www.bsc.es

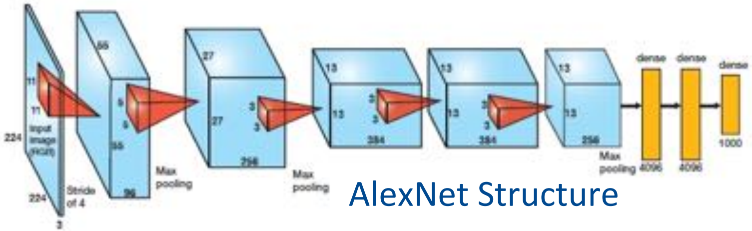
Improved Convolution Implementations on NVIDIA GPUs

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

JLESC April 2019 - Knoxville

Introduction

- Interest in neural networks resurged in recent years
    - Deep Neural Networks (DNNs)
- Convolutional Neural Networks (CNNs)
    - High accuracy in image classification benchmarks
    - Several conv algorithms (Direct, GEMM, FFT, Winograd, …)
- Our convolution implementation for NVIDIA GPUs
    - Based on direct application of the convolution formula
    - Efficiently exploit incore memories and global memory accesses

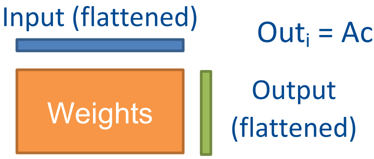


AlexNet Structure

Convolutional Neural Networks (CNNs)

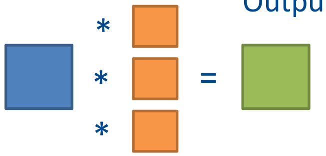
- Inclusion of convolutional layers
- Convolutional layer
  - **Weights** are grouped in **filters**
  - Filters are shared by several output elements
  - Uses convolution operations as part of its computation
- Advantage over fully-connected layers
  - Storage and computational cost does not depend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

Fully-connected layer



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

Convolutional layer

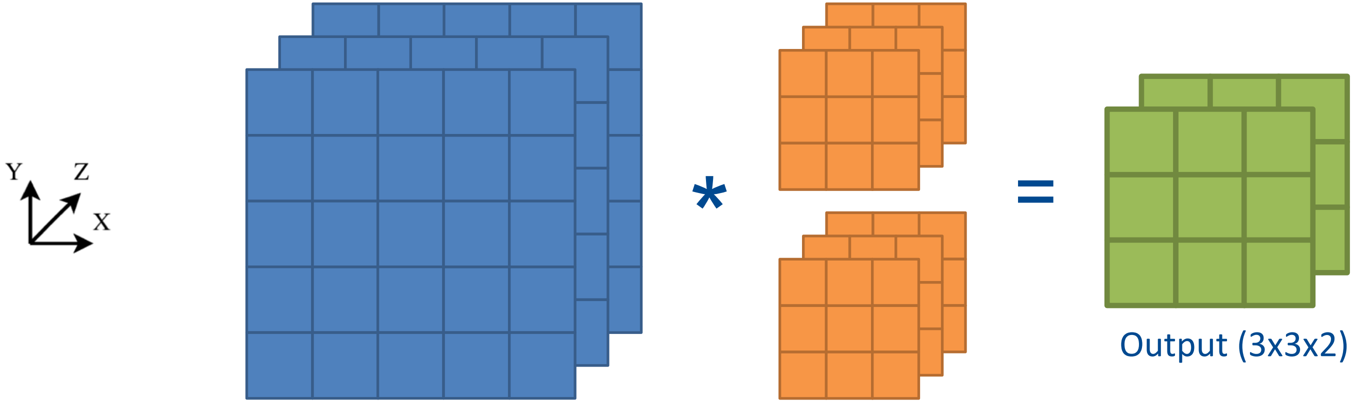


Output = ActivationFunc(ConvolutionOps(Input, Filters) + bias)

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)



Input (5x5x3)

Filters (3x3x3)

Output (3x3x2)

Output elements are the scalar product of one filter and a subvolume of the input

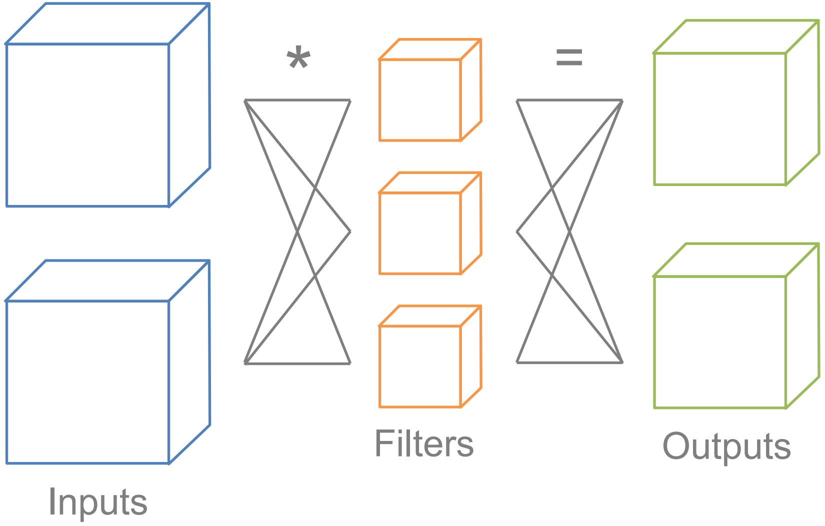
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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



Design – Data reuse

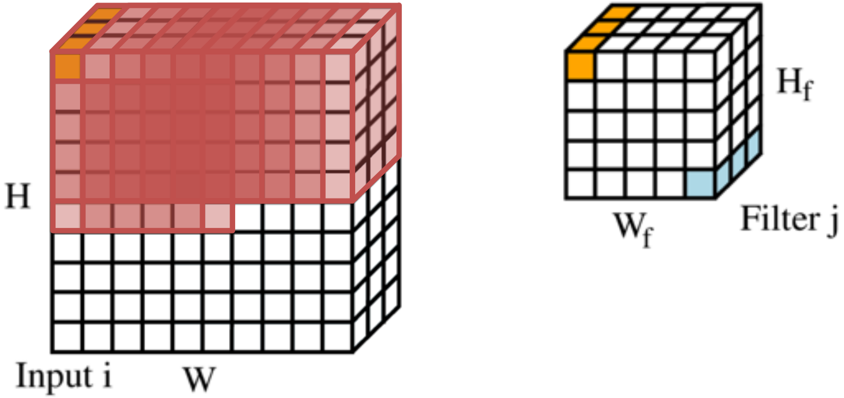
The convolutions of a convolutional layer expose **two levels** of data reuse

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

Design – Data layout

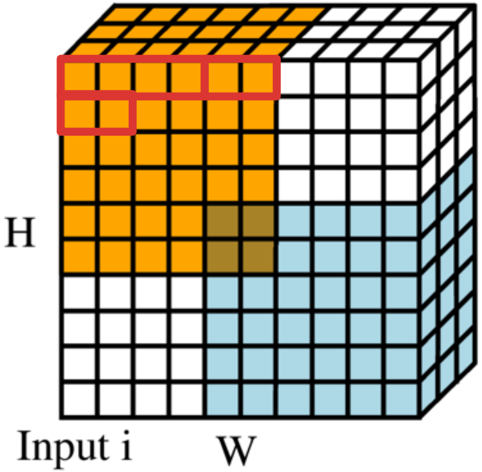
Considering data layout + data reuse + coalescing

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



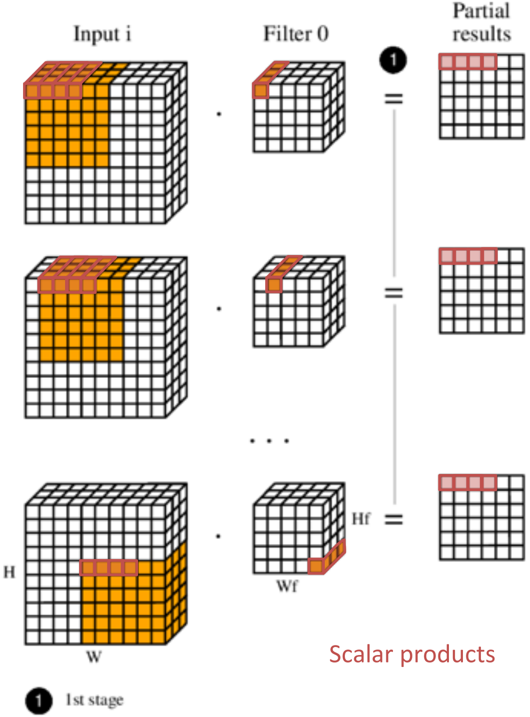
Example with warp size = 4

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



Scalar products

Design – Algorithm

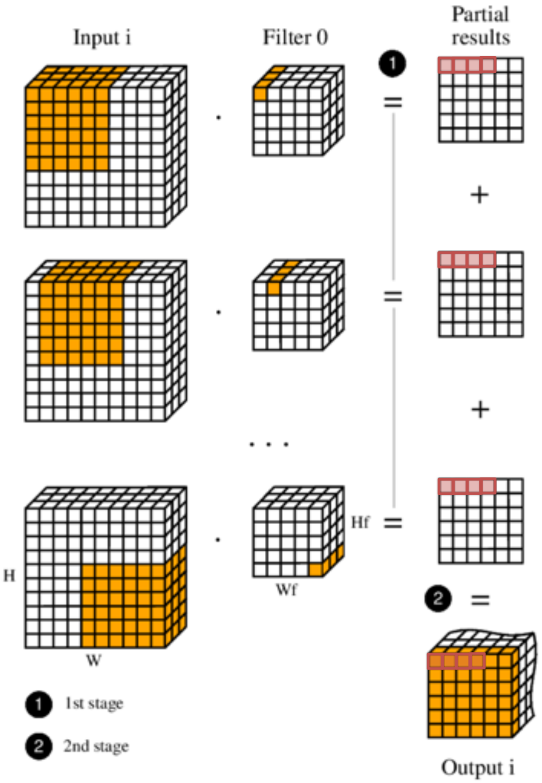
Computation is split into 2 stages:

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



Experimental Evaluation

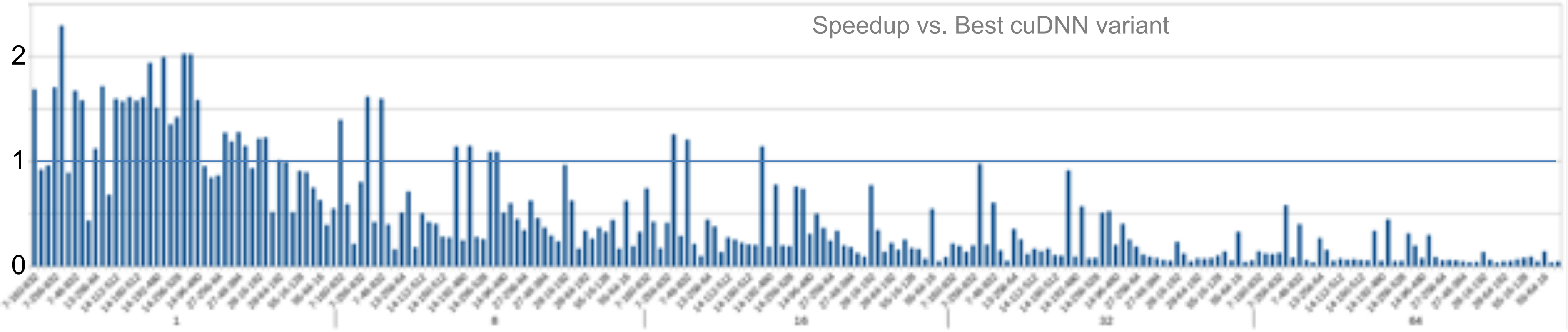
Evaluation dataset

- 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

Results



- Overall, our implementation is faster than the best cuDNN variant in 8.31% of the tested 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 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

Improved Convolution Implementations on NVIDIA GPUs

For further info:
marc.jorda@bsc.es

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

JLESC April 2019 - Knoxville