In Memory of Nacho Navarro

PUMPS+AI 2022

Lecture 4: GPU Implementation of CNN

Wen-mei Hwu, Mateo Valero, Toni Pena, Juan Gomez-Luna, Marc Jorda, Leonidas Kosmidis, Bernat Font
Convolutional Neural Networks

- Neural network with Convolutional layers (combined with other types)
- Example: LeNet-5, CNN for Hand-Written Digit Recognition
LeNet-5: CNN for Hand-Written Digit Recognition

6 filters of 5x5 pixels
LeNet-5: CNN for Hand-Written Digit Recognition

The filters are used in groups of 6 (# of input feature maps)

3D convolution of a 14x14x6 pixels input with 16 filters of 5x5x6 pixels each.

96 (6x16) filters of 5x5 px.
Example of the Forward Path of a Convolutional Layer

((N_1 - K_1+1) x (N_2 - K_2+1)) pixels each

(K_1 x K_2) pixels each

(N_1 x N_2) pixels each
Parallelism in a Convolution Layer

All output feature map pixels can be calculated in parallel
- Launch a single kernel to calculate all pixels of all output feature maps
- Each thread block calculates a tile of output feature map pixels
- Essentially a collection of summation of 2D tiled convolutions

See Kirk, Hwu, and El Hajj, PMPP 4th Edition Chapter 16 for such a kernel design
One possible Design of a Basic Kernel

Each block computes a tile of output pixels
- TILE_WIDTH pixels in each dimension

Thread Blocks Grid
- The x dimension maps to the M output feature maps
- The y dimension maps to the tiles in the output feature maps (linearization)
Some Observations

- The amount of parallelism is quite high as long as the total number of pixels across all output feature maps is large.
  - For early stages, there are fewer but larger output feature maps (large Y dimension).
  - For later stages, there are more but smaller output feature maps (large X dimension).
  - This scheme keeps the total number of threads stable across the stages.
Some Observations (2)

Each input tile is loaded from global memory multiple times (by multiple blocks), once for each block that calculates the output tile that requires the input tile

- Not very efficient in global memory bandwidth
- This becomes more of a problem when the output feature maps become small (and increase in number).
Reducing Convolution Layers to Matrix Multiplications

- Convolution layers are the **compute intensive** parts of a CNN.

- GPUs have extremely high-performance implementations of matrix multiplications.

- Tiling techniques for matrix multiplication naturally reuse input features across output feature maps.

- Converting convolutions in a convolution layer to a matrix multiplication helps to keep the level of parallelism stable across CNN layers.
Implementing a Convolutional Layer with Matrix Multiplication

\[
\begin{array}{cccc}
\text{Input Features} & \text{Convolution Filters} & \text{Output Features} \\
X & W & \Delta Y \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
2 & 2 & 0 & 1 \\
\end{array}
\times
\begin{array}{cccc}
1 & 2 & 1 & 1 \\
2 & 0 & 1 & 3 \\
1 & 1 & 0 & 2 \\
1 & 3 & 2 & 2 \\
0 & 2 & 0 & 1 \\
2 & 1 & 1 & 2 \\
0 & 1 & 1 & 1 \\
1 & 2 & 1 & 0 \\
1 & 2 & 0 & 1 \\
2 & 1 & 1 & 3 \\
0 & 1 & 3 & 3 \\
1 & 3 & 3 & 2 \\
\end{array}
= \begin{array}{cccc}
12 & 18 & 13 & 22 \\
10 & 20 & 15 & 22 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{Convolution Filters} & \text{Input Features} \\
W' & X \text{ (unrolled)} \\
\end{array}
\]
A C Function that Generates “Unrolled” X

```c
void unroll(int C, int H, int W, int K, float* X, float* X_unroll) {
    int H_out = H - K + 1;
    int W_out = W - K + 1;
    for(int c = 0; c < C; c++) {
        // beginning row index of the section for channel C input feature
        // map in the unrolled matrix
        w_base = c * (K*K);
        for(int p = 0; p < K; p++)
            for(int q = 0; q < K; q++) {
                int h_unroll = w_base + p*K + q;
                for(h = 0; h < H_out; h++)
                    for(int w = 0; w < W_out; w++) {
                        int w_unroll = h * W_out + w;
                        X_unroll[h_unroll, w_unroll] = X(c, h + p, w + q);
                    }
            }
    }
}
```
Implementing a Convolutional Layer with Matrix Multiplication

```
01 void unroll(int C, int H, int W, int K,
               float* X, float* X_unroll) {
02    int H_out = H - K + 1;
03    int W_out = W - K + 1;
04    for(int c = 0; c < C; c++) {
        // beginning row index of the section for channel C
05        w_base = c * (K*K);
06        for(int p = 0; p < K; p++)
07           for(int q = 0; q < K; q++) {
08              int h_unroll = w_base + p*K + q;
09              for(h = 0; h < H_out; h++)
10                 for(int w = 0; w < W_out; w ++) {
11                    int w_unroll = h * W_out + w;
12                    X_unroll(h_unroll, w_unroll) = X(c, h + p, w + q);
13                }
14            }
15        }
16    }
17 }
```

```
K*K=2*2
```

$H_{\text{out}} = W_{\text{out}} = 2$

$C = 3$
Each product matrix element is an output feature map pixel

This inner product generates element 0 of output feature map 0
Each block calculates **one 2x2 output tile** – 2 elements from each output map

Each block loads **one 2x2 filter tile and one 2x2 input tile** into the shared memory and performs two dot product steps for all its 2x2 output elements

Each input element is **reused 2 times in the shared memory**
Tiled Matrix Multiplication 2x4 Shared-Memory Tiling
(Another Toy Example)

Each block calculates one 2x4 output tile – 4 elements from each output map

Each block loads one 2x2 filter tile and one 2x4 input tile into the shared memory and performs two dot-product steps for all its 2x4 output elements

Each input element is reused 2 (or 4 for filters) times in the shared memory
Each output map element requires its own replicated $K \times K$ input feature map elements

- Not replicated for different output feature maps
- There are $H_{\text{out}} \times W_{\text{out}}$ output feature map elements
- Each requires $K \times K$ replicated input feature map elements
- So, the total number of input elements after replication is $H_{\text{out}} \times W_{\text{out}} \times K \times K$ times for each input feature map
- The total number of elements in each original input feature map is $(H_{\text{out}} + K - 1) \times (W_{\text{out}} + K - 1)$

Expansion ratio = \[
\frac{C \times H_{\text{out}} \times W_{\text{out}} \times K \times K}{C \times (H_{\text{out}} + K - 1) \times (W_{\text{out}} + K - 1)}
\]
Analysis of Efficiency: Total Input Replication (II)

- \( H_{\text{out}} = 2 \)
- \( W_{\text{out}} = 2 \)
- \( K = 2 \)
- There are \( C=3 \) input maps (channels)
- The total number of input elements in the replicated (“unrolled”) input matrix is \( 3 \times 2 \times 2 \times 2 \times 2 \)
- The expansion ratio is \( (3 \times 2 \times 2 \times 2 \times 2)/(3 \times 3 \times 3) = 1.78 \)
Tiled Matrix Multiplication is More Stable in Matrix Sizes

- The filter-bank matrix is an $M \times C \times K \times K$ matrix
  - $M$ is the number of output feature maps
- The expanded input feature map matrix is a $C \times K \times K \times H_{out} \times W_{out}$ matrix
- The sizes of the matrices depend on products of the parameters to the convolution, not the parameters themselves
- For example, while $H_{out} \times W_{out}$ tends to decrease towards later stages of a CNN, $C$ tends to increase at the same time
  - The amount of work for each kernel launch will remain large as we go to the later stages of the CNN
The conversion from convolution to matrix multiplication can be done during execution

- The input feature maps are stored in their original form
- The kernel that implements a convolution layer performs a tiled matrix multiplication on the conceptual unrolled input matrix
- When loading each tile from the “unrolled input matrix”, the kernel extracts the tile elements from the original input feature maps based on a mapping like that used in the C code
- This way, there is no preprocessing cost and the input feature maps in the memory is not expanded due to unrolling
ADVANCED TILING TECHNIQUES FOR MATRIX MULTIPLICATION
Joint Register and Shared-Memory Tiling

- Registers are accessed at extremely high throughput but
  - Private to each thread
  - Register tiling needs thread coarsening

- Shared memory is accessed at lower throughput than registers but still much higher than global memory
  - Visible to all threads in a block
  - Does not need thread coarsening
  - Still needs to be first loaded into registers before used by compute units

- We typically use both for tiling different dimensions of a multidimensional data

- One can use registers even more aggressively with warp programming and tensor cores
Data Reuse in Matrix Multiplication (Revisited)

Reuse of column data in N
Data Reuse in Matrix Multiplication (Revisited)

Reuse of row data in M
Data Reuse in Matrix Multiplication: Example 4x4 Output Tile

Only four elements of M and four elements of N are needed to calculate one step for a 16-element tile of P.
The P output tile does not need to be square.

For a 4x2 tile:
- 4 elements of M and 2 elements of N are needed for each step.
Data Reuse in Matrix Multiplication: Example 4x2 Output Tile (II)

Step 2…
At each step
- For 4x2, only 6 elements need to be loaded for all 8 threads to make progress
- For 4x4, 8 elements for all 16 threads
In the Kernel of the Previous Slide (e.g., 4x4 Output Tile)

- TILE_WIDTH^2 elements of M and TILE_WIDTH^2 element of N are loaded
- Each thread block calculates TILE_WIDTH steps for TILE_WIDTH^2 elements of P
  - TILE_WIDTH is typically at least 16
- According to our analysis, we can use much smaller amount of shared memory by
  - Loading TILE_WIDTH element of M and TILE_WIDTH element of N (input tiles) to calculate 1 step for TILE_WIDTH^2 elements in the output tile.
  - So, why didn’t we do so?
Cost of Loading Smaller Input Tile

If in each iteration of the outer loop
- only a subset of threads load M and N elements (divergence or load imbalance)
- Call __synchthreads()
- All threads calculate one step of the inner product (inner loop degenerates to one iteration)
- Call __synchthreads()
- Go to the next iteration

Even though __synchthreads() is a very efficient function, such intensive use is still going to hurt
Joint Register and Shared Memory Tiling

- Store input M tile and output P tile elements in registers
- Store input N tile elements in shared memory
- Decouple of M and N input tile widths
  - TILE_WIDTH_M, TILE_WIDTH_N
- Key quantities
  - Number of threads = TILE_WIDTH_M
  - Output tile size = TILE_WIDTH_M * TILE_WIDTH_N
  - Reuses for each N element = TILE_WIDTH_M
  - Reuses for each M element = TILE_WIDTH_N
  - Each thread calculates TILE_WIDTH_N P elements

Example:

TILE_WIDTH_M = 4
TILE_WIDTH_N = 2
Have each thread to calculate a horizontal strip of TILE_WIDTH_N P elements

Data loaded from M can be reused TILE_WIDTH_N times through registers
  - Register tiling

Load 1 value from M into r1

r1 can be reused to compute 4 intermediate results for P

\[
\begin{align*}
P[0] & += r1 \times n1; \\
P[1] & += r1 \times n2; \\
P[2] & += r1 \times n3; \\
P[3] & += r1 \times n4;
\end{align*}
\]
Multiple threads collaborate to load TILE_WIDTH*N elements into shared memory.
In One Iteration, Each Thread…

- Accesses one M element from register, accesses TILE_WIDTH_N N elements from shared memory
- Calculates one step for TILE_WIDTH_N P elements
- TILE_WIDTH_N = ~16 in practice (limited by registers needed for P elements)

n1~n4 accessed from shared memory

Intermediate results computed by T1; stored in registers
In One Iteration, Each Block…

- Has TILE_WIDTH_M threads
  - 64 or more in practice

- Loads TILE_WIDTH_M M elements into registers, loads TILE_WIDTH_N N elements into shared memory
  - TILE_WIDTH_N is number of threads folded into one thread in thread coarsening (16 or more in practice)

- However, loading of N will involve only a subset of threads (divergence)
A More Balanced Approach

- In each iteration, all threads in a block collaborate to load a TILE_WIDTH_N * K tile of N into shared memory
  - K is set so that TILE_WIDTH_M = TILE_WIDTH_N * K
  - Every thread loads one N element, no divergence

- Each thread loads K M elements into registers
  - Value of K is limited by the number of registers
  - Each thread needs to use
    - TILE_WIDTH_N registers for output elements
    - K registers for M elements

- Each block calculates K steps for TILE_WIDTH_N * TILE_WIDTH_M P elements
Summary: Joint Register and Shared Memory Tiling

- Each block has TILE_WIDTH_M threads
- Each thread coarsened by TILE_WIDTH_N times
- Each thread loads
  - One N element into the shared memory
  - $K = \frac{TILE\_WIDTH\_M}{TILE\_WIDTH\_N}$ M elements
  - To calculate $K$ steps of TILE_WIDTH_N P elements
For a Toy Example

- Each block has 8 threads
- Each thread coarsened by 4 times
- Each thread loads
  - One N element
  - \(8/4 = 2\) M elements
  - To calculate 2 steps of 4 P elements

\[
\text{TILE_WIDTH}_M = 8
\]

\[
\text{TILE_WIDTH}_N = 4
\]
Each block has 64 threads
Each thread coarsened by 16 times
Each thread loads
- One N element
- $64/16 = 4$ M elements
- To calculate 4 steps of 16 P elements
A Comparative Analysis

**Tiled SGEMM using shared memory for both inputs:**
- Each thread block computes 32x32=1024 results
- Uses 12 KB on-chip memory (register + shared memory)

**Register/Shared-Memory tiled version of SGEMM:**
- Each thread block computes 64x16=1024 results
- Uses only 5 ¼ KB on-chip memory
  - Similar degree of reuse; ~2X more efficient

<table>
<thead>
<tr>
<th>Tiling algorithm</th>
<th># of reuse per data in M</th>
<th># of reuse per data in N</th>
<th># of data computed per block in P</th>
<th>Shared memory usage per block</th>
<th>(M+P) Register usage per TB</th>
<th>Performance on GTX280 in GFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jointly tiled SGEMM</td>
<td>16</td>
<td>64</td>
<td>16x64</td>
<td>(N) 4x16x4 =256Bytes</td>
<td>(64x4+64x16)x4 =5KB</td>
<td>~430</td>
</tr>
<tr>
<td>Shared-Memory Tiled SGEMM</td>
<td>32</td>
<td>32</td>
<td>32x32</td>
<td>(N+M) 32x32x4x2 =8KBytes</td>
<td>32x32x4=4KB</td>
<td>&lt;300</td>
</tr>
</tbody>
</table>
Loading N into shared memory is easily coalesced with the 16x4 tile

Loading M into registers is not coalesced
  – Transpose M for coalescing
CUDNN Library

CUDNN is a library of optimized routines for implementing deep learning primitives.

C-language API that integrates into existing deep learning frameworks (e.g., Caffe, Tensorflow, Theano, Torch).

Same as cuBLAS, CUDNN assumes that input and output data reside in the GPU device memory.
Batched Convolution in CUDNN (I)

Most important primitive in CNN

Two inputs to the convolution:
- D, a four-dimensional $N \times C \times H \times W$ tensor, with input data
- F, a four-dimensional $K \times C \times R \times S$ tensor, with convolutional filters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Number of images in minibatch</td>
</tr>
<tr>
<td>C</td>
<td>Number of input feature maps</td>
</tr>
<tr>
<td>H</td>
<td>Height of input image</td>
</tr>
<tr>
<td>W</td>
<td>Width of input image</td>
</tr>
<tr>
<td>K</td>
<td>Number of output feature maps</td>
</tr>
<tr>
<td>R</td>
<td>Height of filter</td>
</tr>
<tr>
<td>S</td>
<td>Width of filter</td>
</tr>
<tr>
<td>u</td>
<td>Vertical stride</td>
</tr>
<tr>
<td>v</td>
<td>Horizontal stride</td>
</tr>
<tr>
<td>pad_h</td>
<td>Height of zero padding</td>
</tr>
<tr>
<td>pad_w</td>
<td>Width of zero padding</td>
</tr>
</tbody>
</table>
Batched Convolution in CUDNN (II)

The output is also a four-dimensional tensor $O$

- $N \times K \times P \times Q$
- $P = f(H, R, u, \text{pad}_h)$
- $Q = f(W, S, v, \text{pad}_w)$

- Height and width of the output feature maps depend on the input feature map and filter bank height and width, along with padding and striding choices.
- The goal of the striding parameters $(u, v)$ is to reduce the computational load by computing only a subset of the output pixels.
- Padding parameters are for improved memory alignment and/or vectorized execution.