Received April 30, 2019, accepted May 21, 2019, date of publication May 24, 2019, date of current version June 11, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2918851*

# Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs

**MARC JORDÀ, PEDRO VALERO-LARA, AND ANTONIO J. PEÑA**
Barcelona Supercomputing Center, 08034 Barcelona, Spain

Corresponding author: Marc Jordà (marc.jorda@bsc.es)

This work was supported by the European Union's Horizon 2020 Research and Innovation Program under the Marie Sklodowska-Curie under Grant 749516, and in part by the Spanish Juan de la Cierva under Grant IJCI-2017-33511.

**ABSTRACT** Convolutional neural networks (CNNs) have recently attracted considerable attention due to their outstanding accuracy in applications, such as image recognition and natural language processing. While one advantage of the CNNs over other types of neural networks is their reduced computational cost, faster execution is still desired for both training and inference. Since convolution operations pose most of the execution time, multiple algorithms were and are being developed with the aim of accelerating this type of operations. However, due to the wide range of convolution parameter configurations used in the CNNs and the possible data type representations, it is not straightforward to assess in advance which of the available algorithms will be the best performing in each particular case. In this paper, we present a performance evaluation of the convolution algorithms provided by the cuDNN, the library used by most deep learning frameworks for their GPU operations. In our analysis, we leverage the convolution parameter configurations from widely used the CNNs and discuss which algorithms are better suited depending on the convolution parameters for both 32 and 16-bit floating-point (FP) data representations. Our results show that the filter size and the number of inputs are the most significant parameters when selecting a GPU convolution algorithm for 32-bit FP data. For 16-bit FP, leveraging specialized arithmetic units (NVIDIA Tensor Cores) is key to obtain the best performance.

**INDEX TERMS** Neural network, convolution, deep learning, cuDNN, GPU, volta.

## I. INTRODUCTION

Deep neural networks (DNNs) have received considerable attention in recent years due to their outstanding results in applications such as image classification and segmentation, natural language understanding, or speech recognition [14], [17], [20]. In particular, convolutional neural networks (CNNs) have positioned as one of the most efficient ways to address this type of problems, yielding remarkable performance in terms of computing time and accuracy [10].

This emergence has been motivated by two main factors. First, the availability of overwhelmingly large input datasets, annotated with the expected output, which are required to fine-tune the network parameters to attain high accuracy. A relevant example in the context of object recognition is the ImageNet project [6], which provides over fourteen million images manually classified into twenty thousand different categories. Second, the high throughput of modern heterogeneous computing systems, equipped with accelerators such as graphics processing units (GPUs), which are necessary to reduce the training time of large CNNs from weeks to a few days or even several hours. This reduction in training time enables researchers to iterate on the design of a CNN within a reasonable time frame. High-performance systems are also necessary to reduce inference time for latency sensitive applications such as pedestrian detection in autonomous vehicles [2].

CNNs are characterized by the inclusion of convolutional layers as part of the neural network. As in a fully-connected layer, an output element of a convolutional layer is the result of applying a non-linear function (activation) to the weighted sum of several input elements; however, there are two main differences. (1) Each output only depends on a small set of input values (e.g. a 2D tile of an image), instead of being a function of all the input elements; and (2) convolutional

The associate editor coordinating the review of this manuscript and approving it for publication was Shuping He.

layers use a much smaller number of weights, which are shared by all output computations. For 2D inputs, such as images, the number of weights is a multiple of the size of the input tile considered for each output (i.e., a set of 2D weight matrices with the size of the input tile), instead of featuring one weight per each pair of input–output elements. To implement the weighted sums (with shared weights) of tiles of input elements, convolutional layers use a mathematical operation named ''convolution'', which is the origin of their name. The layer's weights are frequently named ''filters'' or ''kernels''.

In terms of benefits compared to fully-connected layers, convolutional layers feature reduced storage and computational costs, and these costs no longer depend on the input and output size. Instead, these layers are defined by a set of hyperparameters (like the size of the 2D tiles of weights), which are defined by the designer of the CNN. In the context of image classification and segmentation tasks (a major field of application of CNNs) convolutional layers yield another important advantage, commonly referred as ''translation invariance''. Since the filters are tuned during the training stage (as any other weight of the neural network), and these are reused in several positions over the image, filters ''learn'' to detect patterns independently of the position where these appear in the image. In contrast, the weights of a fully connected layer are position dependent. Thus, to be able to detect a given pattern in any position of an image, a fully connected layer requires a much more complex training dataset with examples of this pattern in all possible positions.

Convolutional layers may be understood as a form of automatic feature extractors. This provides CNNs a great advantage over other classification/detection methods where the feature extractors have to be manually designed to detect the relevant patterns. Such design needs to be performed by experts in the field and is often time consuming. Contrarily, in CNNs, the features are learned automatically by the convolutional layers as part of the neural network training.

In this paper we present a performance survey of the state–of–the–art of production convolution algorithms used in the context of CNNs: matrix–matrix multiply (GEMM), fast Fourier transform (FFT), and Winograd. We evaluate vendor-provided implementations of these algorithms in the most recent high-end computing and deep learning platform based on GPU technology. cuDNN, provided by NVIDIA as a fine-tuned library for its GPUs, is supported by most deep learning frameworks used in production, such as TensorFlow [1], PyTorch [26], or Caffe2 [8]. We analyze the performance of the cuDNN implementation of the aforementioned algorithms on 602 different convolution parameter configurations, and discuss which parameters are more relevant to select the best performing algorithm for a given convolution configuration. We also profile the execution of several representative configurations to further explain the reasons behind the observed performance behaviors.

The convolution parameter configurations used in our experimental evaluation are extracted from five of the most well known CNNs, a total of over 7,000 test cases. Doing this, we ensure that the performed analysis is focused on the most used and widely known convolution configurations, in order to obtain relevant conclusions for the neural network community.

The results we present in this paper were obtained using the latest NVIDIA GPU for high-performance computing (Tesla V100), which is particularly well-suited for computing convolutions, using the most recent versions of CUDA and cuDNN (9.1 and 7.1, respectively). In this article we provide an extended discussion of the experimental results and a set of guidelines to aid in the selection of the fastest algorithm depending on the convolution parameters. To the best of our knowledge, this article presents the first in-depth performance analysis of all available implementations of convolution algorithms in the latest NVIDIA platform.

The rest of this manuscript is distributed as follows. Section II introduces some background information relevant to understand the subsequent sections. Section III presents the analysis objective of this work. Section IV provides recommendations based on the previous analysis. Section V summarizes our findings.

## II. BACKGROUND

In this background section we introduce the details necessary to understand the rest of this article on convolutions, convolutional layers in CNNs, and convolutional algorithms.

### A. CONVOLUTION

Convolutions are widely used in several domains. For instance, in signal processing convolutions are used to compute the output of linear time-invariant (LTI) systems given an input signal and the impulse response of the system. In probability theory, on the other hand, convolutions are used to compute the probability density function of the sum of two independent random variables.

In general terms, a convolution is a mathematical operation that computes the integral of the product of a function $f$ with the reversed and translated version of a function $g$. It may be understood as a weighted moving average of $f$, where $g$ are the weights:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(u)g(x-u)du$$

A discrete convolution features a similar definition, with the two input functions being discrete sequences:

$$(f * g)[n] = \sum_{m} f[m]g[n-m]$$

### B. CONVOLUTIONAL LAYERS IN CNNs

Convolutional layers are composed by a set of 3D filters (also called ''kernels'') and receive a set of 3D inputs (usually referred as ''input batch''). The convolution is performed on each input–filter pair, resulting in a set of 3D outputs. The convolution of an input–filter pair generates one X-Y plane of the output. The results of the convolutions of one

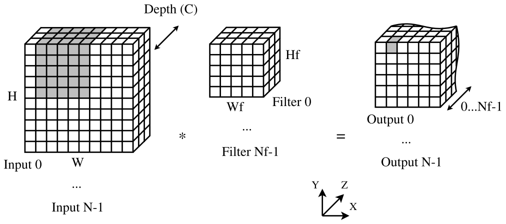

**FIGURE 1.** **Convolution operations in a convolutional layer. The grayed output element is the result of the scalar product of Filter 0 with the grayed patch of input elements performed during the filter translation over the input.**

particular input with all the filters are stacked in the Z direction, generating the corresponding 3D output (see Figure 1). The same process is applied to all the inputs of the batch, resulting on a set of 3D outputs. The sets of inputs, filters, and outputs may be seen as 4D arrays (also known as ''tensors''), corresponding to the 3 dimensions of each input/filter/output, plus the dimension to identify each of these within the set. Mathematically, each output element is computed using the following formula:

$$Out_{n,i,j,k} = \sum_{x=0}^{W_f-1} \sum_{y=0}^{H_f-1} \sum_{z=0}^{C-1} Filter_{k,x,y,z} \cdot In_{n,i+x,j+y,z}$$

where the subscripted coordinates represent the input/filter/output index, followed by the X, Y, and Z coordinates of a given element.

The final step in a convolutional layer consists of adding an offset array called ''bias'' in the convolution outputs to subsequently apply a non-linear activation function, such as *sigmoid* or *ReLU* [27], as in fully-connected layers. Since the cost of these operations is negligible with respect to the convolutions, we do not include them in our analysis.

In the context of CNNs, we use the term ''convolution'' to mean ''2D discrete convolutions''. In fact, using a strict mathematical terminology, a convolution is a 2D discrete cross-correlation, since the filter is not reversed as stated in the mathematical definition of convolution.

In CNNs, since the filter weights are learned during the training, using cross-correlation instead of convolution does not affect the final network results; it only changes the actual values learned. The advantage of cross-correlation is that it avoids the additional step of flipping the filters to perform the convolutions.

Convolutions in CNNs work on 3D arrays, even though they are bi-dimensional. Basically, these operations consist of applying a given filter on the different portions of an input. This is known as ''translation''. Both filters and portions must feature the same size. The translation of the filter over the input is performed in 2 dimensions (X and Y dimensions in Figure 1), where the distance in between each translation position is defined by the ''stride'' parameter. In each position of the result, a 3D tile (portion) of the input, with the same size as the filter, is point-wise multiplied with the filter elements. Then, the results of all multiplications are added to obtain a single output element (the grayed elements in Figure 1 are those involved in one of the translation positions of the filter). The input and filter size in the Z dimension must be equal in order to feature matched pairs in the point-wise multiplication. This process may also be understood as performing regular 2D convolutions in between X-Y planes on the input with the corresponding X-Y plane of the filter, and adding the resulting matrices to obtain an X-Y plane of the output.

There are two more convolution parameters, called ''padding'' and ''dilation''. Padding defines the number of elements added to the edges of the X and Y dimensions of the input, usually with value 0. It effectively increases the size of the input, which in turn increases the number of positions where the filter is applied during the translation. Padding is used to increase the X and Y dimensions of the output, because a convolution without padding (as that illustrated in Figure 1) produces an output smaller than the input. Dilation also defines a sort of padding; in this case, it is the number of elements inserted in between the filter elements in the X and Y dimensions. This internal padding enlarges the filter's X-Y plane without increasing the computational cost because the operations with the padding zeros are not actually performed. Dilation is used in the opposite case of padding, when a larger reduction of the output size is desired, while avoiding the additional cost of employing larger filters.

Besides the introduction of a new layer type, CNNs do not differ further from conventional DNNs. The filters in convolutional layers are updated iteratively during the network optimization process (training stage), like the rest of the network's weights. Similarly, once the CNN is deployed, the trained filters are used in the convolutions involved in computing the network's output (inference stage).

The main advantage of convolutional layers is their low storage and computational cost, compared to fully connected layers with the same input and output sizes. For example, for an input and output image of 50 × 50 pixels, a fully connected layer would feature $50^4 = 6,250,000$ weights and $50^2 = 2,500$ biases, totaling 6,252,500 parameters. In contrast, the number of parameters of a convolutional layer does not depend on the X-Y size of input and output; it depends on the filter size and the number of filters. The filter size is a hyperparameter chosen by the designer of the CNN, while the number of filters depends on the desired output depth. Following with the previous example, where depth $= 1$, the total number of parameters of a convolutional layer with a filter of 10 × 10 would be $10^2 + 2,500 = 2,600$.

Such a reduction in the number of weights is obtained by their reuse in the computation of all the output elements, since the filter is translated over the input by the convolution operation. Moreover, in a convolutional layer, each output element only depends on a small subset of the input elements instead of being a function of all, as in fully connected layers. These input elements, sometimes called the ''receptive field'' of an output element, are those involved in the scalar product with the filter (e.g. grayed elements in Figure 1).

Larger filters increase the size of the receptive field, which may help in learning more complex patterns, but at the

**TABLE 1.** Summary of features of the convolution operations (with stride 1) present in the five well-known CNNs we selected for our evaluation.

| | GoogleNet | SqueezeNet | AlexNet | Resnet50 | VGG19 |
|---|---|---|---|---|---|
| **# Distinct convolution configurations** | 42 | 21 | 4 | 12 | 9 |
| **Network input size** | 224×224×3 | 224×224×3 | 224×224×3 | 224×224×3 | 224×224×3 |
| **Input size to last convolutional layer** | 7×7×832 | 13×13×512 | 13×13×384 | 7×7×1024 | 14×14×512 |
| **Convolution filters sizes (% of conv. configs.)** | 1×1 (57.2%)<br>3×3 (23.8%)<br>5×5 (19%) | 1×1 (71.4%)<br>3×3 (28.6%) | 3×3 (75%)<br>5×5 (25%) | 1×1 (66.7%)<br>3×3 (33.3%) | 3×3 (100%) |

expense of increasing the computation cost and number of parameters. It is worth noting that reusing the filters in several positions of the input renders the learned parameters translation-invariant: if a filter learns to detect a pattern in the images, it will detect it no matter where this pattern appears within the image. To prevent different filters from learning to detect the same patterns, these are usually initialized with random values [9], [18], [24].

The training of the filters may be seen as a form of automatic definition of feature extractors. This provides a great advantage over other classification/detection methods, where the feature extractors have to be manually designed by an expert. Moreover, the design of feature extractors is often application-specific, which renders the automatic feature extraction of CNNs even more appealing.

### 1) CONVOLUTION PARAMETERS IN CNNs

Most convolutional neural networks are shaped similarly, which defines some common features of this type of networks. As an example, in Table 1 we include information from the five well-known CNNs we use in our evaluation, AlexNet [17], VGG19 [28], GoogleNet [29], Resnet50 [13], and SqueezeNet [15]. To simplify the design, most CNNs use square inputs, filters, and outputs, even though different X and Y sizes could be possibly used. The inputs of the network (e.g. images) are scaled to match the shape and resolution expected by the network (around 200 × 200 is a common input resolution). The initial depth is usually small; for example, in the case of images, the 3 RGB color channels are stacked in the Z dimension to obtain an initial depth of 3. As the inputs progress through the network, their size and depth follow an inverse trend. The depth increases progressively, attaining sizes from several hundreds to a few thousands of elements, as in the case of Resnet50, where depth is up to 1,024 (see *input size to last convolutional layer* in Table 1). Contrarily, the input X-Y size is reduced as we progress deeper, being around 10 × 10 in the latest layers. This reduction is usually implemented with pooling layers inserted at certain points of the neural network. Most CNNs use convolutions with a padding of $\frac{W_f-1}{2}$ elements in the X dimension and $\frac{H_f-1}{2}$ elements in the Y dimension (and stride = 1) to obtain outputs with the same X and Y dimensions as the inputs. Each pooling layer halves the input X and Y sizes by reducing tiles of 2 × 2 elements to a single element (using maximum or arithmetic average as the reduction function). All these features are seen in the most popular and widely used CNNs, such as AlexNet [17], VGG19 [28], GoogleNet [29], Resnet50 [13], and SqueezeNet [15].

On the other hand, the convolution filters tend to be small during the process. Some initial convolutional layers may feature filters of around 10 × 10, but the most common sizes are 1 × 1, 3 × 3, and 5 × 5, as observed in Table 1. 1 × 1 filters are a special case because each convolution output will only depend on data over the Z dimension of a given X and Y position, instead of being influenced by a central point and the neighboring elements. In CNNs, these are placed before 3 × 3 or 5 × 5 filter convolutions, as a way to limit their computational cost by reducing the depth of an incoming input batch.

The convolutional layer configurations used in training and inference are the same. In the inference stage, once the CNN is deployed, the trained filters are used to process the input and perform the task at hand.

### 2) DATA TYPE REPRESENTATION IN CNNs

The most common data type to represent values in CNNs is 32-bit floating point; however, there are several efforts to use data representations with lower precision like 16-bit floating-point or even 8-bit integers. Even hardware vendors started including native support for reduced-precision arithmetic in their latest products (e.g., NVIDIA Tensor Cores included in Volta GPUs [21]). Some studies focus on benefiting from the reduced bandwidth and storage requirements of using smaller data types to accelerate the training [3], [5], [11], [12], [23]. However, using data types with less precision poses difficult challenges to the iterative weight-optimization process performed during the training stage in terms of stability (i.e., no convergence) and the final obtained accuracy. Others leave the training stage as usual and apply a quantization process to the trained weights to reduce the computational cost of the inference stage [16], allowing CNN deployment in embedded systems and mobile phones. Due to this growing interest in lower precision data types, we perform our experimental evaluation using both 32 and 16-bit floating-point values.

## *C. CONVOLUTION ALGORITHMS*

Several algorithms may be used to compute a convolution, besides the direct application of the convolution formula. A GEMM-based algorithm transforms the inputs and filters to be able to exploit high-performance matrix–matrix multiply operations. There are two other relevant convolution algorithms, both based on arithmetic strength reduction (i.e., reducing the number of multiplications while increasing the number of additions). One of these is based on FFT operations, computing the convolution in the frequency domain [30]. The other relies on Winograd's minimal

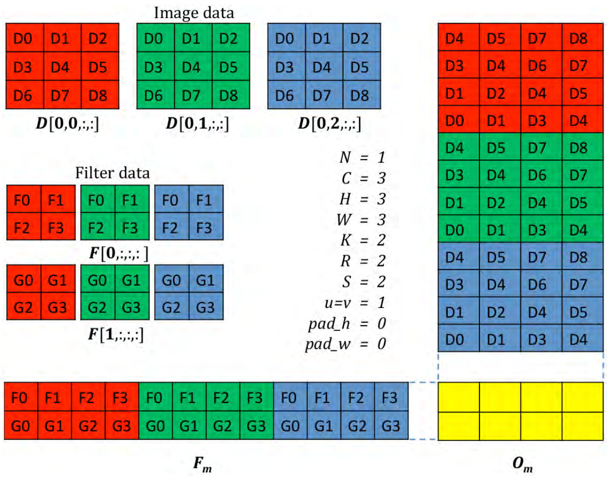

**FIGURE 2.** **Data transformation performed for the GEMM convolution approach [4].**

filtering algorithms [19], known in the context of signal processing to apply finite impulse response (FIR) filters.

### 1) GEMM

One of the most widely used algorithms for convolution is based on the BLAS GEMM operation [4]. It consists of transforming the input into a large matrix, where each column contains the input elements involved in each filter position during the convolution. Another matrix is generated with the elements of each filter ($F_m$ in Figure 2). Finally, the result of the convolution is provided by the matrix multiplication (GEMM) of both matrices, as shown in Figure 2. The elements of the transformed input matrix are placed in such a way that when the GEMM performs the scalar product of one row and one column, it matches the scalar product that would be performed by the convolution. Since the matrix multiplication performs the scalar product of all columns from the transformed input with all rows of the transformed filters, the result of the GEMM operation represents the outputs of the convolution for all filters over all the filter translation positions over the input. The function used to obtain the transformed input matrix is often called ''im2col'', according to its Matlab implementation.

This approach benefits from the highly efficient implementations for matrix–matrix multiplication on GPUs and the nature of GPU architectures, strongly suited for performing these operations [7]. However, this approach requires a significant amount of memory to store the transformed matrices, especially the transformed input matrix, which is larger than the original input because it has to store duplicated elements due to the overlap of the filter positions in the convolution.

### 2) FFT

FFT-based convolution implementations are a well known technique [22], [30]. According to the Convolution Theorem, the Fourier transform of the convolution of two inputs is equal to the product of the Fourier transforms of the same inputs. Thus, convolution in the time domain is equal to element-wise multiplication in the frequency domain. Since multiplication is less expensive to compute than convolution, the benefit of computing a convolution in the frequency domain depends on the cost of computing the FFT of the two inputs, and the cost of computing the inverse FFT of the result of the multiplication, in order to transform the result back to the time domain. While these transformations are too costly to be worth for a single convolution, they may provide an improvement if their cost is amortized by several convolutions. This is the case in convolutional layers where we have to compute the convolution of a batch of inputs with a set of filters, and the FFT of each input and each filter may be reused for all the convolutions where these participate. Thus, the potential improvement of frequency domain transformation increases with the number of inputs and filters.

### 3) WINOGRAD

Another convolution implementation is based on Winograd's minimal filtering algorithms [19]. It applies the transformations described in [31] for the computation of FIR filter outputs, using the minimum number of multiplications. The reduction of multiplications is obtained by an increase in the number of required additions, which are usually less costly to compute. Since the number of additions increases quadratically with the size of the input, the Winograd transforms are better suited for small inputs, because as the size increases, the quadratic factor overcomes the advantage obtained by reducing the number of multiplications. To compute the 3 × 3 convolution of larger inputs, the input is divided in several tiles of 4 × 4 elements, which have to overlap by 2 elements (other filter sizes require different tile sizes). After computing the minimal convolution of each tile, the partial results are summed to obtain the final output.

## III. PERFORMANCE ANALYSIS

In this section we describe the performance analysis we conducted in order to evaluate the presented algorithms on a representative set of convolutional layer configurations. Our main motivation is to detect which features/algorithms are more profitable in terms of computing time and memory occupancy, depending on the parameters of the convolutions.

To analyze the performance of the convolution algorithms we extract the convolutional layer configurations present in five well-known CNNS: AlexNet [17], VGG19 [28], GoogleNet [29], Resnet50 [13], and SqueezeNet [15]. From these configurations we extract the input size, filter size, number of filters, and the input and filter depth. For the batch size we explore several common values (1, 8, 16, 32, 64, 128, 256), since this is usually tuned by the end user of the CNN. The final set includes 602 configurations with the most widely used filter sizes: 1 × 1, 3 × 3, and 5 × 5. We have also leveraged the most common input sizes and number of filters. The former range from 7 × 7 to 224 × 224, while the number of filters ranges from 16 to 2,048. The depth of both filters and inputs is similar to the number of filters, using 1,024 as a maximum.

All the convolution configurations are evaluated with single-precision (32-bit) and half-precision (16-bit) floating point values. For the latter type, we also include results enabling the Tensor Cores (TCs) of our NVIDIA V100 GPU, which are arithmetic units specialized for half-precision matrix multiplication and are available in NVIDIA GPUs since the Volta generation. Note that only some convolution algorithm variants from cuDNN support half-precision and only a subset of these may use the TCs.

We evaluate the implementations available in the NVIDIA CUDA Deep Neural Network library (cuDNN), the reference library for DNN operations on NVIDIA GPUs.[1] In this library, there are several variants of each of the most popular algorithms above described (GEMM, FFT, and Winograd). For instance, we find 3 different variants of the GEMM algorithm: the basic GEMM, where the transformed inputs are stored in a temporary matrix, and 2 other variants, GEMM-impl and GEMM-impl-precomp, where the transformation is performed on the fly by the kernel that computes the GEMM. The difference among these two variants is that GEMM-impl-precomp precomputes some indices needed during the implicit transformation of the inputs. For FFT and Winograd, cuDNN provides the basic implementations plus one variant for each of these: a tiled version of FFT, where the inputs are split into smaller tiles, and a fused version for Winograd, where the transformations of inputs, filters, and outputs are included in the kernel that computes the multiplication.

The convolution algorithms in general and also in cuDNN feature some parameter limitations (different for each algorithm), which render them unavailable for certain convolution configurations. These also require an additional buffer in GPU memory to store intermediate results, with varying size depending on the algorithm and the convolution parameters. We limit the temporary allocation size to 1 GB. This only affects a small number of algorithm/configuration cases ($< 4\%$), most of which are irrelevant because of attaining poor performance compared to other algorithms for the same convolution configuration, even if more space is available.

The test platform used for this work is an IBM POWER9 server running Red Hat Enterprise Linux Server 7.4, equipped with a NVIDIA Tesla V100-SXM2 GPU. The GPU software stack is CUDA 9.1 and cuDNN 7.1.

The plots in this section show the execution time of the configurations normalized to the fastest algorithm for each configuration. Thus, the fastest algorithm is found in the horizontal line $y = 1$, while the rest of lines show normalized time with respect to the fastest algorithm (e.g. a point at $y = 2$ means that the algorithm corresponding to that point runs in twice the time of the fastest algorithm).

The execution time reported for each experiment (i.e. a combination of a given convolution configuration and a given algorithm) is the mean of several repeated executions. For experiments with 32-bit FP values, we use a sample of 10 executions for each experiment. For the experiments with 16-bit FP values, including the variants that use Tensor Cores, we use a sample of 15 executions for each experiment because their execution time is shorter, which makes them more sensitive to execution time variations introduced by the system (e.g. operating system background tasks, etc.). Comparing the relative standard deviation (RSD) of both types of experiments, for 32-bit FP values, the mean RSD is 1.7%, while the mean RSD for the 16-bit FP experiments is 4.5%. Both percentages are quite small, showing that our execution time measurements have a low dispersion. The repeated executions were performed as batches with all the experiments. This way, repetitions of a given experiment are separated in time instead of being consecutive, which avoids temporary system interferences from affecting all the measures of a given experiment. In the plots, the dispersion of the execution times measured for each experiment is shown as a shaded area around each line. The bounds of these areas are the sample mean ± the sample standard deviation of each experiment.

### A. PERFORMANCE ANALYSIS WITH 32-BIT FP VALUES

We have analyzed a total of 602 convolution configurations (86 different configurations extracted from the referenced CNNs, each explored with 7 different common batch sizes), tested with all convolution algorithms available in cuDNN.

After collecting the results from all the tests, the first conclusion is that the parameter which influences the most on performance is the filter size. While for $1 \times 1$ filters the GEMM-based algorithms present high performance, the Winograd variants are faster when dealing with $3 \times 3$ filters. Although Winograd is still a fast algorithm when $5 \times 5$ filters are used, there are other algorithms which are able to outperform it when the batch size is sufficiently large.

In the rest of this section we analyze deeper those algorithms presenting high performance regarding the filter size. We deepen further on relevant configurations by means of the CUDA profiling tools [25], comparing those algorithms which present the highest performance, in order to identify the features of algorithms and configurations which are more influential on performance. In the rest of the paper we use the following format to represent the set of configurations tested: [inputs X&Y size]-[batch size]-[filters X&Y size]-[number of filters]-[depth]. For example, 7-256-1-48-832 refers to a configuration with 256 $7 \times 7$ inputs and 48 $1 \times 1$ filters, all featuring a depth of 832.

#### 1) CONFIGURATIONS WITH $1 \times 1$ FILTERS

Figure 3 shows the execution time of the configurations with $1 \times 1$ filters. For the 315 configurations with $1 \times 1$ filters, the two implicit variants of GEMM prove to be the fastest algorithms, especially GEMM-impl-precomp, which is the fastest 72.4% of the cases. The explicit variant of GEMM is close to the fastest in very few cases ($< 5$) and is usually over 20% slower than the fastest algorithm.

Starting from the left in Figure 3, we observe that the best algorithm for most of the configurations with batch size

[1] cuDNN, https://developer.nvidia.com/cudnn

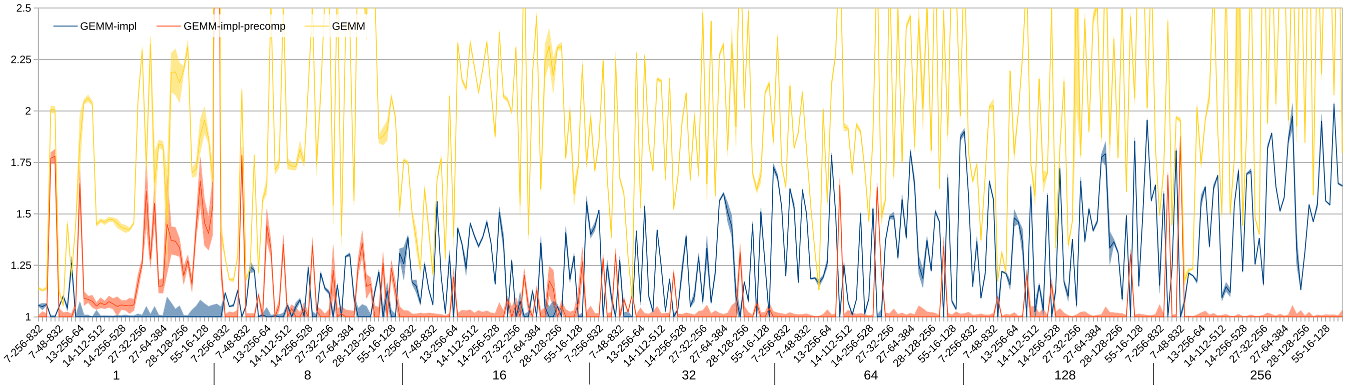

**FIGURE 3.** Normalized execution time relative to the fastest algorithm for configurations with 1 × 1 filters. X axis intervals indicate the batch size; labels indicate [input X&Y size]-[number of filters]-[depth]. Shaded areas show the ± sample standard deviation range for each data point.

**TABLE 2.** Execution time (microseconds) of each relevant algorithm for the selected representative configurations with 1 × 1 filters.

| Algorithm | GPU Kernel | A | B | C |
|---|---|---|---|---|
| **GEMM impl.** | impl_convolve_sgemm<1024>[a] | 78.35 | 80.96 | |
| | impl_convolve_sgemm<128>[a] | | | 426.18 |
| | **Total** | **78.35** | **80.96** | 426.18 |
| **GEMM impl. precomp.** | computeOffsetsKernel | 2.40 | 2.46 | 2.43 |
| | volta_scudnn_128x64_relu_inte[a] | 107.97 | | 213.80 |
| | volta_scudnn_128x32_relu_inte[a] | | 128.48 | |
| | **Total** | 110.37 | 130.95 | **216.23** |

**A**: 7-1-1-192-832, **B**: 14-64-1-32-528, **C**: 27-64-1-64-384.

[a] Some kernels present different versions depending on the configuration.

equal to 1 is the GEMM-impl variant. When leveraging batch sizes of 8 and 16, it is difficult to identify a clear winner. For larger batch sizes the fastest algorithm is in most cases the GEMM-impl-precomp variant. The only exceptions are a few configurations with large depth and small number of filters, such as 14-64-1-32-528 and 7-256-1-48-832, where GEMM-impl is faster than GEMM-impl-precomp.

The profiling results of the two implicit GEMM variants for three representative configurations are presented in Table 2. cuDNN implements GEMM-impl with a single CUDA kernel and GEMM-impl-precomp with two kernels: one for the precomputation of the indices and the other for the implicit convolution. For configuration **A** (see Table 2), an example configuration with batch size of 1, GEMM-impl is the fastest variant. Due to the small batch and input size, this configuration features a low computation load. Since the second kernel in GEMM-impl-precomp has to load the precomputed indices from the GPU memory, the overhead of the memory accesses is too high, rendering the optimization of using precomputed indices slower than the basic implicit variant. A similar issue happens with configuration **B** (batch size of 64 and 14 × 14 inputs), which is one of the rare configurations where GEMM-impl is faster than GEMM-impl-precomp for large batch sizes. Last, configuration **C** represents the common case of GEMM-impl-precomp being the highest performing algorithm. An input size of 27 × 27, added to the batch size of 64, increases the computation load to a point where the optimization implemented by GEMM-impl-precomp is beneficial to reduce the total execution time to almost 50% of the time spent by GEMM-impl ($216.23\mu s$ vs. $426.18\mu s$).

In terms of memory usage, GEMM-impl does not need any additional storage, while GEMM-impl-precomp requires a few KBs (up to 18) to store the precomputed indices. This amount is insignificant on current NVIDIA discrete GPUs and may only affect the algorithm selection in very constrained embedded systems.

The two FFT variants are not competitive in this case, since the computational load of convolutions using such small filters is too low to obtain any advantage from performing the convolution in the frequency domain (i.e. the cost of FFT and inverse FFT overcome the speedup from the multiplication).

### 2) CONFIGURATIONS WITH 3 × 3 FILTERS

The normalized execution time of configurations involving 3 × 3 filters is shown in Figure 4. The Winograd variants are the fastest algorithms in 74.6% of the configurations, especially Winograd non-fused, which is the fastest algorithm in 58.9% of these. GEMM-impl-precomp is still competitive in some configurations, while FFT is the fastest algorithm in some configurations with large batch sizes. The general trend is similar to the 1 × 1 filters, but in this case with the two Winograd variants instead of the implicit GEMMs. For batch size 1 Winograd is the dominating algorithm, followed by Winograd non-fused and GEMM-impl-precomp, which are about 10%- 50% slower in most cases. There is no clear best option for batch size 8, where the previous 3 variants are each the fastest in several cases, and the second and third fastest options are usually less than 20% slower. For larger batch sizes, Winograd non-fused is clearly the fastest in most cases, where the rest of algorithms are about 50% slower. Only in some configurations which make use of a large input size, the GEMM-impl-precomp is faster than Winograd non-fused.

To extract more details we profiled three characteristic configurations: configuration **A**, with batch size of 1, where Winograd is the fastest algorithm; configuration **B**, with batch size of 16, where Winograd non-fused presents the

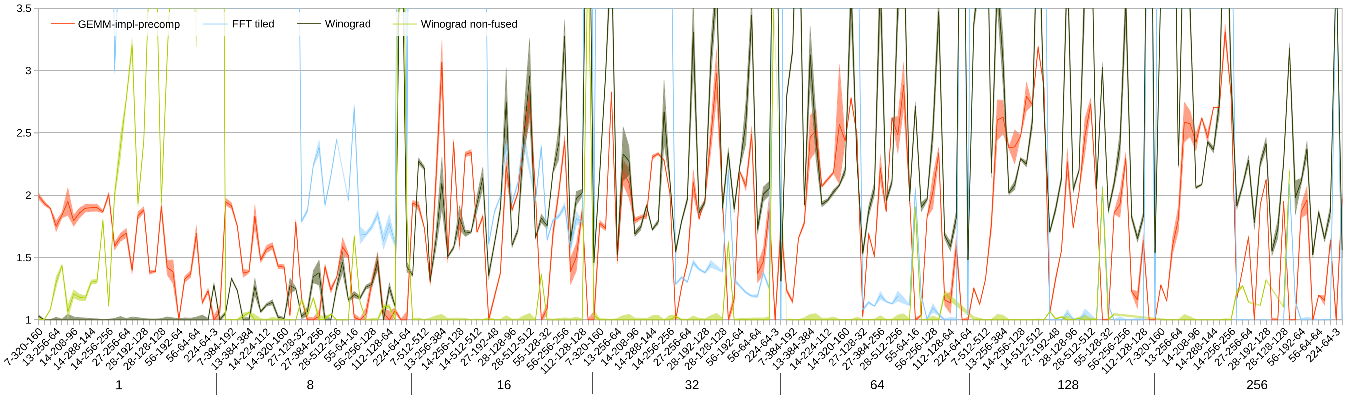

**FIGURE 4.** Normalized execution time relative to the fastest algorithm for configurations with 3 × 3 filters. X axis intervals indicate the batch size; labels indicate [input X&Y size]-[number of filters]-[depth]. Shaded areas show the ± sample standard deviation range for each data point.

**TABLE 3.** Execution time (microseconds) of each relevant algorithm for the selected representative configurations with 3 × 3 filters.

| Algorithm | GPU Kernel | A | B | C |
|---|---|---|---|---|
| **Winograd** | generateWinogradTilesKernel | 6.94 | 5.25 | 7.58 |
| | winograd3x3Kernel<1,4,8>[a] | 71.23 | 2478.00 | |
| | winograd3x3Kernel<2,2,8>[a] | | | 863.22 |
| | ***Total*** | ***78.18*** | *2483.25* | *870.80* |
| **Winograd non-fused** | winogradForwardData4x4 | 9.50 | 53.73 | 51.04 |
| | winogradForwardFilter4x4 | 9.15 | 4.83 | 9.12 |
| | (delay) | 52.00 | 46.00 | 31.00 |
| | volta_sgemm_128x64_nn[a] | 62.34 | | 233.93 |
| | volta_sgemm_128x128_nn[a] | | 741.43 | |
| | winogradForwardOutput4x4 | 10.02 | 1059.42 | 82.27 |
| | ***Total*** | *143.01* | *1905.41* | ***407.36*** |
| **GEMM impl. precomp.** | computeOffsetsKernel | 2.27 | 2.78 | 2.40 |
| | volta_scudnn_128x64_relu_inte | 142.00 | 435.05 | 923.51 |
| | ***Total*** | *144.27* | ***437.83*** | *925.91* |

**A**: 14-1-3-256-128, **B**: 224-16-3-64-3, **C**: 14-128-3-256-128.

[a] Some kernels present different versions depending on the configuration.

best performance; and configuration **C**, where GEMM-impl-precomp outperforms both Winograd variants. Table 3 contains the execution time of these configurations along with a breakdown of the execution time of each CUDA kernel involved in the three algorithms.

The implementation of both Winograd variants is composed of several kernels. The fused Winograd features an initial kernel that precomputes the input tiles and a second kernel that implements the transformations and the multiplication. In Winograd non-fused, all steps of the algorithm are performed by a separate kernel call; the initial two kernels are used to transform the inputs and filters, and after this is performed, the convolutions compute first the multiplication and second the transformations to finally obtain the outputs. The profiler reveals a noticeable delay before the multiplication kernel is executed, caused by the overhead of preparing the kernel arguments and launching the kernel in the GPU. This delay is significant for configuration **A** (batch size 1), where it increases the total execution time by more than 50%, and partly explains why the Winograd variant is faster than the Winograd non-fused for small batch sizes. The other reason is that the specialized multiplication kernel is not fast enough to overcome the cost of the separate transformations. However, in configuration **C** (batch size of 128), the multiply kernel is much faster than the fused Winograd (233.93$\mu$s compared to 863.22$\mu$s), which renders this variant more than 2 times faster. Configuration **B** represents one of the configurations where GEMM-impl-precomp is proven to be the fastest. A large input size of 224 × 224 changes the multiplication kernel of Winograd non-fused to a version able to handle larger products, which is much less efficient than the GEMM-impl-precomp kernel. It also increases the output size, which poses a high impact in the execution time of the output transformation kernel.

There is a significant difference in the amount of memory needed by the three fastest algorithms. Winograd non-fused requires an average of 214 MB in these configurations, a considerable amount even on current NVIDIA discrete GPUs. Winograd needs an average of 3.7 MB and GEMM-impl-precomp requires 27.6 KB. This difference places us in the typical trade-off of speed versus memory size commonly found in computing. If speed is a must, a smarter memory management will be needed to share temporary storage across layers and reduce its impact in the overall memory consumption; otherwise, leveraging the second or third highest performing algorithms instead of the fastest would save a significant amount of memory and reduce the memory management complexity.

### 3) CONFIGURATIONS WITH 5 × 5 FILTERS

The dominance of Winograd non-fused continues for 5 × 5 filter configurations and batch sizes smaller than 64 (Figure 5). It is the fastest algorithm in 57.1% of the 63 configurations. For batch sizes of 64 and larger, GEMM (explicit variant) and GEMM-impl are able to improve the performance of the Winograd variants only on a few configurations with small input size (7 × 7), although for most of these the FFT variants are those showing highest performance. These results match the expectation of FFT variants for large convolutions (i.e. larger filters, large inputs, and large batch

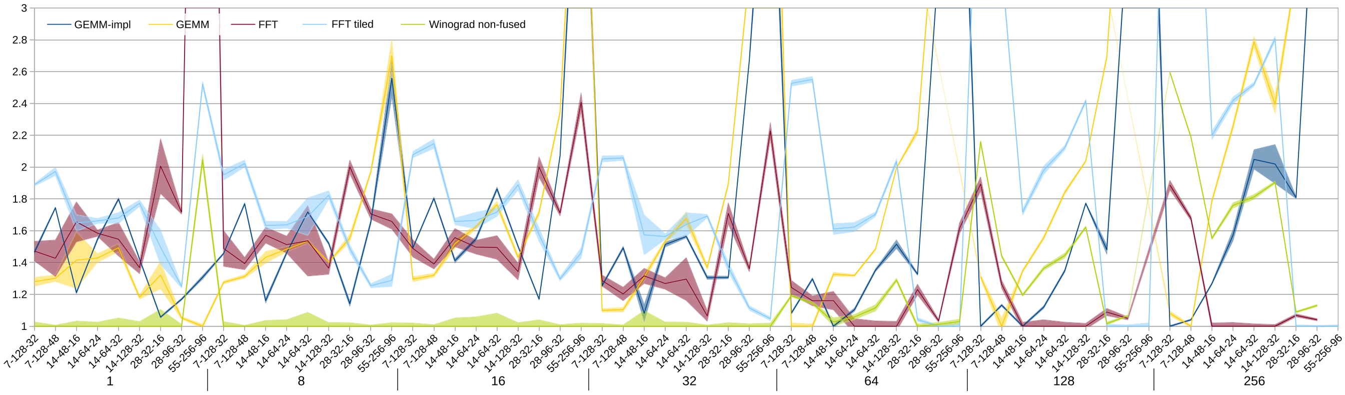

**FIGURE 5.** Normalized execution time relative to the fastest algorithm for configurations with 5 × 5 filters. X axis intervals indicate the batch size; labels indicate [input X&Y size]-[number of filters]-[depth]. Shaded areas show the ± sample standard deviation range for each data point.

**TABLE 4.** Execution time (microseconds) of each relevant algorithm for the selected representative configurations with 5 × 5 filters.

| Algorithm | GPU Kernel | A | B | C |
|---|---|---|---|---|
| **Winograd non-fused** | winogradForwardData9x9_5x5 | 11.23 | 61.41 | 49.51 |
| | winogradForwardFilter9x9_5x5 | 4.80 | 12.48 | 5.06 |
| | (delay) | 31.00 | 19.00 | 21.00 |
| | volta_sgemm_128x64_nn | 21.50 | 40.03 | 62.66 |
| | winogradForwardOutput9x9_5x5 | 13.54 | 105.35 | 74.66 |
| | ***Total*** | ***82.07*** | *238.27* | *212.88* |
| **FFT** | flip_filter | 5.31 | 12.38 | 5.44 |
| | (delay) | 16.00 | 14.00 | 16.00 |
| | fft2d_r2c_16x16 | 10.56 | 17.60 | 11.74 |
| | (delay) | 9.50 | 2.00 | 8.00 |
| | fft2d_r2c_16x16 | 6.75 | 15.68 | 17.31 |
| | (delay) | 23.00 | 13.00 | 11.00 |
| | compute_gemm_pointers | 1.76 | 1.82 | 1.86 |
| | (delay) | 29.00 | 34.00 | 29.00 |
| | volta_cgemm_32x64_tn[a] | 21.63 | | |
| | volta_cgemm_32x32_tn[a] | | 90.21 | 40.87 |
| | fft2d_c2r_16x16 | 10.75 | 42.95 | 28.93 |
| | ***Total*** | *134.27* | *243.64* | ***170.15*** |
| **FFT tiled** | fft2d_r2c_32x32<float,5,true> | 23.00 | 50.63 | 23.17 |
| | (delay) | 7.00 | -16.00[b] | 9.00 |
| | fft2d_r2c_32x32<float,1,true> | 17.44 | 67.62 | 51.00 |
| | (delay) | 18.00 | 0.00 | 0.00 |
| | volta_gcgemm_32x32_nt | 40.00 | 271.62 | 122.95 |
| | fft2d_c2r_32x32 | 16.96 | 122.72 | 77.99 |
| | ***Total*** | *122.40* | *496.59* | *284.10* |
| **GEMM** | im2col4d_kernel | 52.99 | 58.59 | 81.57 |
| | (delay) | 55.00 | 27.00 | 25.00 |
| | explicit_convolve_sgemm | 61.89 | 92.03 | 165.09 |
| | ***Total*** | *169.88* | ***177.63*** | *271.66* |

**A**: 14-16-5-64-32, **B**: 7-128-5-128-48, **C**: 14-128-5-64-32.

[a] Some kernels present different versions depending on the configuration.

[b] The execution of the kernels is partially overlapped.

size), but given that the configurations which use 5 × 5 filters only correspond to 10.5% of all the tested configurations, the impact of FFT is low.

We profiled three representative configurations with 5 × 5 filters, shown in Table 4. As Winograd, the FFT variants are composed of several CUDA kernels. Both variants implement the algorithm steps (transformations and multiplication) in separate kernels. FFT features two auxiliary kernels, *flip_filter* and *compute_gemm_pointers*. It is worth noting the significant launch overhead for short kernels such as *compute_gemm_pointers*, where the delay is an order of magnitude higher than kernel execution time.

Configuration **A** represents the common case for small batch sizes, where Winograd non-fused attains the highest performance. Comparing it to FFT, we see that both multiplication kernels exhibit a similar execution time, while the two auxiliary kernels and their associated setup delays increase FFT's total time, rendering it around 1.6 times slower. FFT tiled and GEMM are in need of costly transformations and multiplications with respect to the other two variants. In **B**, a configuration with small input size and large batch size, the GEMM algorithm is the fastest. The large batch size and the 128 filters pose a high impact on the execution time for the output transformations implemented in Winograd non-fused, which last more than twice the time spent in the multiplication kernel, being only a fraction of it in configuration **A**. A large batch size and a relatively high number of filters also affects the execution time of the multiplication and inverse FFT kernels in both FFT variants. The multiplication kernel in GEMM, however, exposes only a 50% increase in time (being the input transform only a 10.6%). All this and the fact that there is no need to transform the outputs, results in GEMM being the most competitive algorithm for this configuration.

The last configuration shown in Table 4 represents a case where FFT is the fastest alternative. The input size of 14 × 14 harms the performance of the GEMM multiply kernel considerably. Compared to **B**, by halving the number of filters to 64, it improves the time of the multiplication and output transforms of the Winograd non-fused and FFT variants. Compared to configuration **A**, where the only change is found in an increase of batch size from 16 to 128, the multiplication and output transform kernels of FFT only spend twice the time, while for Winograd non-fused and FFT tiled, the time is increased 3 × or even more. This better scalability with respect to the increased batch size explains why FFT overcomes the rest of algorithms for batches of 64 inputs and larger, as shown in Figure 5.

The temporary storage requirement of FFT variants is usually the highest among the 3 algorithms. For the 5 × 5 configurations, FFT needs an average of 82 MB and FFT

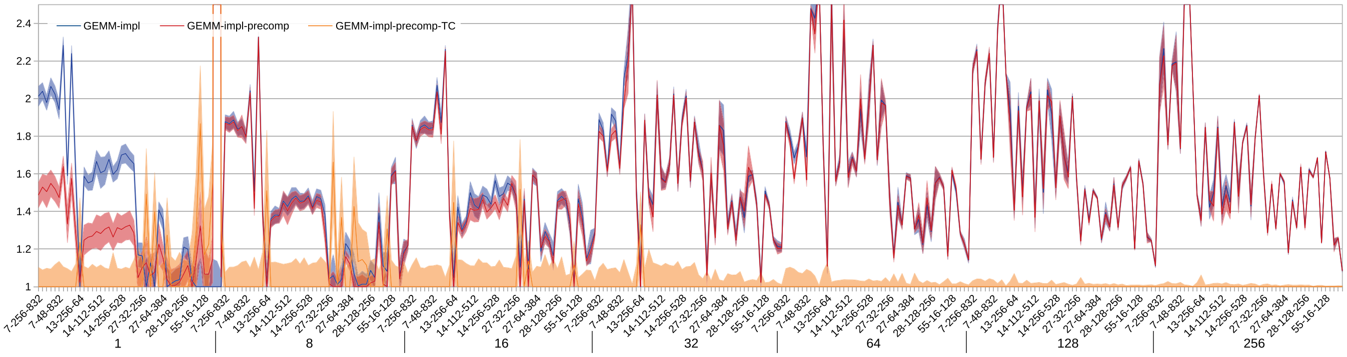

**FIGURE 6.** Normalized execution time relative to the fastest algorithm for configurations with 1 × 1 filters and half-precision data type. X axis intervals indicate the batch size; labels indicate [input X&Y size]-[number of filters]-[depth]. Shaded areas show the ± sample standard deviation range for each data point.

**TABLE 5.** Kernel execution time (microseconds) for the selected representative configurations with 1 × 1 filters and half-precision values.

| Algorithm | GPU Kernel | A | B |
|---|---|---|---|
| **GEMM implicit** | implicit_convolve_hhgemm | 22 | 100 |
| | ***Total*** | *22* | *100* |
| **GEMM impl. Precomp.** | implicit_convolve_hhgemm | 21.6 | 101 |
| | ***Total*** | ***21.6*** | *101* |
| **GEMM impl. Precomp. w/ Tensor Cores** | nchwToNhwcKernel | 4.1 | 5.57 |
| | computeOffsetsKernel | 2.1 | 2 |
| | volta_h884cudnn_256x64_ldg8 | 12.7 | 29.15 |
| | nhwcToNchwKernel | 4.32 | 4.6 |
| | (inter-kernel delays) | 10.2 | 17.9 |
| | ***Total*** | *33.42* | ***59.22*** |

**A**: 13-8-1-256-64, **B**: 14-8-1-256-528

**TABLE 6.** Kernel execution time (microseconds) for the selected representative configurations with 3 × 3 filters and half-precision values.

| Algorithm | GPU Kernel | A | B |
|---|---|---|---|
| **Winograd Non-fused w/ Tensor Cores** | winogradForwardData4x4 | 34.18 | 491 |
| | winogradForwardFilter4x4 | 3.2 | 2.73 |
| | volta_h884gemm_128x256_ldg8_nn | 259.87 | 1047 |
| | winogradForwardOutput4x4 | 140.1 | 547.22 |
| | ***Total*** | *437.35* | ***2087.95*** |
| **GEMM impl. Precomp. w/ Tensor Cores** | nchwToNhwcKernel | 28.1 | 257.54 |
| | computeOffsetsKernel | 1.95 | 2.28 |
| | volta_h884cudnn_256x64..._v1 | 121.12 | 2502 |
| | nhwcToNchwKernel | 72.42 | 281.34 |
| | (inter-kernel delays) | 14.8 | 16 |
| | ***Total*** | ***238.39*** | *3059.16* |

**A**: 55-64-3-64-16, **B**: 56-64-3-256-256

tiled an average of 79.3 MB, while Winograd non-fused requires only 27.3 MB.

### B. PERFORMANCE ANALYSIS WITH 16-BIT FP VALUES

The convolution variants in cuDNN that support 16-bit FP representations are just a subset of the available algorithms in 32-bit FP. Only GEMM-impl, GEMM-impl-precomp, and Winograd non-fused may be used with half-precision data, and only GEMM-impl-precomp and Winograd non-fused may use the TC units. Since the values computed by the TCs may be different due to the change in precision, its use must be explicitly enabled by calling the ''cudnnSetConvolutionMathType'' function.

At a first glance, the performance advantage of the two algorithms with the TCs enabled is clear. Winograd non-fused with TCs is the highest performing variant for most of the configurations with 3 × 3 filters, while GEMM-impl-precomp with TCs is the fastest for most of the remaining configurations.

For 1 × 1 filters and batch sizes of 32 or larger, GEMM-impl-precomp-TC is always the fastest variant, with an average speedup of 1.2 × over the following variant (see Figure 6). For smaller batch sizes, there are a few configurations where the two GEMM variants that do not use the TCs are faster than GEMM-impl-precomp-TC. All of these configurations feature a small depth (64 or smaller). Table 5 contains the kernel execution times for two similar configurations with the aforementioned behavior. Only GEMM-impl-precomp-TC is implemented with multiple kernels. We see that the main kernel needs inputs in NHWC[2] layout instead of NCHW[2], requiring two additional kernels to perform the layout transformation before and after the convolution. GEMM-impl and GEMM-impl-precomp launch the same kernel, suggesting that the indices precomputation is done within the kernel. Comparing the two configurations, we observe that the main kernel of GEMM-impl-precomp-TC scales better than *implicit_convolve_hhgemm* with the 8.25 × increase in the depth of configuration **B**: the former is 2.3 × slower, while the latter attains 4.5 ×. The auxiliary kernels of GEMM-impl-precomp-TC experience similar execution times in both configurations.

As shown in Figure 7, for 3 × 3 filters Winograd-TC is the highest performing variant in most of the evaluated configurations. However, for configurations with large input size (27 or more), GEMM-impl-precomp-TC performs higher than Winograd-TC in some of the cases. To understand which parameters influence this behavior, we analyzed the kernel execution times of two similar configurations where each of the two variants experience a clear performance advantage over their counterpart (Table 6). These configurations differ in the number of filters and depth (both larger in configuration **B**), and after seeing the per-kernel execution times, we observe that both parameters pose a

[2] N: batch; C: depth; H: height; W: width.

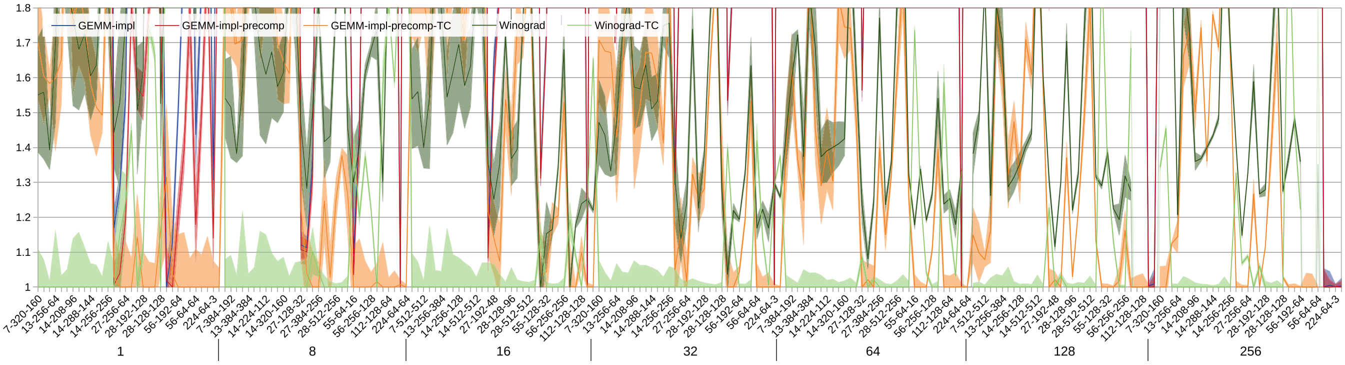

**FIGURE 7.** Normalized execution time relative to the fastest algorithm for configurations with 3 × 3 filters and half-precision data type. X axis intervals indicate the batch size; labels indicate [input X&Y size]-[number of filters]-[depth]. Shaded areas show the ± sample standard deviation range for each data point.

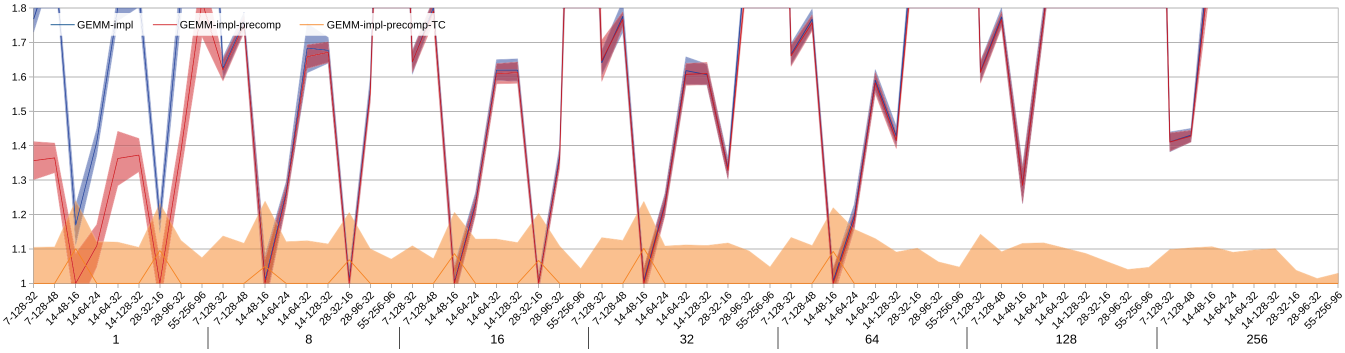

**FIGURE 8.** Normalized execution time relative to the fastest algorithm for configurations with 5 × 5 filters and half-precision data type. X axis intervals indicate the batch size; labels indicate [input X&Y size]-[number of filters]-[depth]. Shaded areas show the ± sample standard deviation range for each data point.

relevant impact, whereas on different kernels. The order of magnitude increase in the execution time of *winogradForwardData4x4* and *nchwToNhwcKernel* is only related to the increase of the depth from 16 to 256, because the transformations performed by these two kernels do not depend on the number of filters. The inverse happens for *winogradForwardOutput4x4* and *nhwcToNchwKernel*, since the depth of the output tensor is actually the number of filters, thus being this parameter the sole responsible of the execution time increase of these two kernels in configuration **B**. For the main kernels, *volta_h884gemm_128x256_ldg8_nn* and *volta_h884cudnn_*_exp_interior_nhwc_tn_v1*, the relative impact of the increase of the number of filters and depth is not as clear, because the computation performed depends on both parameters. We profiled the execution of two additional configurations, changing only one of the two parameters in each, and we observed that the parameter which poses more impact in the execution time is the depth. Only changing the number of filters from 64 to 256 increased the execution time 1.2 × for *volta_h884gemm_128x256_ldg8_nn* and 3.3 × for *volta_h884cudnn_*_exp_interior_nhwc_tn_v1*. Only changing the depth had a larger impact, 2.9 × for *volta_h884gemm_128x256_ldg8_nn* and 6.2 × for *volta_h884cudnn_*_exp_interior_nhwc_tn_v1*. Moreover, the combined effect on the main kernels execution time is what explains the change in the fastest algorithm: for configuration **A**, GEMM-impl-precomp-TC is faster than Winograd-TC; however, the 20 × slowdown in the main kernel caused by the larger parameters of configuration **B** renders it less competitive than Winograd-TC for configurations with more computational load.

For 5 × 5 filters (Figure 8), GEMM-impl-precomp-TC is over 30% faster than the rest of variants, with the exception of eight configurations of depth 16, where GEMM-impl-precomp is the fastest, showing a pattern similar to the configurations with 1 × 1 filters.

### C. PERFORMANCE COMPARISON BETWEEN 32-BIT AND 16-BIT FP VALUES

Comparing the execution time of the best algorithm for 16-bit values versus the best algorithm for 32-bit values for each of the studied configurations, we observe that the execution time for 16-bit FP is faster in 76% of the cases. The speedup of 16-bit FP ranges from 10.5 × in the best case to 0.62 × in the worst case. The filter size is a key parameter again: for configurations with 5 × 5 filters the 16-bit FP version is only faster than 32-bit FP in 46% of the cases. Contrarily, for 3 × 3 filters, it is better in 92% of the tested configurations. For the rest (1 × 1 filters), 16-bit FP obtains higher performance in 70% of the cases, experiencing lower performance than 32-bit FP in configurations with small batch and large input sizes (27 and larger). Since there are few configurations with 5 × 5 filters, overall the performance of 16-bit FP is higher than that of 32-bit FP, as expected given the use of specialized hardware (TCs) in some of the algorithms.

We note that other well-known artifacts derived from numerical representation precision in CNNs besides raw performance—such as memory space or convergence properties—are out of the scope of this study.

## IV. GUIDELINES

To summarize the results of the previous section we provide a collection of recommendations to guide users and programmers in the selection among the different convolution algorithms found in the cuDNN library, depending on the parameters of the convolution.

If 32-bit floating point values are to be used, our recommendations are:

- **1 × 1 filters:** *GEMM-impl* is proven to be efficient for batch size of 1 input, and configurations with large batch sizes, large depth, and a small number of filters. On the other side, *GEMM-impl-precomp* performs better for a batch size greater than 1. This dependency on the batch size makes *GEMM-impl* better suited for the inference stage, once the network is deployed, where inputs are processed one by one or in small batches. However, *GEMM-impl-precomp* is more suited for the training stage, where batches are larger to reduce the training time.
- **3 × 3 filters:** In this case, *Winograd* is able to attain high performance for batch size 1 and *Winograd non-fused* otherwise, except for configurations with large input sizes, where *GEMM-impl-precomp* is able to outperform both Winograd variants. As in the previous case (1 × 1 filters), *Winograd* may be used in the inference stage, and the other two during the network training stage (i.e. larger input batches).
- **5 × 5 filters:** *Winograd non-fused* attains high performance for batch sizes up to 32. Then, *GEMM* shows as the best candidate to deal with small inputs (e.g. 7 × 7), *FFT* for mid-sized inputs (e.g. 14 × 14), and *FFT tiled* for larger input sizes (e.g. 28 × 28 and above).

On the other hand, if the application is able to use reduced precision like 16-bit FP, the options are less, specially given the clear advantage of the two algorithms that exploit the TC arithmetic units. With the exception of some configurations of 3 × 3 filters where GEMM-impl-precomp-TC is faster than Winograd-TC (see analysis in Section III-B), selecting GEMM-impl-precomp-TC for configurations with 1 × 1 and 5 × 5 filters, and Winograd-TC for configurations with 3 × 3 filters is the option yielding highest performance.

## V. CONCLUSIONS

We have presented the main features of the convolution operation in the context of CNNs, reviewing in detail all convolution algorithms available in the NVIDIA cuDNN library, the reference implementation of DNN operations for NVIDIA GPUs and state–of–the art in production runs. We have evaluated these algorithms, in terms of performance and memory occupancy, on one of the latest NVIDIA GPUs, the Volta V100, using 602 different convolution configurations extracted from five well-known CNNs. As part of these evaluations, we have analyzed the performance of each of the algorithms on the aforementioned configurations, with a total of more than 7,000 different test cases. The advantages of each algorithm depending on the parameters of the convolution, including filter size, batch size, and FP precision, were not only identified but also discussed and analyzed in detail. From our findings, we provided a set of guidelines for users and programmers on recommending the algorithm to use depending of the convolution configuration and memory occupancy requirements.

## REFERENCES

[1] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," Mar. 2016, *arXiv:1603.04467*. [Online]. Available: https://arxiv.org/abs/1603.04467

[2] A. Angelova, A. Krizhevsky, V. Vanhoucke, A. Ogale, and D. Ferguson, "Real-time pedestrian detection with deep network cascades," in *Proc. Brit. Mach. Vis. Conf. (BMVC)*, 2015, pp. 1–12.

[3] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, "Mixed precision iterative refinement techniques for the solution of dense linear systems," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 4, pp. 457–466, Nov. 2007.

[4] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "CuDNN: Efficient primitives for deep learning," Oct. 2014, *arXiv:1410.0759*. [Online]. Available: https://arxiv.org/abs/1410.0759

[5] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," Dec. 2014, *arXiv:1412.7024*. [Online]. Available: https://arxiv.org/abs/1412.7024

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. F. Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[7] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, "The design and performance of batched BLAS on modern high-performance computing systems," in *Int. Conf. Comput. Sci. (ICCS)*, pp. 495–504, 2017.

[8] Facebook Open Source. (2017). *Caffe2: A New Lightweight, Modular, and Scalable Deep Learning Framework*. [Online]. Available: http://caffe2.ai

[9] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *J. Mach. Learn. Res.*, vol. 9, pp. 249–256, May 2010.

[10] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, and T. Chen, "Recent advances in convolutional neural networks," *Pattern Recognit.*, vol. 77, pp. 354–377, May 2018.

[11] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. Int. Conf. Mach. Learn.*, Jun. 2015, pp. 1737–1746.

[12] A. Haidar, P. Wu, S. Tomov, and J. Dongarra, "Investigating half precision arithmetic to accelerate dense linear system solvers," in *Proc. 8th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, Nov. 2017, p. 10.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[14] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[15] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," Feb. 2016, *arXiv:1602.07360*. [Online]. Available: https://arxiv.org/abs/1602.07360

[16] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018, *arXiv:1806.08342*. [Online]. Available: https://arxiv.org/abs/1806.08342

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[18] S. K. Kumar, "On weight initialization in deep neural networks," Apr. 2017, *arXiv:1704.08863*. [Online]. Available: https://arxiv.org/abs/1704.08863

[19] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 4013–4021.

[20] H. Li, "Deep learning for natural language processing: Advantages and challenges," *Nat. Sci. Rev.*, vol. 5, no. 1, pp. 24–26, 2017.

[21] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2018, pp. 522–531.

[22] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," Dec. 2013, *arXiv:1312.5851*. [Online]. Available: https://arxiv.org/abs/1312.5851

[23] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," Oct. 2017, *arXiv:1710.03740*. [Online]. Available: https://arxiv.org/abs/1710.03740

[24] D. Mishkin and J. Matas, "All you need is a good init," Nov. 2015, *arXiv:1511.06422*. [Online]. Available: https://arxiv.org/abs/1511.06422

[25] NVIDIA. (2018). *CUDA Toolkit*. https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[26] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *Proc. NIPS-W*, 2017, pp. 1–4.

[27] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," Oct. 2017, *arXiv:1710.05941*. [Online]. Available: https://arxiv.org/abs/1710.05941

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," Sep. 2014, *arXiv:1409.1556*. [Online]. Available: https://arxiv.org/abs/1409.1556

[29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 1–9.

[30] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," Dec. 2014, *arXiv:1412.7580*. [Online]. Available: https://arxiv.org/abs/1412.7580

[31] S. Winograd, *Arithmetic Complexity Of Computations*. Philadelphia, PA, USA: SIAM, 1980.

**MARC JORDÀ** is currently a Research Engineer with the Barcelona Supercomputing Center (BSC), working in the Accelerators and Communications for HPC Team. He is also a GPU Application Porting Specialist and a Technical Lead of the Deep Learning Line of Research of the Team.

**PEDRO VALERO-LARA** was a Researcher on the European Flagship Human Brain Project, NLAFET, and SyeC. His career has led him to work in several R&D Centers and universities, including UCLM, CIEMAT, UCM, BCAM, The University of Manchester, and ICL-University of Tennessee. In all these, He worked with a number of prestigious researchers and participated in multiple projects. He is currently a Researcher with the Barcelona Supercomputing Center (BSC), focused on high-performance, parallel, and scientific computing applied to problems of research and industrial interest.

**ANTONIO J. PEÑA** received the B.S. and M.S. degrees in computer engineering and the M.S. and Ph.D. degrees in advanced computer systems from Universitat Jaume I, Spain. He was with the Argonne National Laboratory as a Postdoctoral Appointee, driving research in heterogeneous memory and accelerator computing within the programming models and runtime systems group. He is currently a Senior Researcher with the Barcelona Supercomputing Center (BSC), where he leads the Accelerators and Communications for HPC Team and the BSC/UPC NVIDIA GPU Center of Excellence. He is also part of the Teaching and Research Staff at the Universitat Politecnica de Catalunya. H was a Juan de la Cierva Fellow. He is currently a Marie Sklodowska-Curie Fellow. He was a recipient of the 2017 IEEE TCHPC Award for Excellence for Early Career Researchers in HPC.