Specific Architectures for DNNs

Kanishkan Vadivel, Henk Corporaal, Pekka Jääskeläinen

Electrical Engineering - Electronic Systems group
Outline

• Introduction
• Background on DNN computations
• DNN on Traditional Compute Platforms (CPU, DSP, and GPU)
  • Accelerators (ASICs) for DNN
    • Design for data reuse
    • Performance evaluation framework
    • Practical Example
  • Specialized Architectures
Reference

Slides are Based on Eyeriss Tutorial, ISCA 2019.

• “DNN Accelerator Architecture” and
• “RS Dataflow and NoC”

ISCA Tutorial (2019)
Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen
Compute Paradigms

• Example

\[ y = Ax^2 + Bx + C \]

• Implementation
  • Temporal Architecture
    • Programmable computing element that runs pre-defined set of instructions
  • Spatial Architecture
    • Defined by **fixed functionality** and **connectivity** of the Hardware elements
Compute Paradigms

• Temporal Computing

\[ y = Ax^2 + Bx + C \]

Source: Lecture notes, EECC550 – Shaaban 2012
Compute Paradigms

• Temporal Computing

\[ y = Ax^2 + Bx + C \]

Source: Lecture notes, EECC550 – Shaaban 2012
Compute Paradigms

• Temporal Computing

\[ y = Ax^2 + Bx + C \]

Source: Lecture notes, EECC550 – Shaaban 2012
Compute Paradigms

- Spatial Computing

Spatial (using Hardware)

Defined by fixed functionality and connectivity of hardware elements

Source: Lecture notes, EECC550 – Shaaban 2012
Compute Paradigms

• Spatial Computing

Spatial (using Hardware)

Defined by fixed functionality and connectivity of hardware elements

Source: Lecture notes, EECC550 – Shaaban 2012
Compute Paradigms

• Spatial Computing

Spatial (using Hardware)

Defined by fixed functionality and connectivity of hardware elements

Source: Lecture notes, EECC550 – Shaaban 2012
Compute Paradigms

• Spatial Computing

Spatial (using Hardware)

Defined by fixed functionality and connectivity of hardware elements

Source: Lecture notes, EECC550 – Shaaban 2012
Spatial Architecture for DNN

- Global Buffer (100s – 1000s kB)
- On-Chip Network (NoC)
  - Global Buffer to PE
  - PE to PE
- Processing Element (PE)
- Reg File: 1 – 10 kB
- Control
Spatial Data Reuse
Data Access Cost

Normalized Energy Cost:

- 1x (Reference)
- 1x
- 2x
- 6x
- 200x
What we can reuse?
Types of Data Reuse in DNN

**Convolutional Reuse**

CONV layers only
(sliding window)

Reuse: Activations
Filter weights
Types of Data Reuse in DNN

Convolutional Reuse

CONV layers only
(sliding window)

Reuse:
- Activations
- Filter weights
Types of Data Reuse in DNN

**Convolutional Reuse**
CONV layers only  
(sliding window)

**Reuse:**
- Activations
- Filter weights

**Fmap Reuse**
CONV and FC layers

**Reuse:** Activations
Types of Data Reuse in DNN

**Convolutional Reuse**
CONV layers only (sliding window)

- **Filter**
- **Input Fmap**

Reuse: Activations, Filter weights

**Fmap Reuse**
CONV and FC layers

- **Filters**
- **Input Fmap**

Reuse: Activations

**Filter Reuse**
CONV and FC layers (batch size > 1)

- **Filter**
- **Input Fmaps**

Reuse: Filter weights
If all data reuse is exploited, DRAM accesses in AlexNet can be reduced from **2896M** to **61M** (best case).
Dataflow Taxonomy

- Output Stationary (OS)
- Weight Stationary (WS)
- Input Stationary (IS)
Convolution Layer - Output Stationary

MACs = R x S x (H – R + 1) x (W - S + 1) x C
Convolution Layer - Output Stationary

Many output fmap weights

Number of MACs = R x S x (H – R + 1) x (W - S + 1) x C x M
Variants of Output Stationary

Parallel Output Region

# Output Channels: Multiple
# Output Activations: Single
Variants of Output Stationary

Parallel Output Region

<table>
<thead>
<tr>
<th># Output Channels</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td># Output Activations</td>
<td>Single</td>
</tr>
</tbody>
</table>
Variants of Output Stationary

Parallel Output Region

<table>
<thead>
<tr>
<th></th>
<th>$\text{OS}_C$</th>
<th>$\text{OS}_B$</th>
</tr>
</thead>
<tbody>
<tr>
<td># Output Channels</td>
<td>Multiple</td>
<td>Multiple</td>
</tr>
<tr>
<td># Output Activations</td>
<td>Single</td>
<td>Multiple</td>
</tr>
</tbody>
</table>
# Variants of Output Stationary

<table>
<thead>
<tr>
<th></th>
<th>$\text{OS}_C$</th>
<th>$\text{OS}_B$</th>
<th>$\text{OS}_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallel</strong></td>
<td><img src="DiagramC.png" alt="Diagram" /></td>
<td><img src="DiagramB.png" alt="Diagram" /></td>
<td><img src="DiagramA.png" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Output Region</strong></td>
<td><img src="DiagramC.png" alt="Diagram" /></td>
<td><img src="DiagramB.png" alt="Diagram" /></td>
<td><img src="DiagramA.png" alt="Diagram" /></td>
</tr>
<tr>
<td><strong># Output Channels</strong></td>
<td>Multiple</td>
<td>Multiple</td>
<td>Single</td>
</tr>
<tr>
<td><strong># Output Activations</strong></td>
<td>Single</td>
<td>Multiple</td>
<td>Multiple</td>
</tr>
</tbody>
</table>
1-D Convolution – Output Stationary

Weights \( R \) \* Inputs \( H \) = Outputs \( E \)

\[ E = H - R + 1 \]

```c
int I[H]; // Input activations
int W[R]; // Filter Weights
int O[E]; // Output activations

for (e = 0; e < E; e++)
    for (r = 0; r < R; r++)
        O[e] += I[e+r] * W[r];
```
Output Stationary - Reference Pattern

for (\(e = 0; e < E; e++\))
  for (\(r = 0; r < R; r++\))
    \[O[e] \leftarrow I[e+r] \times W[r];\]

Layer Shape:
- \(H = 12\)
- \(R = 4\)
- \(E = 9\)
Output Stationary - Reference Pattern

Observations:
- Single output is reused many times (R) [0 reads]
Observations:

- Single **output** is reused many times (R)
- All **weights** reused repeatedly  **[E x R reads]**
Output Stationary - Reference Pattern

Observations:
- Single **output** is reused many times (R)
- All **weights** reused repeatedly
- Sliding window of **inputs** (size = R)  

[ExR reads]
Output Stationary – Summary

- Minimize partial sum R/W
- Local accumulation
- Minimizes energy

```c
for (e = 0; e < E; e++)
   for (r = 0; r < R; r++)
      O[e] += I[e+r] * W[r];
```
OS Example: ENVISION - ISSCC 2017
Weight Stationary

```c
for (r = 0; r < R; r++)
    for (e = 0; e < E; e++)
        O[e] += I[e+r] * W[r];
```

<table>
<thead>
<tr>
<th></th>
<th>OS</th>
<th>WS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACs</td>
<td>E*R</td>
<td>E*R</td>
</tr>
<tr>
<td>Weight Reads</td>
<td>E*R</td>
<td>R</td>
</tr>
<tr>
<td>Input Reads</td>
<td>E*R</td>
<td>E*R</td>
</tr>
<tr>
<td>Output Reads</td>
<td>0</td>
<td>E*R</td>
</tr>
<tr>
<td>Output Writes</td>
<td>E</td>
<td>E*R</td>
</tr>
</tbody>
</table>
Weight Stationary - Reference Pattern

Observations:
- Single weight is reused many times (E)
- Large sliding window of inputs (size = E)
- Fixed window of outputs (size = E)
WS Example: ISSAC - ISCA 2016

MAC Array for VMM
WS Example: ISSAC - Simplified View
Input Stationary

Weights (R) \* Inputs (H) = Outputs (E = H-R+1)

```c
int I[H]; // Input activations
int W[R]; // Filter weights
int O[E]; // Output activations

for (e = 0; e < E; e++)
    for (r = 0; r < R; r++)
        O[e] += I[e+r] * W[r];
```

How can we implement input stationary with no input index?
Input Stationary

Weights (R) * Inputs (H) = Outputs (E = H-R+1)

```c
int I[H];    // Input activations
int W[R];    // Filter weights
int O[E];    // Output activations

for (e = 0; e < E; e++)
    for (r = 0; r < R; r++)
        O[e] += I[e+r] * W[r];
```

How can we implement input stationary with no input index?
Input Stationary

Weights (R) \* Inputs (H) = Outputs (E = H-R+1)

```c
int I[H];     // Input activations
int W[R];     // Filter weights
int O[E];     // Output activations

for (e = 0; e < E; e++)
    for (r = 0; r < R; r++)
        O[e] += I[e+r] * W[r];
```

How can we implement input stationary with no input index?

```c
int I[H];     // Input activations
int W[R];     // Filter weights
int O[E];     // Output activations

for (h = 0; h < H; h++)
    for (r = 0; r < R; r++)
        O[h-r] += I[h] * W[r];
```
Input Stationary - Reference Pattern

Observations:
- **Inputs** used repeatedly (R times)
- **Weights** reused in large window (size = R)
- Sliding window of **outputs** (size = R)
Minimum Costs

<table>
<thead>
<tr>
<th></th>
<th>OS</th>
<th>WS</th>
<th>IS</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACs</td>
<td>E*R</td>
<td>E*R</td>
<td>E*R</td>
<td>E*R</td>
</tr>
<tr>
<td>Weight Reads</td>
<td>E*R</td>
<td>R</td>
<td>E*R</td>
<td>R</td>
</tr>
<tr>
<td>Input Reads</td>
<td>E*R</td>
<td>E*R</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>Output Reads</td>
<td>0</td>
<td>E*R</td>
<td>E*R</td>
<td>0</td>
</tr>
<tr>
<td>Output Writes</td>
<td>E</td>
<td>E*R</td>
<td>E*R</td>
<td>E</td>
</tr>
</tbody>
</table>

Assume: $W \sim= E$

Which one is the best solution?
Can we do better?

Output Stationary

Global Buffer

Activation

Weight

Psum

Weight Stationary

Global Buffer

Psum

Activation

Weight

Input Stationary

Global Buffer

Psum

Weight

Act
Row Stationary - Motivation

• Maximize data reuse at RF
• Optimize for **overall** Energy Efficiency
Eyriiss - ISCA 2019 Tutorial

- “RS Dataflow and NoC” slides 4 to 33
Can we still do better?
Can we still do better?
Can we still do better?

```c
int I[H];  // Input activations
int W[R];  // Filter Weights
int O[E];  // Output activations

// Level 1
for (e1 = 0; e1 < E1; e1++)
    for (r1 = 0; r1 < R1; r1++)
        // Level 0
            for (e0 = 0; e0 < E0; e0++)
                for (r0 = 0; r0 < R0; r0++)
                    O[e1*E0+e0] += I[e1*E0+e0 + r1*R0+r0] * W[r1*R0+r0];
```

Note E and R are factored so:
E0*E1 = E
R0*R1 = R
Performance Evaluation Framework

- Energy efficiency of DNN is dependent on the mappings
- Fast methods are needed for Design space exploration of different DNN accelerators

Fig. 1. Histogram of the energy efficiency of various mappings of VGG_conv3_2 on an example architecture.
Performance Evaluation Framework

- Energy efficiency of DNN is dependent on the mappings
- Fast methods are needed for Design space exploration of different DNN accelerators

Fig. 1. Histogram of the energy efficiency of various mappings of VGG_conv3_2 on an example architecture.
Performance Evaluation Framework

- Energy efficiency of DNN is dependent on the mappings
- Fast methods are needed for Design space exploration of different DNN accelerators

Fig. 1. Histogram of the energy efficiency of various mappings of VGG_conv3_2 on an example architecture.
Performance Evaluation
Practical Example: Google TPU

TPU v1 (Edge) 2016

TPU Data centre
Practical Example: Google TPU v1 (2016)

- The Matrix Unit: 65,536 (256x256) 8-bit multiply-accumulate units
- 700 MHz clock rate
- Peak: 92T operations/second
  - $65,536 \times 2 \times 700M$
- >25X as many MACs vs GPU
- >100X as many MACs vs CPU
- 4 MiB of on-chip Accumulator memory
- 24 MiB of on-chip Unified Buffer (activation memory)
- 3.5X as much on-chip memory vs GPU
- Two 2133MHz DDR3 DRAM channels
- 8 GiB of off-chip weight DRAM memory
Practical Example: Google TPU

• “Systolic Execution” to compute data on the fly in buffers
• Relies on data from different directions arriving at cells in an array at regular intervals and being combined
Practical Example: Google TPU

Inputs

Weights

\[
\begin{align*}
(x_{11}w_{11} + x_{12}w_{12} + x_{13}w_{13}) (x_{11}w_{21} + x_{12}w_{22} + x_{13}w_{23}) \\
(x_{21}w_{11} + x_{22}w_{12} + x_{23}w_{13}) (x_{21}w_{21} + x_{22}w_{22} + x_{23}w_{23}) \\
(x_{31}w_{11} + x_{32}w_{12} + x_{33}w_{13}) (x_{31}w_{21} + x_{32}w_{22} + x_{33}w_{23})
\end{align*}
\]
Practical Example: Google TPU

y1 = x11 * w11
Practical Example: Google TPU

Weights

\[ y_{11} = x_{11} \times x_{11} + w_{12} \times x_{12} \]
\[ y_{21} = w_{11} \times x_{21} \]
\[ y_{12} = w_{21} \times x_{11} \]
Practical Example: Google TPU

Weights

\[ y_{11} = x_{11} \times x_{11} + w_{12} \times x_{12} + w_{13} \times x_{13} \]
\[ y_{21} = w_{11} \times x_{21} + w_{12} \times x_{22} \]
\[ y_{12} = w_{21} \times x_{11} + w_{22} \times x_{12} \]
\[ y_{31} = w_{11} \times x_{31} \]
\[ y_{22} = w_{21} \times x_{21} \]
Practical Example: Google TPU

Weights

\[ y_{11} = x_{11} \times x_{11} + w_{12} \times x_{12} + w_{13} \times x_{13} \]
\[ y_{21} = w_{11} \times x_{21} + w_{12} \times x_{22} + w_{13} \times x_{23} \]
\[ y_{12} = w_{21} \times x_{11} + w_{22} \times x_{12} + w_{23} \times x_{12} \]
\[ y_{31} = w_{11} \times x_{31} + w_{12} \times x_{32} \]
\[ y_{22} = w_{21} \times x_{21} + w_{22} \times x_{22} \]
\[ y_{32} = w_{21} \times x_{31} \]
Practical Example: Google TPU

Weights

\[ y_{11} = x_{11} \times x_{11} + w_{12} \times x_{12} + w_{13} \times x_{13} \]
\[ y_{21} = w_{11} \times x_{21} + w_{12} \times x_{22} + w_{13} \times x_{23} \]
\[ y_{12} = w_{21} \times x_{11} + w_{22} \times x_{12} + w_{23} \times x_{12} \]
\[ y_{31} = w_{11} \times x_{31} + w_{12} \times x_{32} + w_{13} \times x_{33} \]
\[ y_{22} = w_{21} \times x_{21} + w_{22} \times x_{22} + w_{23} \times x_{23} \]
\[ y_{32} = w_{21} \times x_{31} + w_{22} \times x_{32} \]
Practical Example: Google TPU

Weights

\[
\begin{align*}
y_{11} &= x_{11} \cdot w_{11} + x_{12} \cdot w_{12} + x_{13} \cdot w_{13} \\
y_{21} &= w_{11} \cdot x_{21} + w_{12} \cdot x_{22} + w_{13} \cdot x_{23} \\
y_{12} &= w_{21} \cdot x_{11} + w_{22} \cdot x_{12} + w_{23} \cdot x_{13} \\
y_{31} &= w_{11} \cdot x_{31} + w_{12} \cdot x_{32} + w_{13} \cdot x_{33} \\
y_{22} &= w_{21} \cdot x_{21} + w_{22} \cdot x_{22} + w_{23} \cdot x_{23} \\
y_{32} &= w_{21} \cdot x_{31} + w_{22} \cdot x_{32} + w_{23} \cdot x_{33}
\end{align*}
\]
Practical Example: Google TPU
Practical Example: Google TPU v2/v3

TPU v2 - 4 chips, 2 cores per chip

TPU v3 - 4 chips, 2 cores per chip
Summary

• Data reuse is key to achieving **high Energy Efficiency**
• **Hardware** capability and **Mapping** technique plays major role in performance, Energy Efficiency, and Flexibility for DNN accelerators
Outline

• Introduction
• Background on DNN computations
• DNN on Traditional Compute Platforms (CPU, DSP, and GPU)
  • Accelerators (ASICs) for DNN
    • Design for data reuse
    • Performance evaluation framework
    • Practical Example
  • Specialized Architectures
Specialized Architectures

TIE-51257 Parallel Computing

Roberto Airoldi, Nokia Networks
roberto.airoldi@nokia.com

Content by Roberto Airoldi (Nokia), Kanishkan Vadivel (TU Eindhoven) and Pekka Jääskeläinen (TAU)
Motivation for Customized Hardware

“People who are really serious about software should make their own hardware” - Alan Kay
Motivation for Customized Hardware

Highly conflicting requirements: Low-power, High processing requirement, and Flexible

- **General SW-based solution**: Highly flexible and powerful but not energy-efficient
- **ASICs**: Power/performance/cost optimized, but not flexible
- **FPGAs**: Low processing latency, high processing capabilities, somewhat flexible but some power limitations

FPGA doesn’t seem to be too bad. Is it the winner? Hmm, not necessarily
FPGAs

- Historically composed of 3 key building blocks
  - IO Blocks, Interconnects, Logic Blocks
- Massively parallel devices
  - HW re-configurable / programmable
  - ---Cost (very expensive)
- Nowadays
  - Way more complex architectures. Generally includes, ARM core cluster, advanced DSP Blocks, ASIPs, Memory controller, IO interface, and so on
  - Interesting example: Xilinx AI Engines
    [https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf]
Trade-off in System Design

• What is the best Performance vs Flexibility trade-off? (i.e. Best mix of HW and SW)

  • Challenging, but no magic formulas. It is a trade-off between opposite requirements and unknowns.

Which one is the right HW/SW split?
The only way to figure that out is to model it. SoC modeling allows you to play with what/if scenarios
Hardware-Software Co-Design

- SW + HW Solution
  - Blend of HW and SW
  - Bring HW acceleration towards SW, or vice-versa
  - New class of processors, highly specialized for a certain (set of) tasks

Application Specific Instruction-set Processors (ASIP)
ASIPs - Design Space

Source: ChipEx2016, Synopsys
ASIP - Example

• Example kernel: Matrix Multiplication (formally, gemm)
• How the algorithm look like? 3-nested loop
• Does it require some special processing/flow control? FMA operations would definitely help
• What about data type? DNN- 1/2/4/8-bit int, and bfloat
• Can we exploit any type of parallelism? Yes, DLP (data level parallelism). ILP, not so much!
Gemm - Example

- Native Temporal architecture - **Not so efficient**
- How can we improve?
  - A new set of custom instructions
    - Custom compute operations (MACs at low precision. E.g. INT8 multiplication and INT32 accumulations)
    - SIMD – Exploit DLP
  - Local memory for data reuse (how about the area trade-offs?)
    - Trade off is area overhead and latency
    - How about SIMD Load/Store? (Vector LD/ST units will help)
  - Custom datapath – Wide/bit-precise datapath. We can even create a spatial model
Gemm - Example

• What else can be done?
  • Take advantage of data parallelism: I can work with multiple matrices at the same time (SIMD and multi-processing)
• Ok, but how wide is SIMD?

  It’s a trade-off: how much area do I want to spend for it? Power? Does it have a negative impact on frequency?

There are plenty of optimizations to further explore!! The design – benchmarking – validation can be a very time consuming task. Especially if our plans turn out to be not very effective. Luckily, we can rely on tools to speed up this exploration
ASIP Toolset Example - Synopsys
ASIP Toolset Example - TCE Toolset
Conclusion

- A true architecture freedom with ASIPs (large design space)
- It is up to the user to exploit the optimizations
  - Energy/Area/Performance trade-off for specific application
- Wide application domain. Quite few tape-outs, and used in practical applications.