Filling the Performance Gap in Convolution Implementations for NVIDIA GPUs

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

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

\[
\text{Out}_i = \text{ActivationFunc}(\sum_{j=0}^{\#\text{In}} (W_{ij} \cdot \text{In}_j) + \text{bias})
\]

Convolutional layer

\[
\text{Output} = \text{ActivationFunc}(\text{ConvolutionOps} (\text{Input, Filters}) + \text{bias})
\]

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
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

<table>
<thead>
<tr>
<th># Distinct convolution configurations</th>
<th>GoogleNet</th>
<th>SqueezeNet</th>
<th>AlexNet</th>
<th>Resnet50</th>
<th>VGG19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network input size</td>
<td>224x224x3</td>
<td>224x224x3</td>
<td>224x224x3</td>
<td>224x224x3</td>
<td>224x224x3</td>
</tr>
<tr>
<td>Input size to last convolutional layer</td>
<td>7x7x832</td>
<td>13x13x512</td>
<td>13x13x384</td>
<td>7x7x1024</td>
<td>14x14x512</td>
</tr>
<tr>
<td>Convolution filters sizes (% of conv. configs.)</td>
<td>1x1 (57.2%)</td>
<td>1x1 (71.4%)</td>
<td>3x3 (75%)</td>
<td>1x1 (66.7%)</td>
<td>3x3 (100%)</td>
</tr>
<tr>
<td></td>
<td>3x3 (23.8%)</td>
<td>3x3 (28.6%)</td>
<td>5x5 (25%)</td>
<td>3x3 (33.3%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5x5 (19%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

Pooling of 2x2 tiles halves the X & Y size

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

$W_f \times 2$
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, ...

<table>
<thead>
<tr>
<th></th>
<th>GoogleNet</th>
<th>SqueezeNet</th>
<th>AlexNet</th>
<th>Resnet50</th>
<th>VGG19</th>
</tr>
</thead>
<tbody>
<tr>
<td># Distinct conv layers</td>
<td>42</td>
<td>21</td>
<td>4</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>Network input size</td>
<td>224×224×3</td>
<td>224×224×3</td>
<td>224×224×3</td>
<td>224×224×3</td>
<td>224×224×3</td>
</tr>
<tr>
<td>Input size to last conv</td>
<td>7×7×832</td>
<td>13×13×512</td>
<td>13×13×384</td>
<td>7×7×1024</td>
<td>14×14×512</td>
</tr>
<tr>
<td>Convolution filters sizes (% of conv. configs.)</td>
<td>1×1 (57.2%)</td>
<td>1×1 (71.4%)</td>
<td>3×3 (75%)</td>
<td>1×1 (66.7%)</td>
<td>3×3 (100%)</td>
</tr>
<tr>
<td></td>
<td>3×3 (23.8%)</td>
<td>3×3 (28.6%)</td>
<td>5×5 (25%)</td>
<td>3×3 (33.3%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5×5 (19%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)
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

- Fast Fourier Transform
  - FFT + Transformation + Inverse FFT

\[
Y = A^T \left[ \begin{bmatrix} GgG^T \end{bmatrix} \cdot \begin{bmatrix} B^T dB \end{bmatrix} \right] A
\]

\[
f \ast g = \mathcal{F}^{-1} \left\{ \mathcal{F}\{f\} \cdot \mathcal{F}\{g\} \right\}
\]
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>
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
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
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
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
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
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
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
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
  - 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
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
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