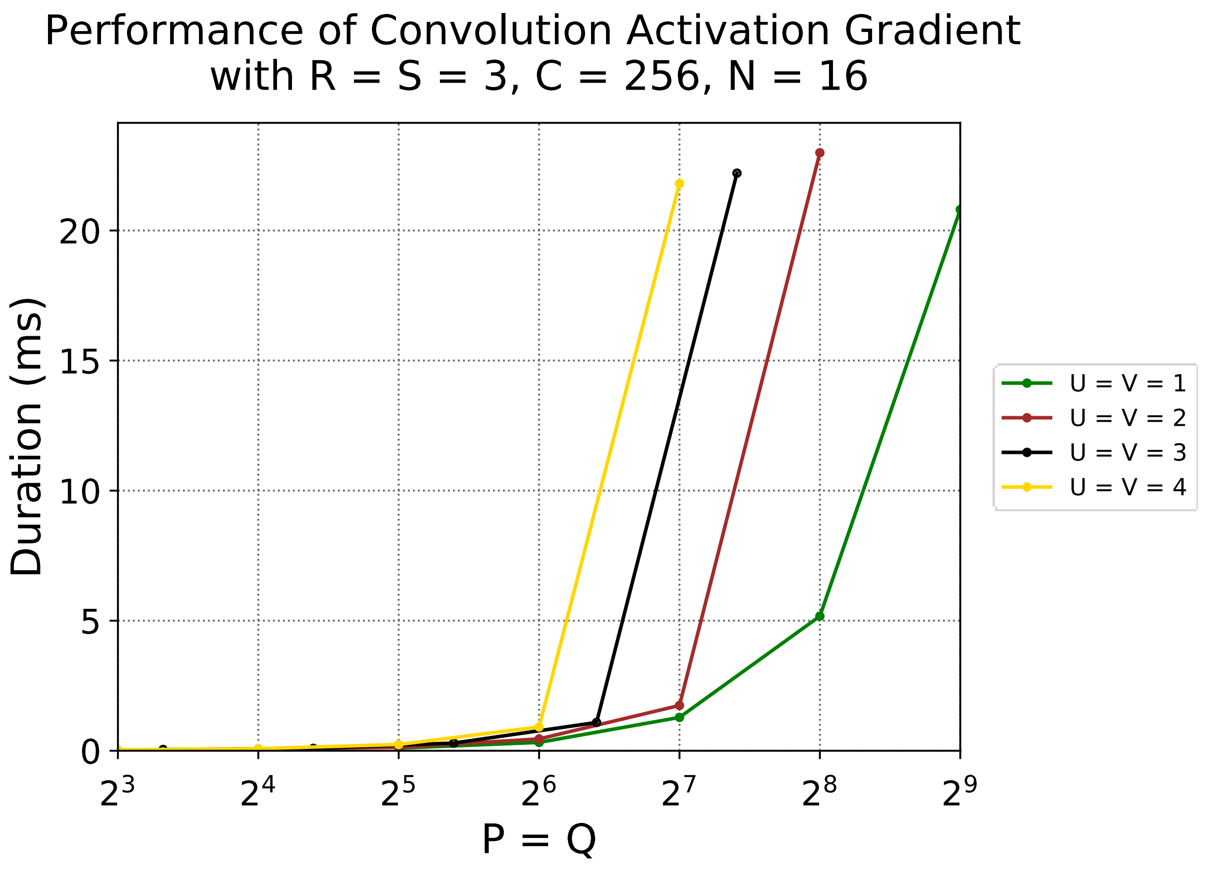
Filter strides (U and V) impact performance mostly via their effect on input and output tensor dimensions. Using a horizontal and vertical stride of 1, H and W are roughly equal to P and Q respectively, depending on the filter size and padding. However, when larger strides are used, there is a several-fold difference in the size of input and output feature maps. In turn, this impacts GEMM dimensions and performance.

Figure 11. Performance of forward convolution and weight gradient calculation is relatively unaffected by variations in stride or input height and width as long as output height and width are constant. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



Forward convolution and weight gradient calculations perform similarly for equal P and Q, although H, W, U, and V vary (Figure 11).

Figure 12. Activation gradient computation performance is not exclusively governed by output dimensions; input height and width have an impact. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.

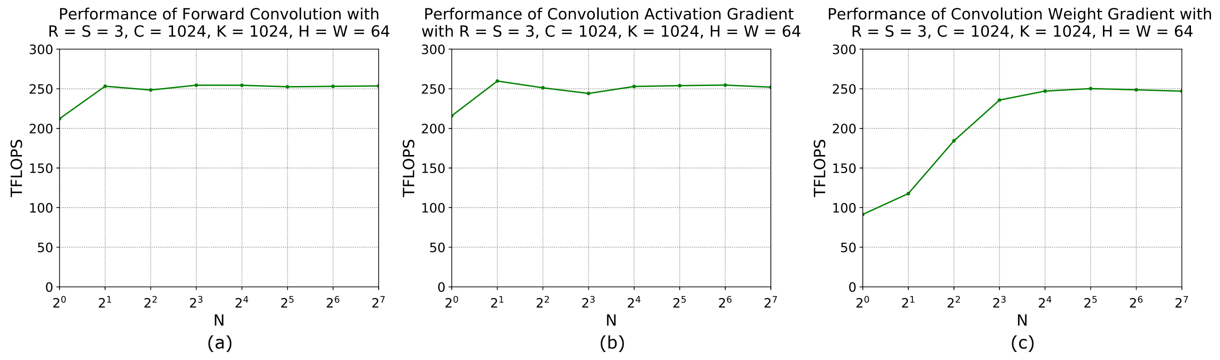


In contrast, activation gradient calculation is affected more heavily by input feature map size (Figure 12).

4.6. High-Performance Example

An example of a convolution with high performance is shown in Figure 29. This scenario is based on a convolutional layer with input feature maps of size 64x64, filters of size 3x3, 1024 input channels, and 1024 output channels; each parameter is reasonably large. NHWC data is used to avoid overhead from additional transposes. Input and output channel numbers are divisible by 8, so Tensor Cores will be enabled.

Figure 13. Good performance can be seen for the convolutions with larger batch sizes here, close to 250 TFLOPS for all three passes. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



At larger batch sizes, performance for this case is approximately 250 TFLOPS for forward convolution and activation and weight gradient calculation.

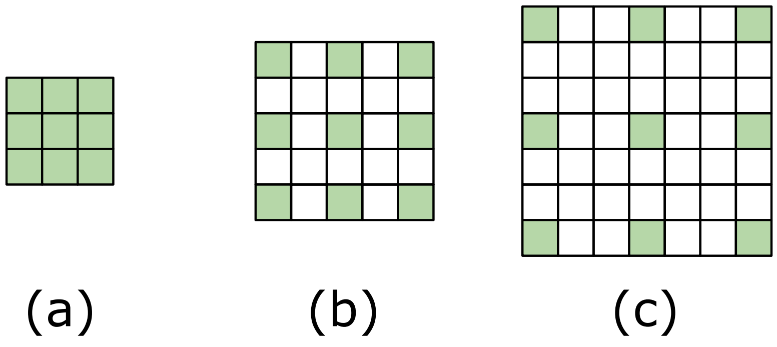
Chapter 5. Convolution Variants

5.1. Dilated

Dilated convolutions are a variant of a regular convolutional layer that effectively expands the filter being applied by inserting zeros between filter elements.

The dilation factor is one greater than the number of zeros added between each pair of elements. As a result, the overall 2D area overlapping with each channel of the filter increases.

Figure 14. Dilation of a 3x3 filter; dilation factors are (a) 1, (b) 2, and (c) 3.

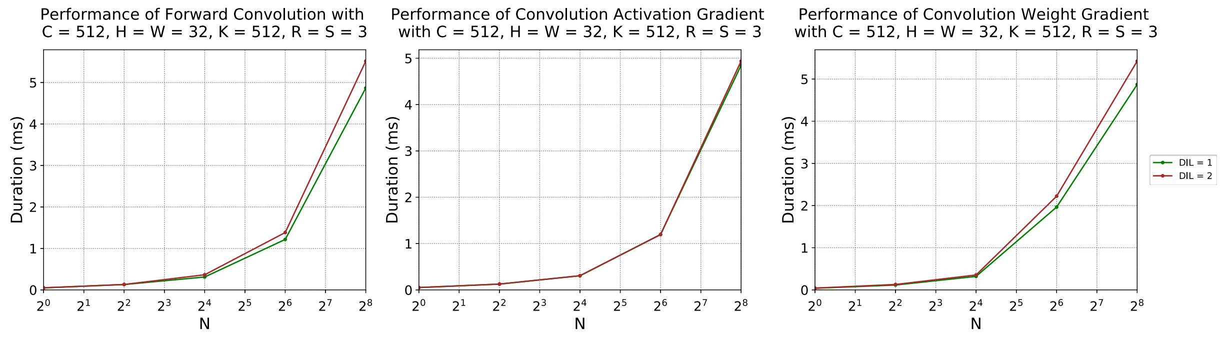


$$r_{\text{effective}} = \text{dilation}_h \cdot (r - 1) + 1$$

$$s_{\text{effective}} = \text{dilation}_w \cdot (s - 1) + 1$$

Choice of dilation factor affects how a convolution is represented as a virtual GEMM, but does not actually change the dimensions of that GEMM; therefore, performance for forward convolution and activation gradient computation is similar regardless of dilation factor (shown below).

Figure 15. Performance of forward convolution, activation gradient computation, and weight gradient computation is very similar for a non-dilated convolution and one with a dilation factor of 2. NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.


Performance of Forward Convolution with C = 512, H = W = 32, K = 512, R = S = 3
Performance of Convolution Activation Gradient with C = 512, H = W = 32, K = 512, R = S = 3
Performance of Convolution Weight Gradient with C = 512, H = W = 32, K = 512, R = S = 3
Duration (ms)
N
DIL = 1
DIL = 2