DNN Accelerators and HLS

Qijing (Jenny) Huang
Outline

1. Deep Neural Network (DNN)
2. Design Methodology
3. Accelerator Architecture
4. High-level Synthesis (HLS)
The basic computational unit of the brain is a neuron

- 86B neurons in the brain

Neurons are connected with nearly $10^{14} - 10^{15}$ synapses

- Neurons receive input signal from **dendrites** and produce output signal along **axon**, which interact with the dendrites of other neurons via **synaptic weights**

- **Synaptic weights** – learnable & control influence strength

* Slide from [http://cs231n.github.io/](http://cs231n.github.io/)
Neural Networks

- NNs are usually feed forward computational graphs constructed from one or more layers
- The “Neuron” computes:
  - Integrate - typically linear transform (dot-product of receptive field)
  - Fire - followed by a non-linear “activation” function

* Slide from http://cs231n.github.io/*
Training vs. Inference

Training

Process for a machine to learn by optimizing models (weights) from labeled data.

Typically computed in the cloud

Inference

Using trained models to predict or estimate outcomes from new inputs.

Deployment at the edge

* Slide from https://www.hotchips.org/archives/2010s/hc30/
## Many AI Chips

### In the Cloud (Training + Inference)
- 10s TFLOPs
- 10s MB on-chip memory
- 8 - 32 bit precision
- 700 MHz - 1 GHz
- 10-100s Watts

### At the Edge (Inference)
- 100s-1000s GFLOPs
- 100s KB on-chip memory
- 1 - 16 bit precision
- 50 MHz - 400 MHz
- 1-10s Watts

### In the Edge SoC/SiP (Inference)
- 10s-1000s GFLOPs
- 100s KB on-chip memory
- 1 - 16 bit precision
- 600 MHz - 1 GHz
- 10-100s mWatts

---

* Data adapted from Prof. Kurt Keutzer's talk at DAC 2018
Computer Vision Applications

- Autonomous Vehicles
- Security Camera
- Drones
- Medical Imaging
- Robots
- Mobile Applications
Computer Vision Tasks

- Image Classification
- Object Detection
- Semantic Segmentation
- Super Resolution
- Activity Recognition
Deep Neural Network
Common Operations

- Convolution (Dilated, Transposed, 3D and etc.)
- ReLU
- Pooling (Average, Max)
- Fully-Connected
- Batch Normalization
Activation/Feature Maps

- Input images have three dimensions with RGB channels
- Intermediate data have more channels after performing convolution
- We refer to them as feature maps

Input Image:

One Feature Map:

Channel Dimension

height

width
Weights/Kernels

- weights for full convolution typically have four dimensions:
  - input channels, width, height, output channels
- input channel size matches the channel dimension of input features
- output channel size specifies the channel dimension of output features
3x3 Convolution - Spatially

- 3x3 Conv with No Stride, No Padding
- Weights = [[0, 1, 2], [2,2,0], [0,1,2]]

\[ O_{00} = I_{00} \times W_{00} + I_{01} \times W_{01} + I_{02} \times W_{02} + I_{10} \times W_{10} + I_{11} \times W_{11} + I_{12} \times W_{12} + I_{20} \times W_{20} + I_{21} \times W_{21} + I_{22} \]

* Source: [http://deeplearning.net/software/theano_versions/dev/_images/](http://deeplearning.net/software/theano_versions/dev/_images/)
3x3 Convolution - 3D

* gif from [https://cdn-images-1.medium.com/max/800/1*q95f1mqXAVsj_VMHaOm6Sw.gif](https://cdn-images-1.medium.com/max/800/1*q95f1mqXAVsj_VMHaOm6Sw.gif)
3x3 Convolution - 3D

http://cs231n.github.io/assets/conv-demo/index.html

* gif from https://cdn-images-1.medium.com/max/800/1*q95f1mqXAVsj_VMHaOm6Sw.gif
Fully-Connected Layer (FC)

- Each input activation is connected to every output activation
- Essentially a matrix-vector multiplication

Weights: \( OC \times IC \)

Input Activations: \( IC \times 1 \)

Output Activations: \( OC \times 1 \)
ReLU Activation Function

- Implements the concept of “Firing”
- Introduces non-linearity
- Rectified Linear Unit
  - \( R(z) = \max(0, z) \)
- Not differentiable at 0
Batch Normalization (BN)

- Shifts and scales activations to achieve zero-centered distribution with unit variance
  - Subtracts mean
  - Divides by standard deviation

* images from https://en.wikipedia.org/wiki/Normal_distribution
Pooling

- **Downsamples**
  - Takes the maximum
  - Takes the average
- **Operates at each feature map independently**

* images from http://cs231n.github.io/convolutional-networks/
Full DNN Example: AlexNet

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-1 Accuracy</td>
<td>57.1%</td>
</tr>
<tr>
<td>Top-5 Accuracy</td>
<td>80.2%</td>
</tr>
<tr>
<td>Model Size</td>
<td>61M</td>
</tr>
<tr>
<td>MACs</td>
<td>725M</td>
</tr>
</tbody>
</table>
Full DNN Example: ResNet-34

<table>
<thead>
<tr>
<th>Top-1 Accuracy</th>
<th>73.3%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-5 Accuracy</td>
<td>91.3%</td>
</tr>
<tr>
<td>Model Size</td>
<td>83M</td>
</tr>
<tr>
<td>MACs</td>
<td>2G</td>
</tr>
</tbody>
</table>
Design Methodology
The Roofline Model

- Performance is upper bounded by the peak performance, the communication bandwidth, and the operational intensity.
- Arithmetic Intensity is the ratio of the compute to the memory traffic.

\[
P = \min \left\{ \frac{\pi}{\beta \times I} \right\}
\]

Image from https://en.wikipedia.org/wiki/Roofline_model
The Roofline Model

Log Rooflines for CPU, GPU, TPU

Figure from https://arxiv.org/ftp/arxiv/papers/1704/1704.04760.pdf
Conv2D to Matrix-Matrix Multiplication

- **Im2Col** stores in each column the necessary pixels for each kernel map
  - Duplicates input feature maps in memory
  - Restores output feature map structure

* Image from http://nmhkahn.github.io/CNN-Practice
Im2col Transform

* from https://www.researchgate.net/publication/327070011_Accelerating_Deep_Neural_Networks_on_Low_Power_Heterogeneous_Architectures
Image to column operation (im2col)
Slide the input image like a convolution but each patch become a column vector.

Kernel Width: 2
Kernel Height: 2
Stride: 1
Padding: 0

\[ \begin{align*}
W_{\text{out}} &= (W_{\text{in}} - kW + 2\cdot PW) / S + 1 \\
H_{\text{out}} &= (H_{\text{in}} - KH + 2\cdot PH) / S + 1
\end{align*} \]

\[ \begin{align*}
W_{\text{out}} &= (W_{\text{in}} - 2\cdot PW + 1) / 1 + 1 = 3 \\
H_{\text{out}} &= (H_{\text{in}} - 2\cdot PH + 1) / 1 + 1 = 3
\end{align*} \]

We can multiply this result matrix [12x9] with a kernel [1x12].
result = kernel x matrix
The result would be a row vector [1x9].
We need another operation that will convert this row vector into an image [3x3].

Consider col2im as a row major reshape.

We get true performance gain when the kernel has a large number of filters, i.e. \( F=4 \)
and/or you have a batch of images (\( N=4 \)). Example for the input batch [4x4x3x4], convolved with 4 filters [2x2x3x2].
The only problem with this approach is the amount of memory.

Reshaped kernel: [4x12]
Converted input batch: [12x56]

* Image from https://github.com/numforge/laser/wiki/Convolution-optimisation-resources
Conv2D to Matrix-Vector Multiplication

- For each pixel, we can first perform Matrix-Vector Multiplication along the input channel dimension.
- Then we can use adder-tree to aggregate the sum of K x K pixels (K is the kernel size).
Accelerator Architecture
General Architecture
Systolic Array

- **Systolic Array** is a homogeneous network of tightly coupled data processing units (DPUs).
- Each **DPU** independently computes a partial result as a function of the data received from its upstream neighbors, stores the result within itself and passes it downstream.
- Advantages of systolic array design:
  - Shorter wires -> lower propagation delay and lower power consumption
  - High degree of pipelining -> faster clock
  - High degree of parallelism -> high throughput
  - Simple control logic -> less design efforts
System Architecture

MAC design

\[ C[i][j] = C[i][j] + A[i][k] \times B[k][j] \]

* Images from http://www.telesens.co/2018/07/30/systolic-architectures/
Specialized Architecture
Layer-based Design

Controllers:
- Input
- Weights
- Output
- Output
- Output

Stream Buffer
- DDR
- PE 1
- PE 2
- PE 3
- PE 4
- ... (N-1)
- PE N

Pooling
ReLU
BN

Systolic Array for Convolution / Fully Connected Layer
Execution Model

Conv  ReLu  BN  MaxPool  FC

AlexNet Design
Execution Model

AlexNet Design
Execution Model

Conv  ReLu  BN  MaxPool  FC

AlexNet Design
Execution Model

AlexNet Design
Execution Model

Conv ReLu BN MaxPool FC

AlexNet Design
Execution Model

AlexNet Design
Execution Model

Conv ReLu BN MaxPool FC

AlexNet Design

7
Execution Model

AlexNet Design
Spatial Design

**BRAMs:**
- weights & bias

**Inputs:**
- DDR
  - Conv 3x3
  - ReLU

**Layer1**
- BN

**Layer2**
- Conv 1x1
- ReLU
- BN
- Pool

**LayerN**
- BN
- FC

**...**
Line-Buffer Design

- Buffers inputs to perform spatial operations
- Buffers inputs for reuse to improve the arithmetic intensity

Execution Model

● 2x2 Max Pooling
Execution Model

- 2x2 Max Pooling
Execution Model

- 2x2 Max Pooling

![Image showing 2x2 Max Pooling example]
Execution Model

- 2x2 Max Pooling
Execution Model

- 2x2 Max Pooling
Mixed-precision Processing Elements
Mixed-Precision Processing Element (PE)

Spatial PE: 2-bit mode

Spatial PE: 4-bit mode

Input (4-bit) × Weight (4-bit) = Partial Products

Fusion Unit

4x Parallelism

Spatial Processing Element: 4-8 bit mode

Mixed-precision PE: Temporal vs. Spatial

Bit-Serial: Combines results over time

- Spatial design is normally more efficient in terms of area and power, given the same throughput

Bit-Parallel: Combines results over space

High-level Synthesis
High-Level Synthesis

- Allows users to specify algorithm logic in high-level languages
  - No concept of clock
  - Not specifying register-transfer level activities
- HLS compiler generates RTL based on high-level algorithmic description
  - Allocation
  - Scheduling
  - Binding
- Advantages:
  - Faster development and debugging cycles
  - More structural code
  - Focuses on larger architecture design tradeoffs
HLS Abstraction

- High-level Languages
  - C/C++, OpenCL, GoLang

- Typical hardware mapping
  - C Function -> Verilog Module
  - Function Arguments -> Memory Ports
  - Basic Blocks (blocks without branches) -> Hardware Logic
  - Operators -> Functional Units
  - Arrays -> BRAMs
  - Control Flow Graph (CFG) -> Finite-state Machine (FSM)

- Limitations:
  - No dynamic memory allocation allowed
  - No recursion support
Example: Matrix Multiplication

Step 1: Partition Local Arrays

```c
// Local memory to store input and output matrices
int localA[MAX_SIZE][MAX_SIZE];
#pragma HLS ARRAY_PARTITION variable=localA dim=1 complete

int localB[MAX_SIZE][MAX_SIZE];
#pragma HLS ARRAY_PARTITION variable=localB dim=2 complete

int localC[MAX_SIZE][MAX_SIZE];
#pragma HLS ARRAY_PARTITION variable=localC dim=0 complete
```
Step 2: Design Systolic Array (Implicit)

```c
systolic1: for(int k = 0; k < a_col; k++) {
#pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
#pragma HLS PIPELINE II=1

systolic2: for(int i = 0; i < MAX_SIZE; i++) {
    systolic3: for(int j = 0; j < MAX_SIZE; j++) {

        // Get previous sum
        int last = (k==0) ? 0 : localC[i][j];

        // Update current sum
        // Handle boundary conditions
        int a_val = (i < a_row && k < a_col)? localA[i][k] : 0;
        int b_val = (k < b_row && j < b_col)? localB[k][j] : 0;
        int result = last + a_val*b_val;

        // Write back results
        localC[i][j] = result;
    }
}
```

Step 2: Design
Systolic Array
(Explicit)

```c
#pragma HLS pipeline

for (int r = 0; r < N + 2 * MAX_SIZE - 2; r++) {
    // update data (i.e., reads data from previous PE)
    for (int i = 0; i < MAX_SIZE; i++)
        for (int j = MAX_SIZE - 1; j >= 1; j--)
            localA[i][j] = localA[i][j - 1];

    for (int i = MAX_SIZE - 1; i >= 1; i--)
        for (int j = 0; j < MAX_SIZE; j++)
            localB[i][j] = localB[i - 1][j];

    // read new data from inputs
    // not ok here!
    for (int i = 0; i < MAX_SIZE; i++)
        if (r >= i && r < i + N)
            localA[i][0] = A[i + ii * MAX_SIZE][r - i];
        else
            localA[i][0] = 0;

    for (int j = 0; j < MAX_SIZE; j++)
        if (r >= j && r < j + N)
            localB[0][j] = B[r - j][j + jj * MAX_SIZE];
        else
            localB[0][j] = 0;

    // PE
    for (int i = 0; i < MAX_SIZE; i++)
        for (int j = 0; j < MAX_SIZE; j++)
            C[i + ii * MAX_SIZE][j + jj * MAX_SIZE] += localA[i][j] * localB[i][j];
}```
Step 3: Schedule
Outer Loop
Control Logic and Memory Accesses

* Please see the **SDAccel** page for detailed source code
Resources

- Vivado HLS Design Hubs
- Parallel Programming for FPGAs
- Cornell ECE 5775: High-Level Digital Design Automation
- LegUp: Open-source HLS Compiler
- VTA design example
- Vivado SDAccel design examples
Questions?