Outline

1. Deep Neural Network (DNN) Basics
2. DNN Accelerators
3. High-level Synthesis (HLS)
DNN Basics
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 the influence strength

* Slide from http://cs231n.github.io/
Neural Networks

- NNs are usually feed forward computational graphs constructed from many computational “Neurons”
- The “Neurons”:
  - **Integrate** - typically linear transform (dot-product of receptive field)
  - **Fire** - followed by a non-linear “activation” function

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

- An Neural Network with multiple layers between the inputs and outputs

* Image from Eyeriss Tutorial: [http://eyeriss.mit.edu/tutorial.html](http://eyeriss.mit.edu/tutorial.html)
DNN Examples

- AlexNet 2012 (8 layers)
- GoogLeNet 2014 (22 layers)
- ResNet 2015 (152 layers)
- DenseNet 2016 (dense connections)
- DLA 2017 (deep aggregation)
- NasNet 2017 (NAS design)
Training vs. Inference

Training (supervised)

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

Inference

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

* Slide from https://www.hotchips.org/archives/2010s/hc30/
DNN Applications

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

- Image Classification
  - Sedan: 0.90
  - Motorcycle: 0.02
  - Truck: 0.05
  - Toy: 0.03
  - ...

- Object Detection

- Semantic Segmentation

- Super Resolution

- Activity Recognition
  - Draw Sword: 0.60
  - Stand: 0.02
  - Fence: 0.35
  - Throw: 0.03
  - ...
Nature Language Processing (NLP) Tasks

* Image from “Practical Natural Language Processing”: https://github.com/practical-nlp/practical-nlp
Many Other Tasks

- Recommendation Systems (DLRM)
- Machine Translation (Transformer and GNMT)
- Deep Reinforce Learning (AlphaGo)
DNN Evaluation Metrics

1. Accuracy
2. Computation Complexity
3. Model Size

DNN Accelerators
### 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

> > 112 AI chip companies worldwide ([https://github.com/basicmi/AI-Chip](https://github.com/basicmi/AI-Chip))

![Cloud TPU v3 (45 TFLOP/s)](image1)

![Intel Movidius (4 TFLOP/s)](image2)

![Cambricon-1M IP](image3)
Accelerator Evaluation Metrics

1. Throughput
   ○ Frames per second
2. Latency
   ○ Time to finish one frame
3. Power
4. Energy
5. Hardware Cost
   ○ Resource Utilization

Benchmarks:

https://mlperf.org/
## Example Hardware Comparison

<table>
<thead>
<tr>
<th>#</th>
<th>Metric</th>
<th>Google TPU v3</th>
<th>Nvidia V100</th>
<th>Nvidia A100</th>
<th>Cerebras WSE</th>
<th>GraphCore IPU1</th>
<th>GraphCore IPU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Technology node</td>
<td>&gt;12nm (16 nm est.)</td>
<td>TSMC 12 nm</td>
<td>TSMC 7 nm</td>
<td>TSMC 16 nm</td>
<td>TSMC 16 nm</td>
<td>TSMC 7 nm</td>
</tr>
<tr>
<td>2</td>
<td>Die Area (mm²)</td>
<td>&lt;648 (600 est.)</td>
<td>815</td>
<td>826</td>
<td>46225</td>
<td>900 (est.)</td>
<td>823</td>
</tr>
<tr>
<td>3</td>
<td>Transistor Count (B)</td>
<td>11 (est.)</td>
<td>21</td>
<td>54.2</td>
<td>1200</td>
<td>23.6</td>
<td>59.4</td>
</tr>
<tr>
<td>4</td>
<td>Architecture</td>
<td>Systolic Array</td>
<td>SIMD</td>
<td>SIMD</td>
<td>SIMD</td>
<td>SIMD</td>
<td>SIMD</td>
</tr>
<tr>
<td>5</td>
<td>Theoretical TFLOPS (16-bit mixed precision)</td>
<td>123</td>
<td>125</td>
<td>312</td>
<td>2500</td>
<td>125</td>
<td>250</td>
</tr>
<tr>
<td>6</td>
<td>Freq (GHz)</td>
<td>0.92</td>
<td>1.5</td>
<td>1.4</td>
<td>Unknown</td>
<td>1.6</td>
<td>Unknown</td>
</tr>
<tr>
<td>7</td>
<td>DRAM Capacity (GB)</td>
<td>32</td>
<td>32</td>
<td>80</td>
<td>N/A</td>
<td>N/A</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>DRAM BW (GB/sec)</td>
<td>900</td>
<td>900</td>
<td>2039</td>
<td>N/A</td>
<td>N/A</td>
<td>64 (est.)</td>
</tr>
<tr>
<td>9</td>
<td>Total SRAM Capacity</td>
<td>32MB</td>
<td>36 MB (RF+L1+L2)</td>
<td>87 MB (RF+L1+L2)</td>
<td>18 GB</td>
<td>300 MB</td>
<td>900 MB</td>
</tr>
<tr>
<td>10</td>
<td>SRAM BW (TB/sec)</td>
<td>Unknown</td>
<td>224 @RF+14 @L1+3 @L2</td>
<td>608 @RF+19 @L1+7 @L2</td>
<td>9000</td>
<td>45</td>
<td>47.5</td>
</tr>
<tr>
<td>11</td>
<td>Max TDP (Watts)</td>
<td>450</td>
<td>450</td>
<td>400</td>
<td>20K</td>
<td>150</td>
<td>150 (est.)</td>
</tr>
<tr>
<td>12</td>
<td>GEMM Achievable TFLOPS</td>
<td>98% (120 TFLOPS)</td>
<td>88% (110 TFLOPS)</td>
<td>93% (290 TFLOPS)</td>
<td>Unknown</td>
<td>47% (58 TFLOPS)</td>
<td>61% (154 TFLOPS)</td>
</tr>
<tr>
<td>13</td>
<td>Energy Efficiency (Achievable GEMM TFLOPS/Max Watts)</td>
<td>0.26</td>
<td>0.24</td>
<td>0.72</td>
<td>Unknown</td>
<td>0.39</td>
<td>1.0</td>
</tr>
<tr>
<td>14</td>
<td>Theoretical Energy Efficiency (Theoretical TFLOPS/Max Watts)</td>
<td>0.27</td>
<td>0.27</td>
<td>0.78</td>
<td>0.125</td>
<td>0.83</td>
<td>1.6</td>
</tr>
<tr>
<td>15</td>
<td>Memory Capacity (GB)</td>
<td>16</td>
<td>32</td>
<td>80</td>
<td>18</td>
<td>0.3</td>
<td>112</td>
</tr>
<tr>
<td>16</td>
<td>Memory Efficiency (FLOP/DRAMByte)</td>
<td>133</td>
<td>122</td>
<td>158</td>
<td>N/A</td>
<td>N/A</td>
<td>Unknown</td>
</tr>
<tr>
<td>17</td>
<td>Memory Efficiency (FLOP/SRAMByte)</td>
<td>Unknown</td>
<td>32</td>
<td>35</td>
<td>Unknown</td>
<td>1.28</td>
<td>3.2</td>
</tr>
<tr>
<td>18</td>
<td>Area Efficiency (Achievable TFLOPS/mm²)</td>
<td>0.2</td>
<td>0.13</td>
<td>0.35</td>
<td>Unknown</td>
<td>0.06</td>
<td>0.17</td>
</tr>
<tr>
<td>19</td>
<td>Area Efficiency (Achievable TFLOPS/BTrans)</td>
<td>11</td>
<td>5.2</td>
<td>5.3</td>
<td>Unknown</td>
<td>2.5</td>
<td>2.6</td>
</tr>
</tbody>
</table>

How to design your own DNN accelerator?

1. Understand the basic operations
Common DNN Operations

- Convolution (Groupwise, 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 might have more channels after performing convolution
- We refer to them as feature maps

Channel Dimension

Input Image:

One Feature Map:

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]]

- 3x3 Conv with Stride 2, Padding 1
  - Weights = [[2, 0, 1], [1,0,0], [0,1,1]]

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

* gif from [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
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

Pooling

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

Full DNN Example: AlexNet

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-1 Accuracy</td>
<td>73.3%</td>
</tr>
<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>
How to design your own DNN accelerator?

1. Understand the basic operations
2. Analyze the workload
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.

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

- \(\pi\) - the peak compute performance
- \(\beta\) - the peak bandwidth
- \(I\) - the arithmetic intensity
- The attainable throughput \(P\):

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
How to design your own DNN accelerator?

1. Understand the basic operations
2. Analyze the workload
3. Compare different design options
Conv Mapping 1: 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](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.

Input Image [4x4]

<table>
<thead>
<tr>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>28</td>
<td>48</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

Result: [12x9]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

9 possible Sliding window positions

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].

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].

W\_out=(W\_in - kW + 2\*P)/S + 1
H\_out=(H\_in - kH + 2\*P)/S + 1

W\_out=(4-2)/1+1=3
H\_out=(4-2)/1+1=5

2x2x3 column vector


| 38 | 39 | 40 | 42 | 43 | 44 | 46 | 47 | 48 |

Consider col2lm as a row major reshape.

* Image from https://github.com/numforge/laser/wiki/Convolution-optimisation-resources
Optimization: Winograd Algorithm

**Winograd** performs convolution in a transformed domain to reduce the total number of multiplications.

**GEMM Example:**

Inputs:

\[ f = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}, g = \begin{bmatrix} -1 \\ -2 \\ -3 \end{bmatrix} \]

Transformed Inputs:

\[
\begin{align*}
m_1 &= (d_0 - d_2)g_0 \\
m_2 &= \frac{g_0 + g_1 + g_2}{2} \\
m_3 &= \frac{g_0 - g_1 + g_2}{2} \\
m_4 &= (d_1 - d_3)g_2
\end{align*}
\]

Result:

\[
\begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}
\]

**6 MUL**  **4 MUL**
Conv Mapping 2: 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 $K \times K$ pixels ($K$ is the kernel size).

\[
\text{Input Activations:} \quad \text{Weights:} \quad \text{Output Channels (OC)} \\
\text{Input Channels (IC)} \quad \vdots \quad \text{Output Image:} \quad \vdots \quad \text{Output Channels (OC)} \\
\text{Weights:} \quad \text{Input Activations:} \quad \text{Partial Sums} \\
\text{Output Channels (OC)} \quad \vdots \quad \text{OC} \quad \vdots \quad \text{OC}
\]

\[
\text{Input Channels (IC)} \quad \vdots \quad \text{IC} \quad \vdots \quad \text{IC}
\]

\[
\begin{align*}
1 & \times 1 = 1 \\
& \vdots \\
1 & \times 1 = 1
\end{align*}
\]

\[
\begin{align*}
1 & \times 1 = 1 \\
& \vdots \\
1 & \times 1 = 1
\end{align*}
\]
Implementation: 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/*
DNN Accelerator Design 1: Layer-based

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

Stream Buffer

Pooling

ReLU

BN

Systolic Array for Convolution / Fully Connected Layer

PE 1 → PE 2 → PE 3 → PE 4 → ... → PE N-1 → PE N
DNN Accelerator Design 2: Spatially-mapped

BRAMs:
weights & bias

Inputs
DDRR
Conv 3x3
ReLU

BN

Layer1

weights & bias

Conv 1x1
ReLU
Pool

Layer2

weights & bias

FC

LayerN

...
Line-Buffer Design

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

Line-Buffer Execution Model

- 2x2 Max Pooling
Line-Buffer Execution Model

- 2x2 Max Pooling
Line-Buffer Execution Model

- 2x2 Max Pooling

```
4  2  5  6  9
1  3  8  7  3
6  4  2  8  1
4
```
Line-Buffer Execution Model

- 2x2 Max Pooling
Line-Buffer Execution Model

- 2x2 Max Pooling
### How to design your own DNN accelerator?

1. Understand the basic operations
2. Analyze the workload
3. Compare different design options
4. Develop software runtime
Execution Model

AlexNet Design
Execution Model

AlexNet Design
Execution Model

AlexNet Design
Execution Model

AlexNet Design
Execution Model

AlexNet Design
Execution Model

AlexNet Design
Execution Model

AlexNet Design
Execution Model

AlexNet Design
HLS
High-Level Synthesis (HLS)

- 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)
Step 3: Schedule
Outer Loop
Control Logic and
Memory Accesses

* Please see the [SDAccel page](https://www.sdadvisor.com/sdarknet/docs) for detailed source code
Resources

- EE290-2: Hardware for Machine Learning
- MIT Everiss Tutorial
- 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?