Filling the Performance Gap in Convolution Implementations for NVIDIA GPUs

Barcelona Supercomputing Center
Centro Nacional de Supercomputación
www.bsc.es

Universitat Politècnica de Catalunya
BarcelonaTech

NVIDIA GPU Center of Excellence

Antonio J. Peña, Pedro Valero-Lara,
Marc Jordà

GTC 2019 - San Jose

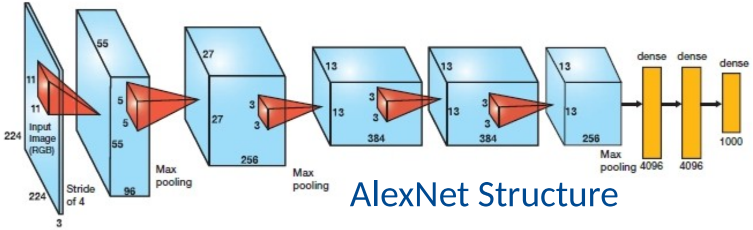
Agenda

- Intro
- Background
  - Convolutional Neural Networks
  - Convolution operation
  - Common characteristics of CNNs
- cuDNN convolution algorithms survey
- Design
  - Data reuse present in conv layers
  - Data layout
  - Algorithm stages
- Performance evaluation
- Conclusions & ongoing work

Introduction

- Interest in neural networks resurged in recent years
  - Deep Neural Networks (DNNs)
- Made possible by
  - Availability of very large annotated datasets (e.g. imageNet)
  - High-throughput heterogeneous systems

- Convolutional Neural Networks (CNNs)
  - High accuracy in image classification benchmarks
  - Several algorithms (Direct, GEMM, FFT, Winograd)



AlexNet Structure

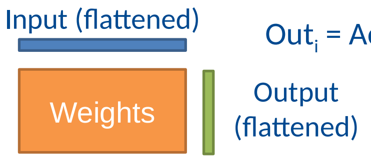
- Our convolution implementation for NVIDIA GPUs
  - Based on direct application of the convolution formula
  - Efficiently exploits in-core memories and global memory accesses

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Convolutional Neural Networks (CNNs)

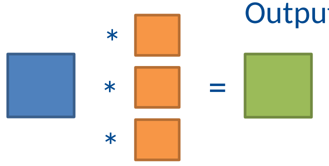
- Inclusion of convolutional layers
- Convolutional layer
  - **Weights** are grouped in **filters**
  - Filters are shared by several output elements
  - Uses convolution operations as part of its computation
- Advantage over fully-connected layers
  - Storage and computational cost does not depend on input or output size
    - Number of filters and its size are a design choice
  - Translation invariance
    - Filters “see” different parts of the input
- Serves as automatic feature extractor
  - Filters are trained to detect relevant patterns

Fully-connected layer

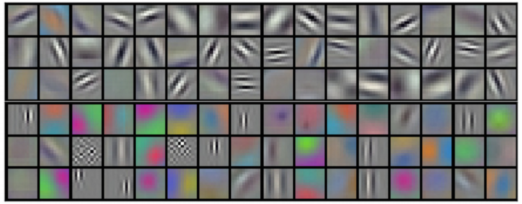


$\text{Out}_i = \text{ActivationFunc}(\text{Sum}_{j=0..\#In}(W_{i,j} \cdot \text{In}_j) + \text{bias})$

Convolutional layer



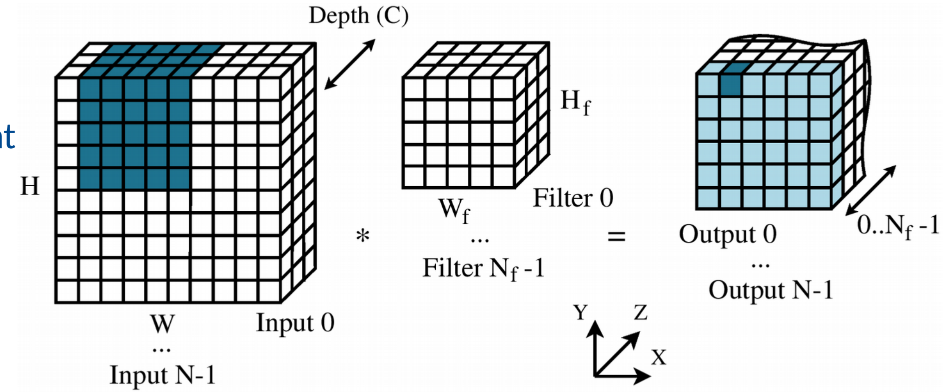
Output = ActivationFunc(ConvolutionOps(Input, Filters) + bias)



Trained filters in the 1st convolutional layer of AlexNet

Convolution Operation

- Output elements are the scalar product of one filter and a subvolume of the input
  - Input and filter depth are equal
  - Different input subvolume for each output element (Dark blue highlight)



- Output planes are the convolution of one input with one of the filters
  - Output depth = number of filters
  - Filter is translated over the X and Y dimensions

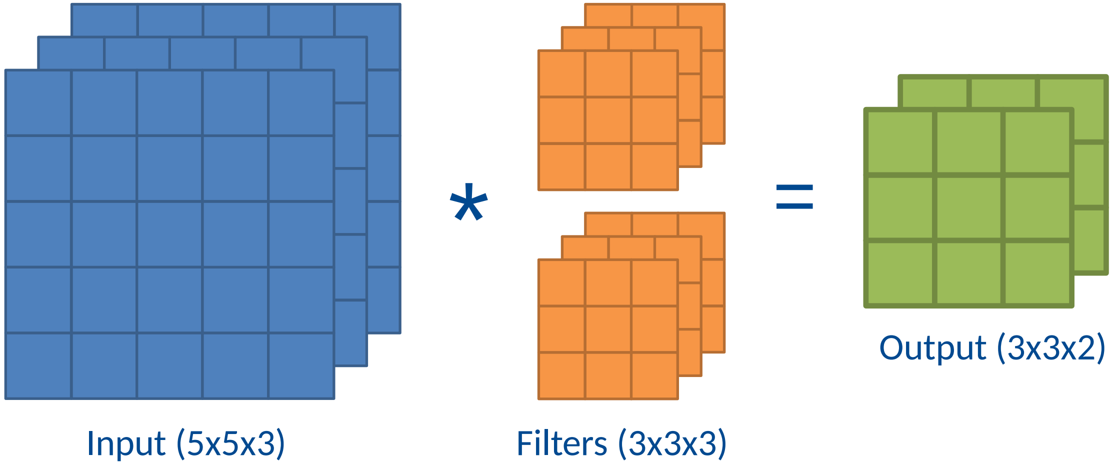
- Convolution parameters
  - # of inputs (aka batch size, N)
  - Input X, Y size (H, W)
  - # of filters (Nf)
  - Filter X, Y size (aka receptive field, Hf, Wf)
  - Depth
  - Stride
  - Padding
  - Dilation

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Convolution Operation - Example

- Example convolution with 1 input and 2 filters
  - 1 input of 5x5x3
  - 2 filters of 3x3x3
  - Stride X and Y = 1

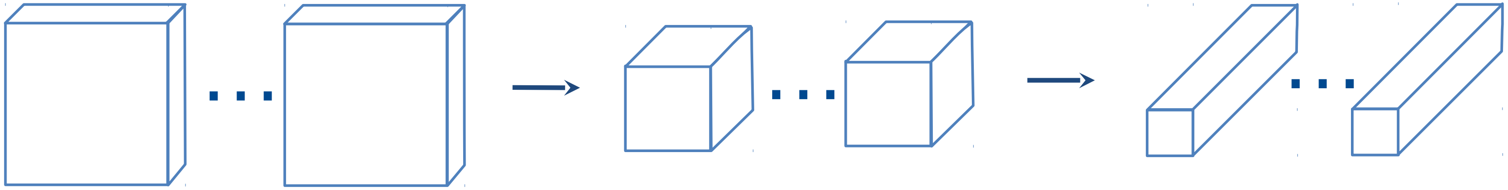
1 output of 3x3x2 (output Z is the number of filters)



Convolution parameter values in CNNs

- Parameters from 5 well-known CNNs
  - AlexNet, GoogleNet, Resnet50, SqueezeNet, VGG19
- Overall structure
  - Initial layers have large input X/Y size, small depth
  - Final layers have small input X/Y size, large depth

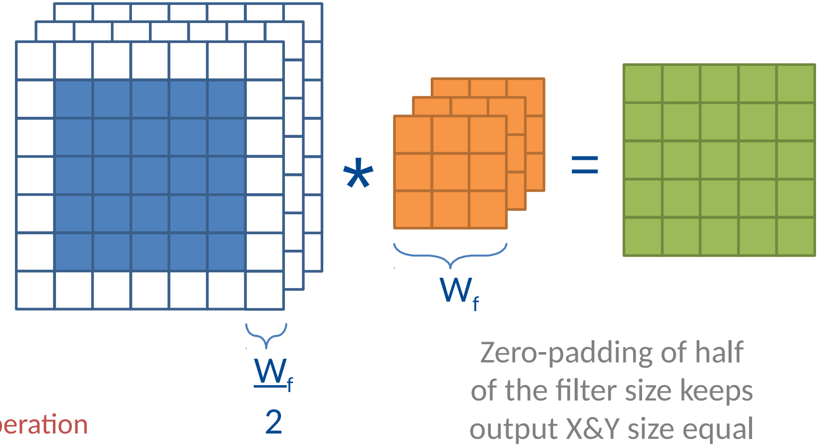
	GoogleNet	SqueezeNet	AlexNet	Resnet50	VGG19
# Distinct convolution configurations	42	21	4	12	9
Network input size	224×224×3	224×224×3	224×224×3	224×224×3	224×224×3
Input size to last convolutional layer	7×7×832	13×13×512	13×13×384	7×7×1024	14×14×512
Convolution filters sizes (% of conv. configs.)	1×1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



Inputs' shape at different layer levels of the CNN

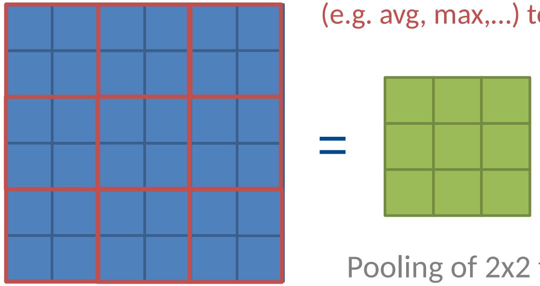
Convolution parameter values in CNNs

- Parameters from 5 well-known CNNs
    - AlexNet, GoogleNet, Resnet50, SqueezeNet, VGG19
- Overall structure
    - Initial layers have large input X/Y size, small depth
    - Final layers have small input X/Y size, large depth
- Padding to maintain X/Y size
    - Input X/Y size reduction is done with pooling layers



Zero-padding of half of the filter size keeps output X&Y size equal to input

Pooling applies a reduction operation (e.g. avg, max,...) to each tile



Pooling of 2x2 tiles halves the X & Y size

Convolution parameter values in CNNs

- Parameters from 5 well-known CNNs
  - AlexNet, GoogleNet, Resnet50, SqueezeNet, VGG19
- Overall structure
  - Initial layers have large input X/Y size, small depth
  - Final layers have small input X/Y size, large depth
- Padding to maintain X/Y size
  - Input X/Y size reduction is done with pooling layers
- Stride = 1 for most convolutions
  - 95% of all convolution configurations
- Filter sizes are small
  - 1x1, 3x3, 5x5, …

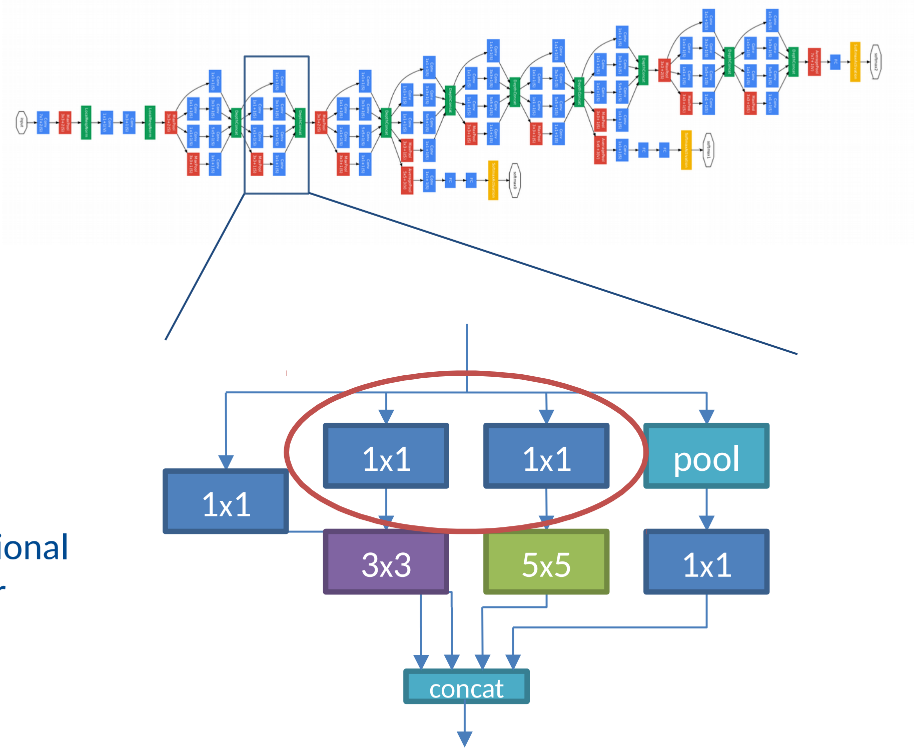
	GoogleNet	SqueezeNet	AlexNet	Resnet50	VGG19
# Distinct convolution configurations	42	21	4	12	9
Network input size	$224\times224\times3$	$224\times224\times3$	$224\times224\times3$	$224\times224\times3$	$224\times224\times3$
Input size to last convolutional layer	$7\times7\times832$	$13\times13\times512$	$13\times13\times384$	$7\times7\times1024$	$14\times14\times512$
Convolution filters sizes (% of conv. configs.)	1×1 (57.2%) 3×3 (23.8%) 5×5 (19%)	1×1 (71.4%) 3×3 (28.6%)	3×3 (75%) 5×5 (25%)	1×1 (66.7%) 3×3 (33.3%)	3×3 (100%)

Characteristics of convolutional layers with stride=1 in the selected CNNs

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Convolution parameter values in CNNs

- Parameters from 5 well-known CNNs
  - AlexNet, GoogleNet, Resnet50, SqueezeNet, VGG19
- Overall structure
  - Initial layers have large input X/Y size, small depth
  - Final layers have small input X/Y size, large depth
- Padding to maintain X/Y size
  - Input X/Y size reduction is done with pooling layers
- Stride = 1 for most convolutions
  - 95% of all convolution configurations
- Filter sizes are small
  - 1x1, 3x3, 5x5, …
- Convolutions with 1x1 filters are a special case
  - Reduce the depth of inputs to reduce the computational cost of the following convolutional layer (with larger filters)



Inception module from GoogleNet

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Convolution algorithms in cuDNN

GEMM-based Algorithm

- Generate two intermediate matrices, multiply them, and reshape the result
  - Filters matrix → flattened filters as rows
  - Inputs matrix → Elements of input subvolumes as columns (im2col in Matlab)
- Pros
  - Can exploit existing high-performance GEMM libs (MKL, cuBLAS, ...)
- Cons
  - Requires extra memory for intermediate matrices
  - Inputs' intermediate matrix is larger than inputs themselves

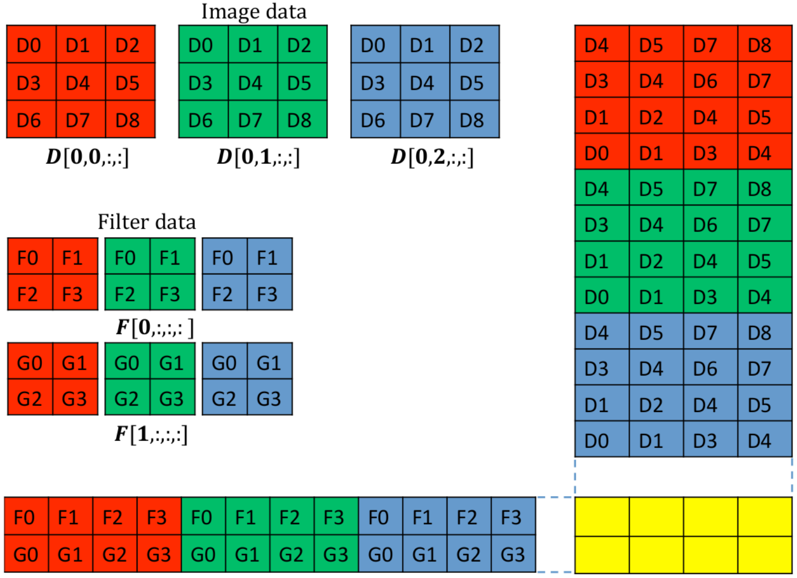


Image from Chetlur et al., *cuDNN: Efficient primitives for deep learning*

Convolution algorithms in cuDNN

Arithmetic strength reduction approaches

- Algorithmic transformation to trade multiplications by additions
  - Additions are faster to execute than multiplications

- Winograd
  - Used in fast FIR filter algorithms in signal processing
  - Inputs: g, d
  - Coefficient matrices: A, B, G

$$Y = A^T \left[[GgG^T] \cdot [B^T dB] \right] A$$

- Fast Fourier Transform
  - FFT + Transformation + Inverse FFT

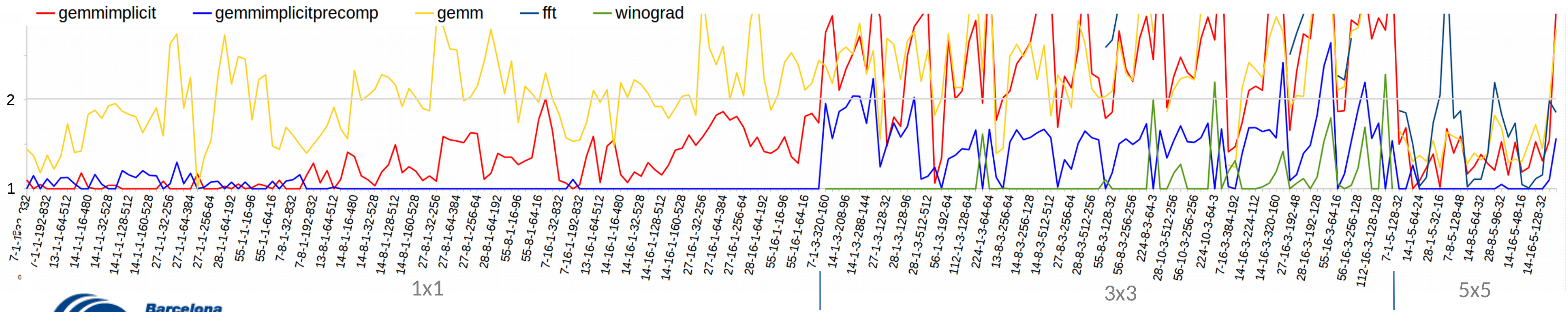
$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}$$

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

cuDNN Convolution Algorithms – Performance survey

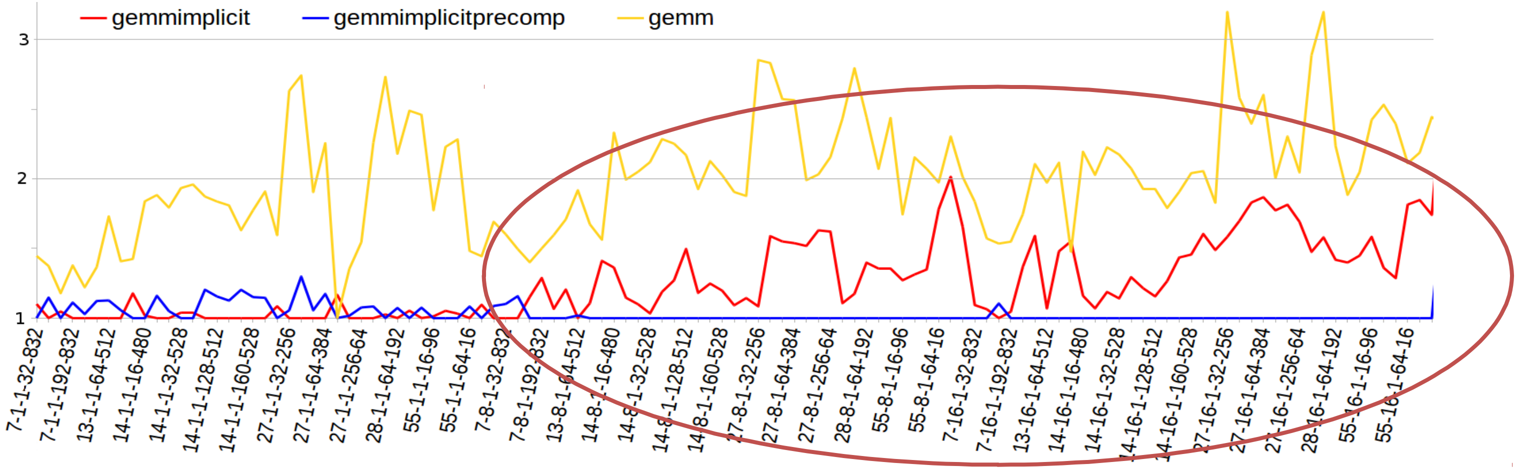
As part of our study, we did a performance survey of cuDNN convolution algorithms

- 3 convolution algorithms
  - GEMM, Winograd, FFT
  - Total of 7 variants: 3 of GEMM (1 explicit input transformation, 2 implicit), 2 of Winograd, and 2 of FFT
- Convolution configurations from well-known CNNs: AlexNet, GoogleNet, Resnet50, SqueezeNet, VGG19
- cuDNN 6 on V100-SXM2 (volta)
- Performance normalized to the best performing algorithm for each convolution configuration
  - Best algorithm is at Y=1
  - X axis labels are <inputXY>-<batch size>-<filter XY>-<#filters>-<depth>



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

cuDNN Convolution Algorithms – Performance survey



Convolution configurations with 1x1 filters (only GEMM variants support this filter size)

- The implicit variants clearly outperform explicit GEMM
  - Explicit GEMM is +1.5x slower for most of the configurations

- GEMM-implicit-precomp is better when the batch size is > 1

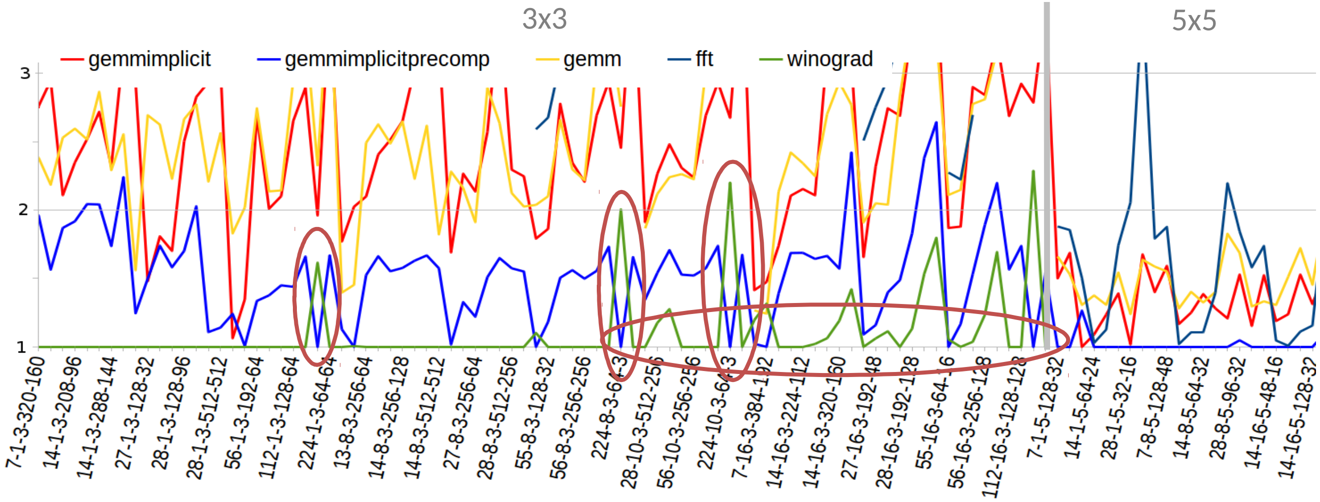
cuDNN Convolution Algorithms – Performance survey

Configurations with 3x3 filters

- Winograd is clearly the best
  - Initially designed for this filter size
- GEMM-impl-precomp outperforms it when depth is small and input X&Y size large

Configurations with 5x5 filters

- GEMM-impl-precomp is the best performing
- FFT gets close in a few cases only
  - Better suited for larger filter sizes



Best is the other winograd variant, not shown to reduce clutter

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

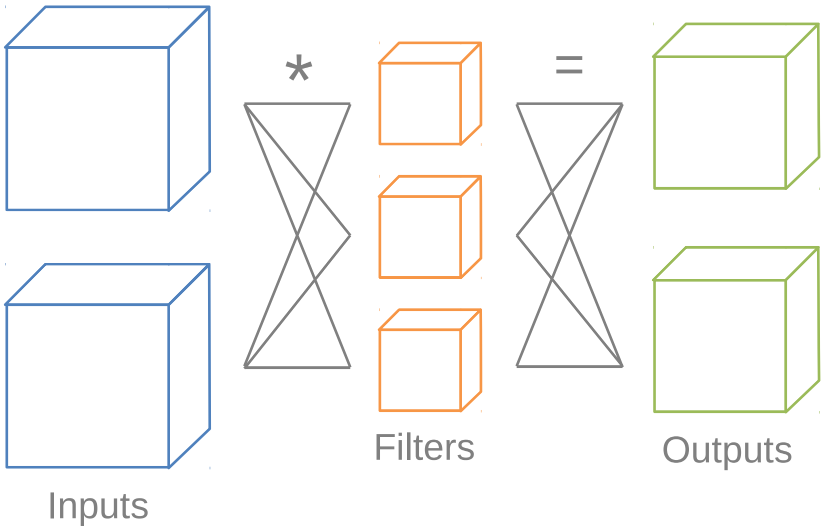
Design

Design – Data reuse

The convolutions of a convolutional layer expose **two levels** of data reuse

At the layer level

- A batch of inputs are convolved with all the layer filters
  - Each filter is used with all the inputs
  - Each input is used with all the filters



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Design – Data reuse

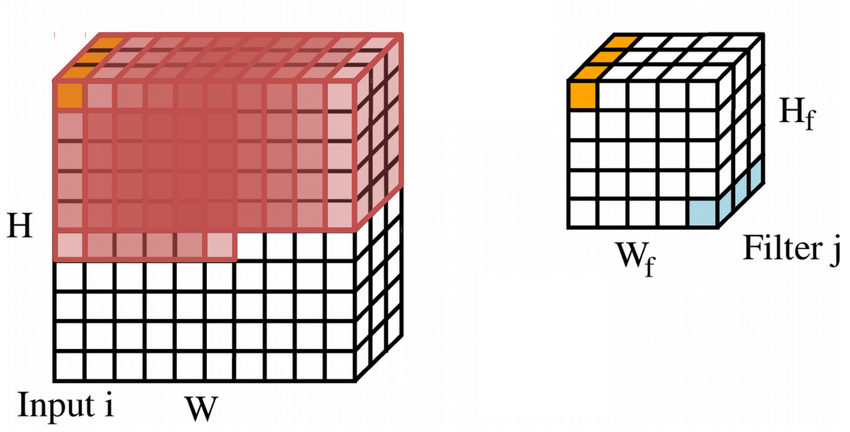
The convolutions of a convolutional layer expose **two levels** of data reuse

At the layer level

- A batch of inputs are convolved with all the layer filters
  - Each filter is used with all the inputs
  - Each input is used with all the filters

At the convolution level

- Input elements reuse
  - Not constant: input z-rows in the center are reused more
- Filter elements reuse
  - Each filter z-row is reused the same amount of times
  - Inputs are usually larger => more reuse of filter z-rows
  - If stride = 1 (common in CNNs), reuse is done by contiguous subvolume



Filter elements reuse: Input elements that reuse two example Z-rows of the filter (in matching colors) in a convolution with stride=1

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Design – Data layout

Flattened representation of the 4-D tensors

- How are data stored in memory
- Denoted as a four letter acronym, one letter per dimension
    - Right-most dim elements are contiguous in memory
- Dimensions
    - N: batch
    - C: depth
    - W: width
    - H: height
- Common layouts in CNNs
    - NCHW
    - NHWC

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Design – Data layout

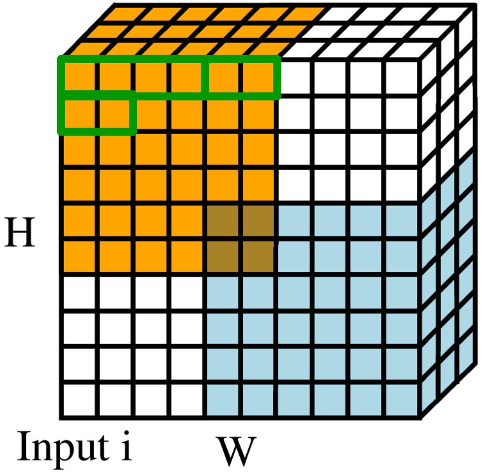
Considering data layout + data reuse + coalescing

If we have

- NCHW layout
- Warps mapped along W dimension
- Stride = 1

We get

- Good coalescing loading inputs
  - Fully-coalesced warps
  - Some warps may have a gap (overhead similar to misaligned accesses)
  - No need for layout transformations before the actual computation
- Threads in a warp reuse filter data
  - Exploit shared mem and shuffle instructions
  - Faster mem access



Example with warp size = 4

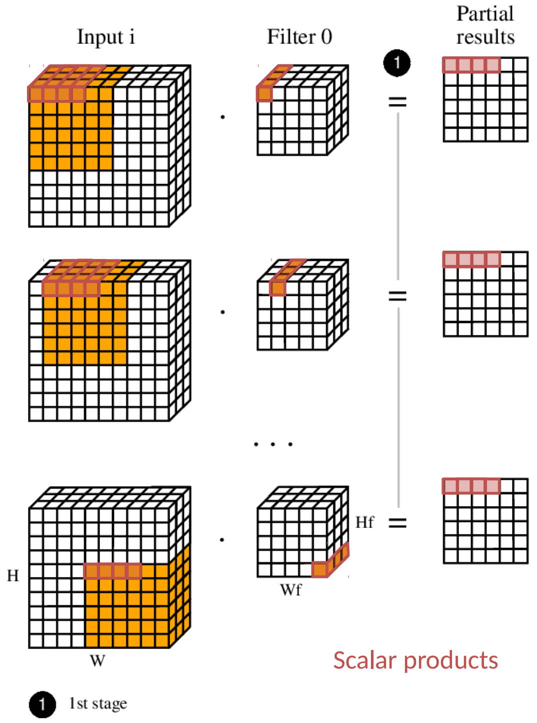
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Design – Algorithm

Computation is split into 2 stages:

1 .- Compute the scalar products between input & filter Z-rows required for the convolutions

- Exploits the reuse of filter elements in shared memory and registers



Design – Algorithm

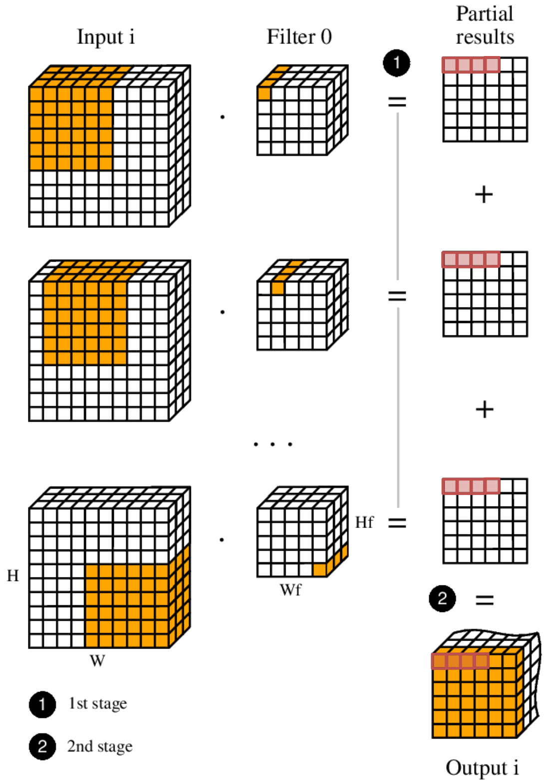
Computation is split into 2 stages:

1 .- Compute the scalar products between input & filter Z-rows required for the convolutions

- Exploits the reuse of filter elements in shared memory and registers

2 .- Add the partial results matrices from the 1st stage to obtain each output X-Y plane.

- Each output element is the sum of one element from each partial results matrix
- Not necessary for convolutions with 1x1 filters
  - Output of 1st stage has to be stored in the correct layout



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Experimental Evaluation

Evaluation dataset

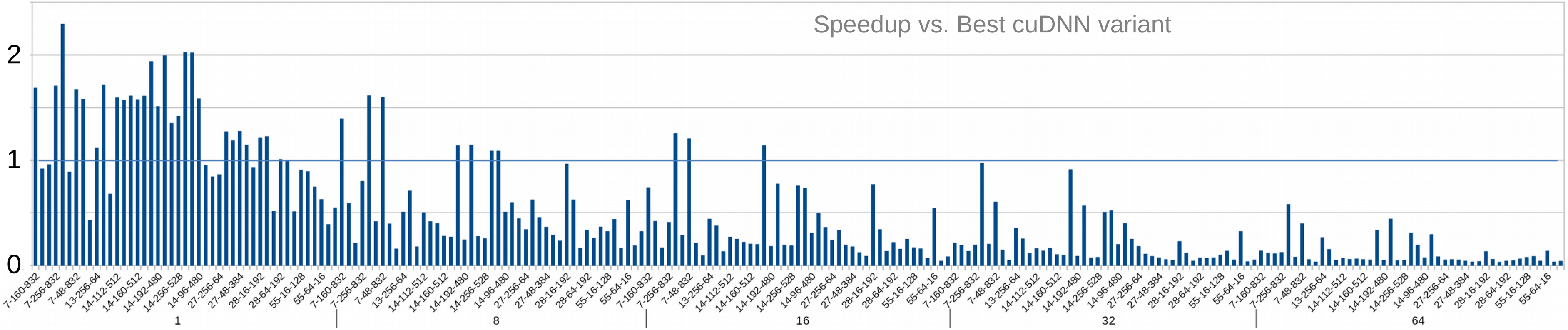
- 602 convolution configurations (X & Y sizes, #filters, depth), from
  - AlexNet, GoogleNet, Resnet50, SqueezeNet, VGG19
- Several input batch sizes: 1, 8, 16, 32, 64, 128, 256
- Total 4000+ configurations
- Single-precision floating point
- Average of 9 executions

Experimental platform

- IBM POWER9 server
- V100-SXM2 (Volta) GPU
- Red Hat Enterprise Linux Server 7.4
- CUDA 9.2
- cuDNN 7.1

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Results



- Overall, our implementation is faster than the best cuDNN variant in 8.31% of the tested configurations
  - Average speedup of 1.46x for these configurations
  - Mainly in smaller batch sizes (up to 16)
  - DL frameworks pick the best algorithm for each convolutional layer
- Insights from performance profiling
  - Our design better exploits thread block-level parallelism for small batch sizes
  - Too many thread blocks negatively impact our performance for large batch sizes
  - Compute & memory access units not fully utilized

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Conclusions & Future work

Our implementation is competitive for certain parameter intervals

- Convolutions with 1x1 filters and small batch sizes
- Speedups of up to 2.29x

Improvements currently in progress

- Support for Tensor Cores for FP16 convolutions
  - Algorithm has to be adapted to the Tensor Cores matrix-matrix multiplication API
- Obtain a better work distribution among thread blocks
  - Work-fusion (e.g. thread coarsening) optimizations
  - Compute units utilization can increase (feedback from profiler)
  - Improve performance for larger batch and filter sizes

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

www.bsc.es

Filling the Performance Gap in Convolution Implementations for NVIDIA GPUs

For further info:
marc.jorda@bsc.es

Antonio J. Peña, Pedro Valero-Lara,
Marc Jordà

GTC 2019 - San Jose