



Designing CNN Accelerators

Day 1

Hyoukjun Kwon
(hyoukjun@gatech.edu)

Georgia Institute of Technology
Synergy Lab (<http://synergy.ece.gatech.edu>)

@SNU

Dec 26, 2017

Goals of This Lecture

- Understanding convolutional neural network
(CNN) computation for inference
- **Learn how to implement hardware** using Bluespec System Verilog (BSV)
- Implement a simplified **CNN accelerator** using BSV
- Understand research opportunity around deep learning accelerators

Lecture Schedule

- **Day 1**
 - Convolutional Neural Networks (CNNs)
 - Bluespec System Verilog (BSV) Basic Syntax and Combinational Logic Implementation
- **Day 2**
 - BSV sequential logic and execution model
 - Traffic in CNN Accelerators
- **Day 3**
 - Processing Element
 - Hierarchical Module Design with BSV

Day 1 Agenda

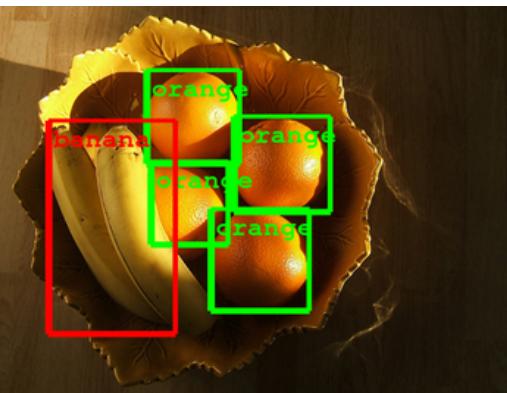
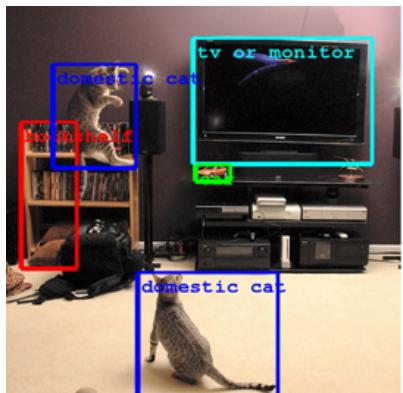
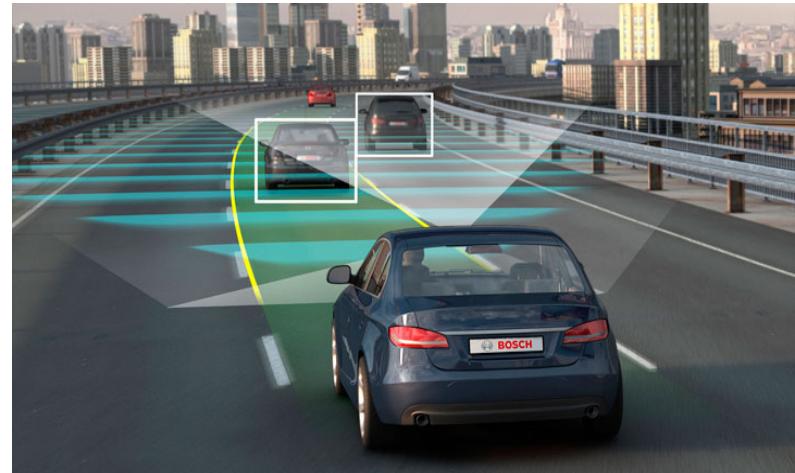
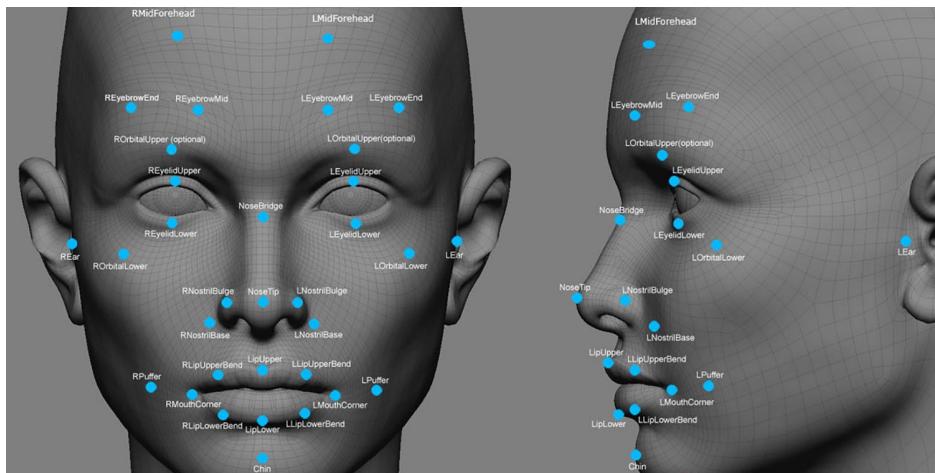
- **Convolutional Neural Networks (CNNs)**
 - Applications
 - CNN structure
 - Layer structure and computation
 - CNN accelerator structure overview
- **Bluespec System Verilog (BSV)**
 - BSV Overview
 - Basic Syntax
 - Combinational logic

Deep Learning Algorithms

- **Convolutional Neural Network (CNN)**
 - Convolution-based deep neural network
 - Currently, the most popular DNN
- **Recurrent Neural Network (RNN)**
 - Considers temporal context
 - Emerging DNN
- **Spiking Neural Network (SNN)**
 - Mimic brain activity
 - Alternative DNN

CNN Applications

- Image/Video recognition



1 1 5 4 3
7 5 3 5 3
5 5 9 0 6
3 5 2 0 0

CNN Applications

- Natural Language Processing (NLP)

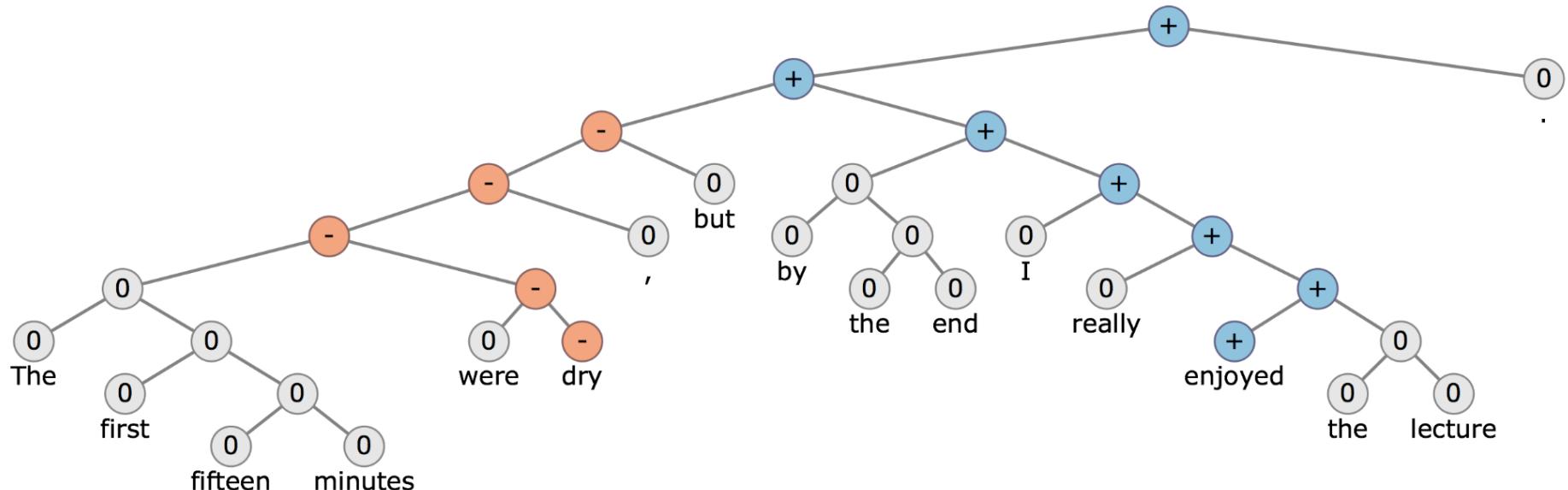


Image source: Standford CS224n (<http://web.stanford.edu/class/cs224n/>)

* Recurrent Neural Network (RNN) is better for accuracy

CNN Applications

- Drug Discovery

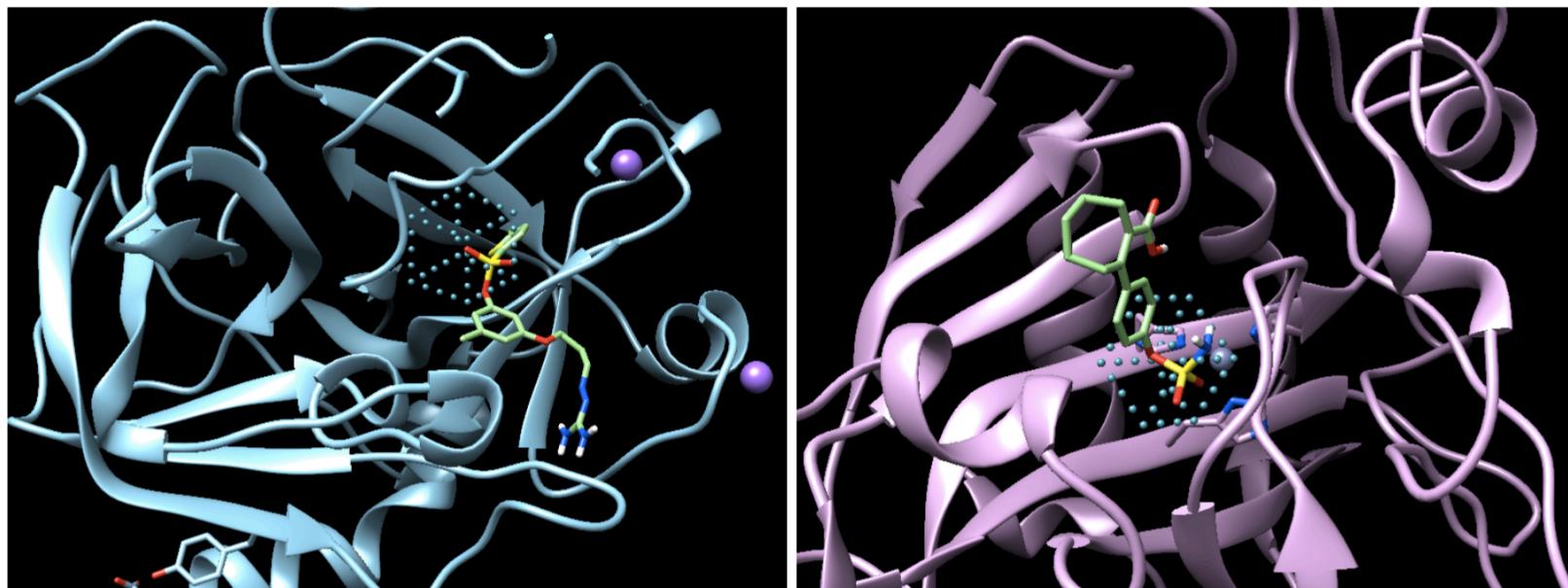


Image source: I. Wallach et al., AtomNet: A Deep Convolutional Neural network for Bioactivity Prediction in Structure-based Drug Discovery, arXiv:1510.02855, 2015

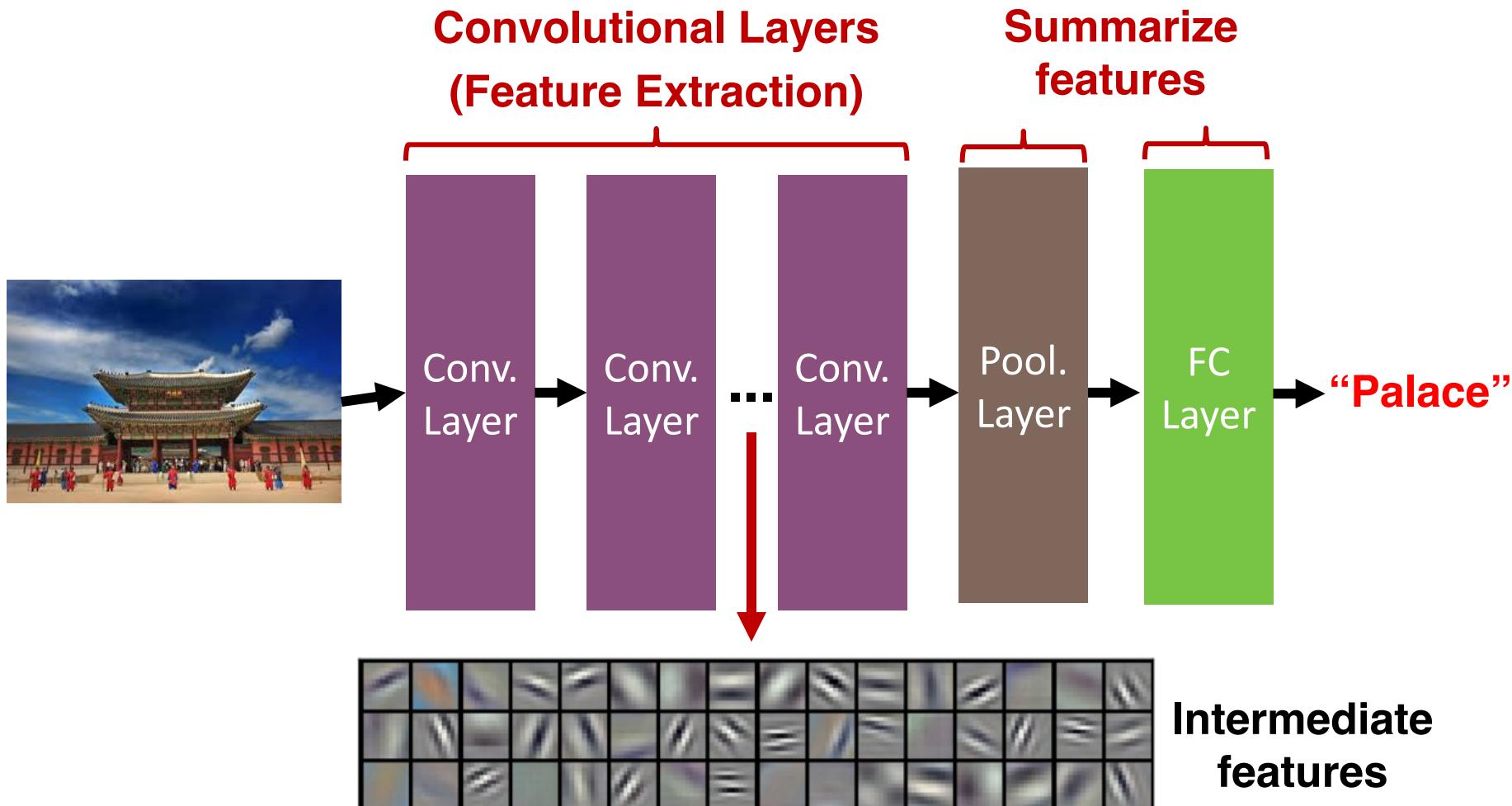
Training vs. Inference

- **Training:** Tuning parameters using training data
 - Stochastic gradient descent is the most popular algorithm
 - Training in data centers and distributing trained data is a common model
 - Because training algorithm changes rapidly, GPU cluster is the most popular hardware (**Low demand for application-specific accelerators**)
- **Inference:** Determining class of a new input data
 - Using a trained model, determine class of a new input data
 - Inference usually occurs close to clients
 - Low-latency and power-efficiency is required (**High demand for application specific accelerators**)

Day 1 Agenda

- **Convolutional Neural Networks (CNNs)**
 - Applications
 - CNN structure
 - Layer structure and computation
 - CNN accelerator structure overview
- **Bluespec System Verilog**
 - BSV Overview
 - Basic Syntax
 - Combinational logic

CNN Structure Overview



Realistic CNN Structure (Alexnet)

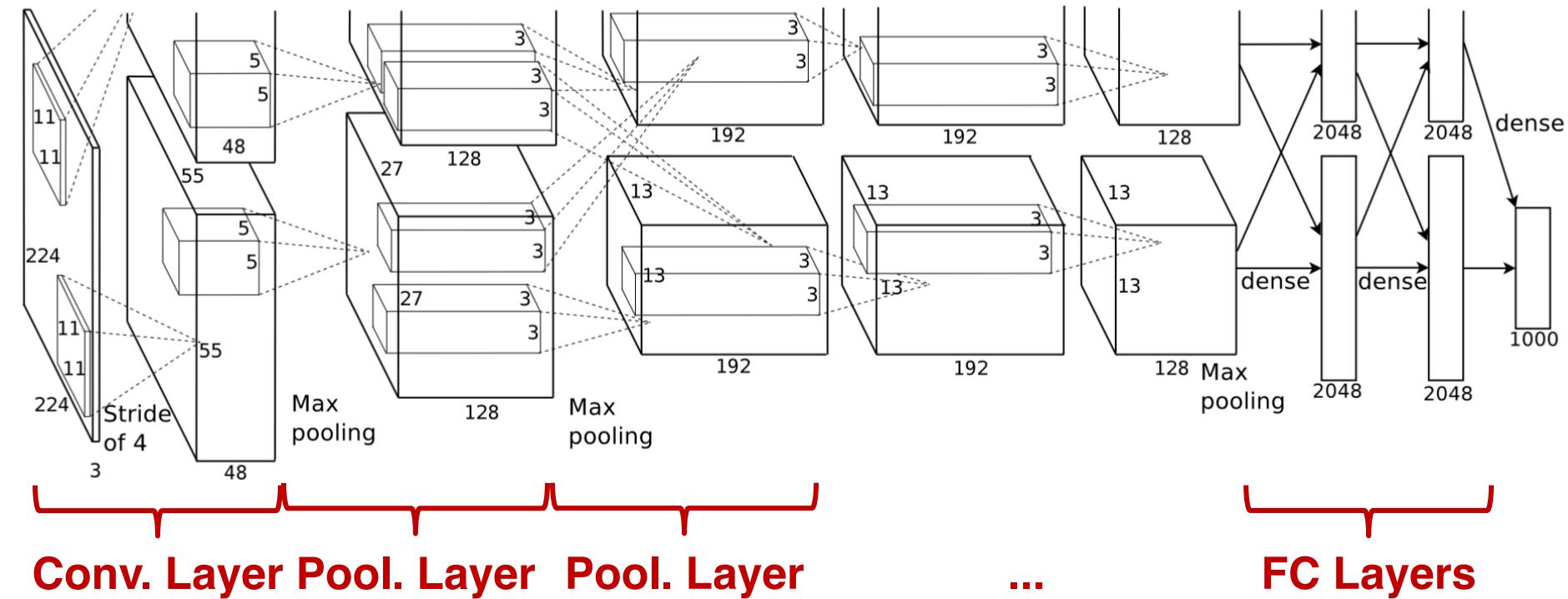


Image source: Alex Krizhevsky et al., ImageNet Classification with Deep Convolutional Neural Networks, NIPS, 2012

Realistic CNN Structure (VGGNet-16)

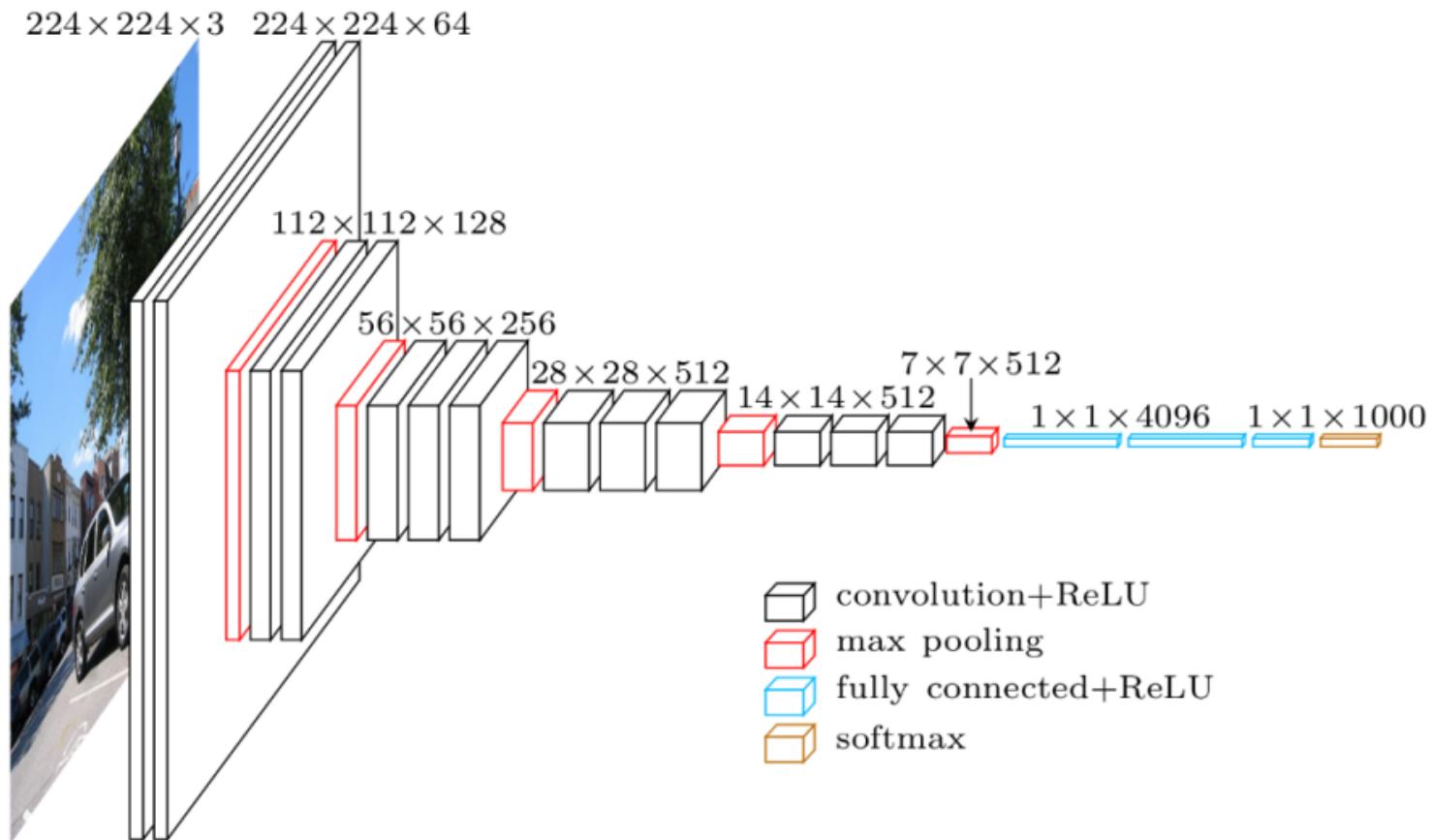


Image source: Heuritech blog (<https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/>)

ResNet, GoogleNet, etc.

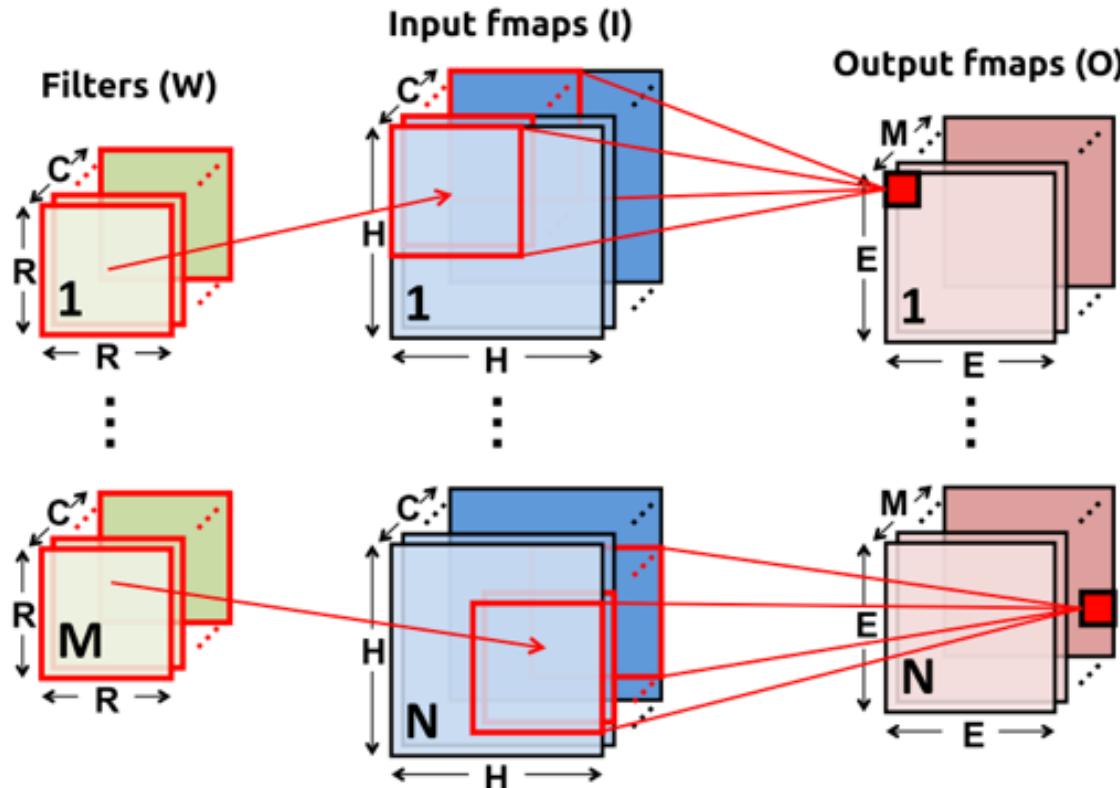
Day 1 Agenda

- **Convolutional Neural Networks (CNNs)**
 - Applications
 - CNN structure
 - Layer structure and computation
 - CNN accelerator structure overview
- **Bluespec System Verilog (BSV)**
 - BSV Overview
 - Basic Syntax
 - Combinational logic

Layers in CNN

- **Convolutional Layer**
 - Feature extraction
 - The most computation-dominant layer in CNNs
- **Pooling Layer**
 - Reduce the dimension of input/output feature map
- **Activation Layer**
 - Normalize input/output feature map values
- **Fully-connected Layer**

Convolutional Layer: Overview



- **Sliding window operation over input featuremaps**

Image source: Y. Chen et al., *Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks*, ISCA 2016

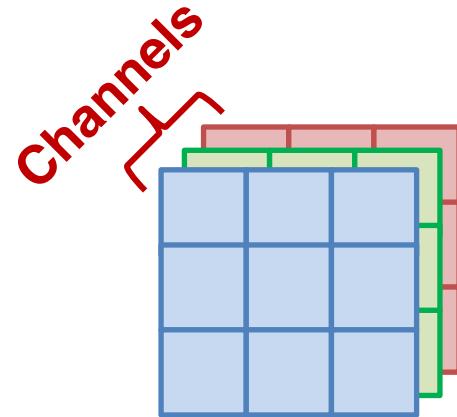
Convolutional Layer: Computation

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
    for(m=0; m<M; m++) { // Weight Filters
        for(c=0; c<C; c++) { // IFMap/Weight Channels
            for(y=0; y<H; y++) { // Input feature map row
                for(x=0; x<W; x++) { // Input feature map column
                    for(j=0; j<R; j++) { // Weight filter row
                        for(i=0; i<S; i++) { // Weight filter column
                            O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```



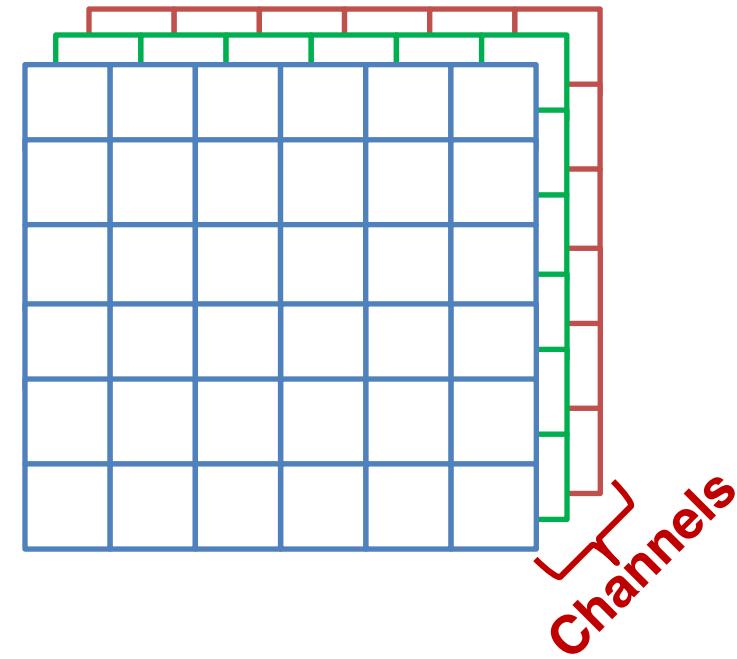
Accumulation Multiplication

Convolutional Layer: Sliding Window Operation



**Multi-dimensional
2D Filters**

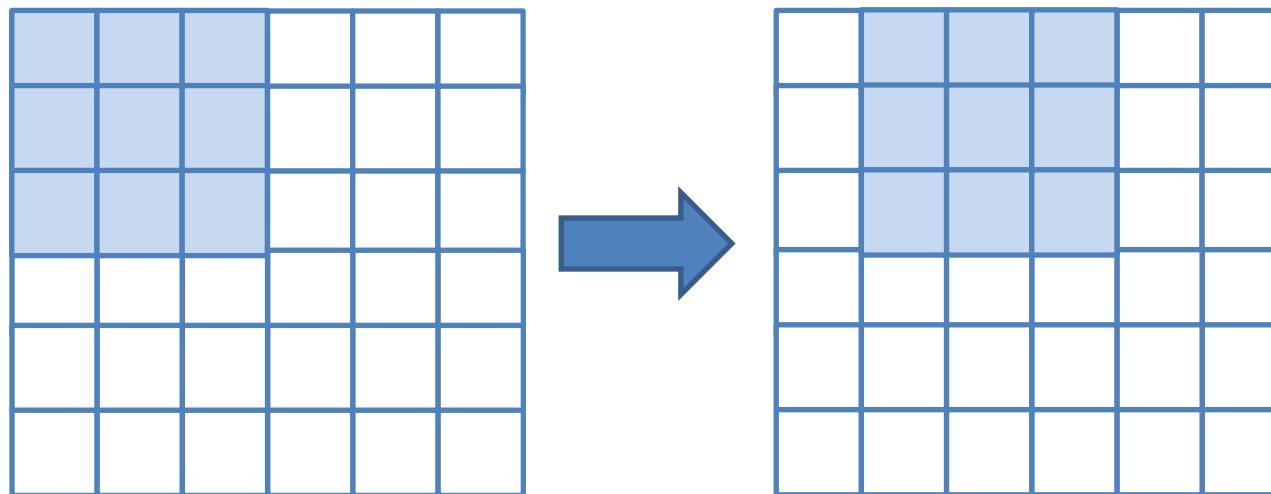
Trained Data



**Input featuremaps
(Input image)**

Data to process

Convolutional Layer: Sliding Window Operation

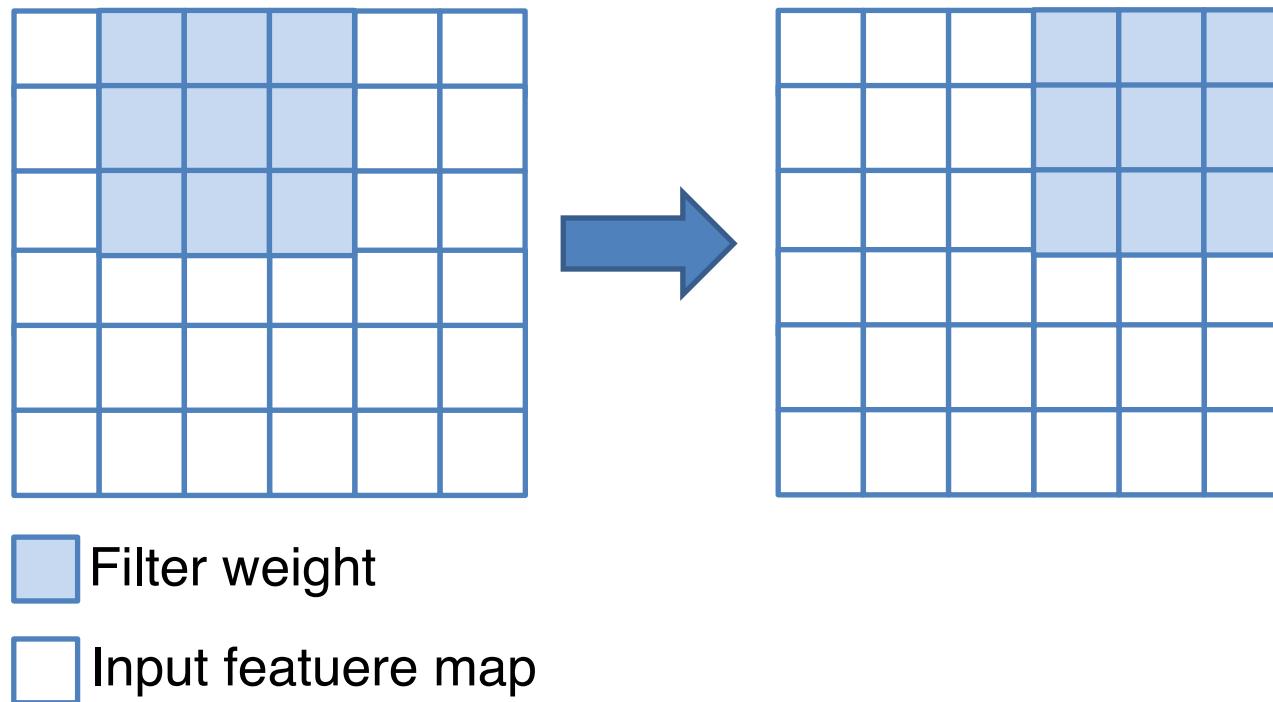


 Filter weight

 Input feature map

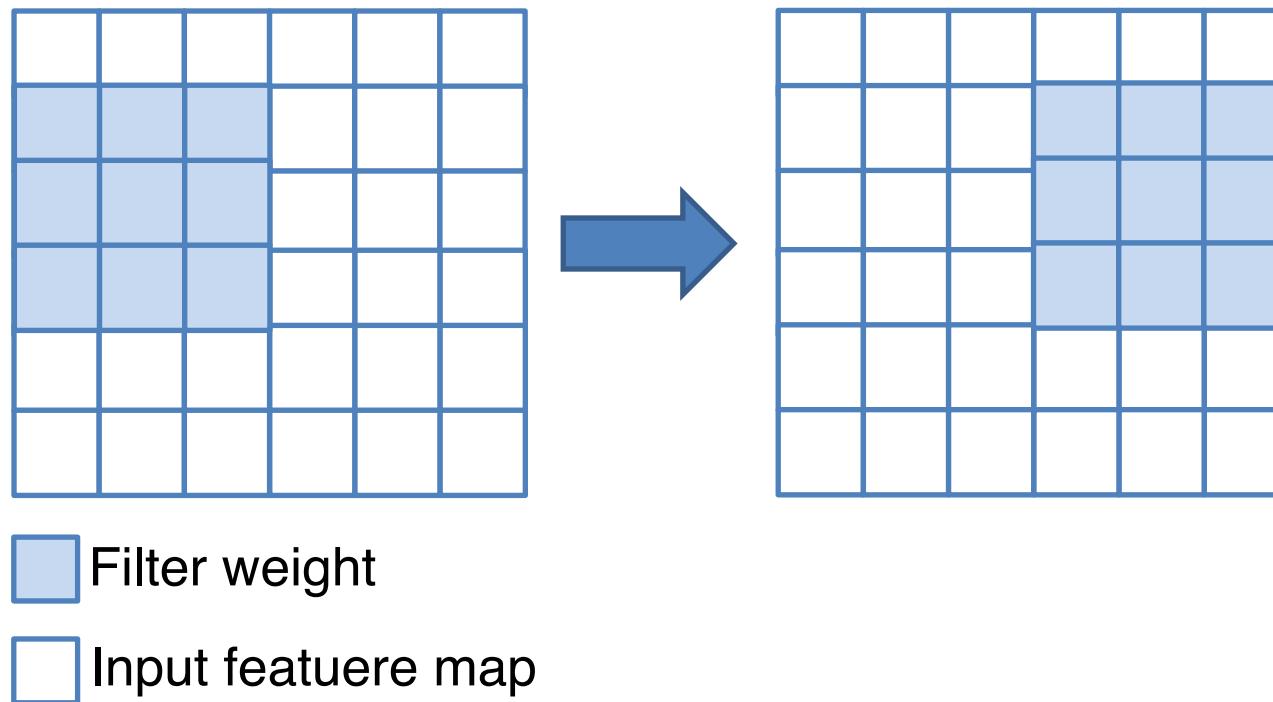
- 1) Multiply each element ($\text{input} * \text{filter}$) “Partial Sum”
- 2) Accumulate all the ($\text{input} * \text{filter}$) values
- 3) Move filter to a dimension “Partial Sum (Channel)”

Convolutional Layer: Sliding Window Operation



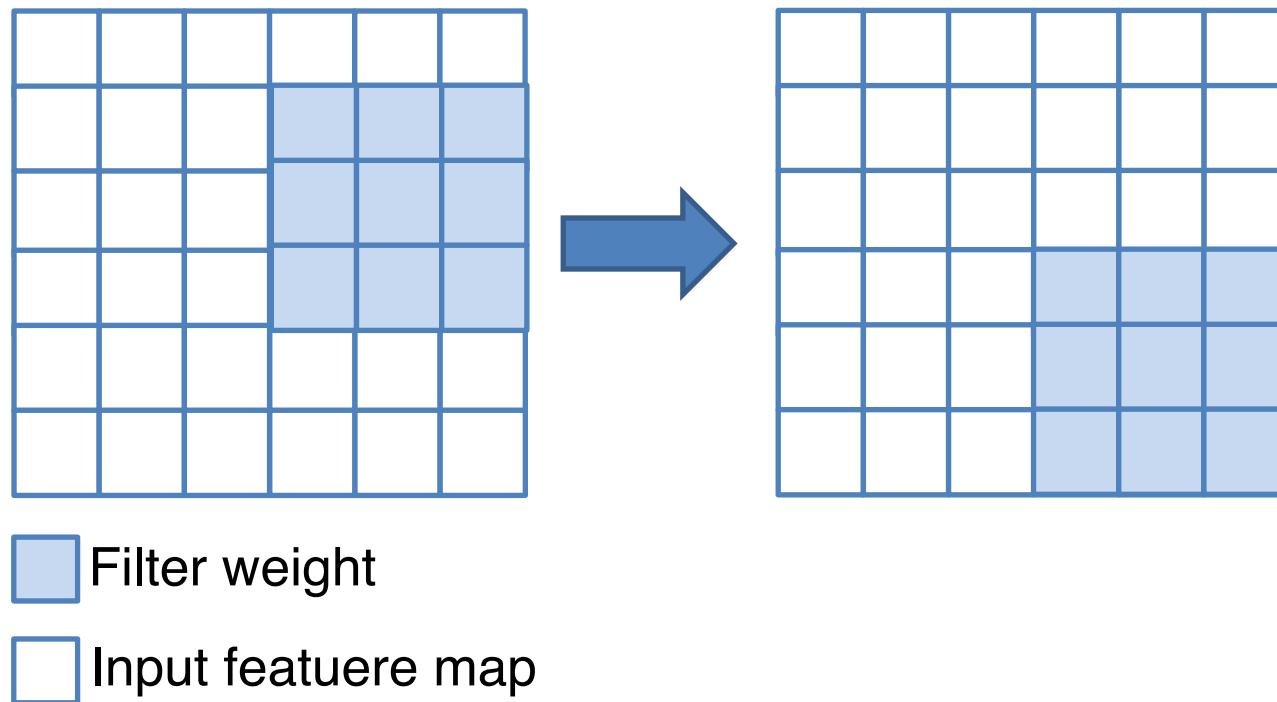
4) Repeat the same process (1-3) until the filter reaches the edge

Convolutional Layer: Sliding Window Operation



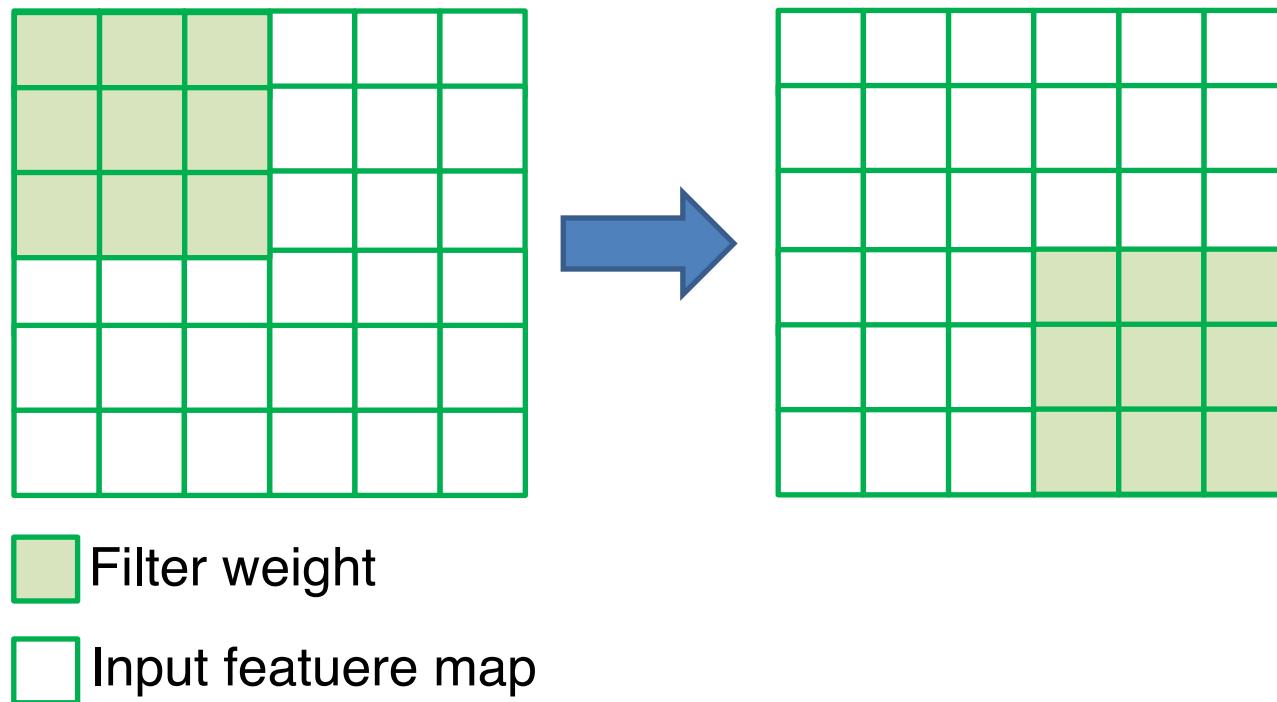
5) Move on to the next row and repeat the same process (1-4)

Convolutional Layer: Sliding Window Operation



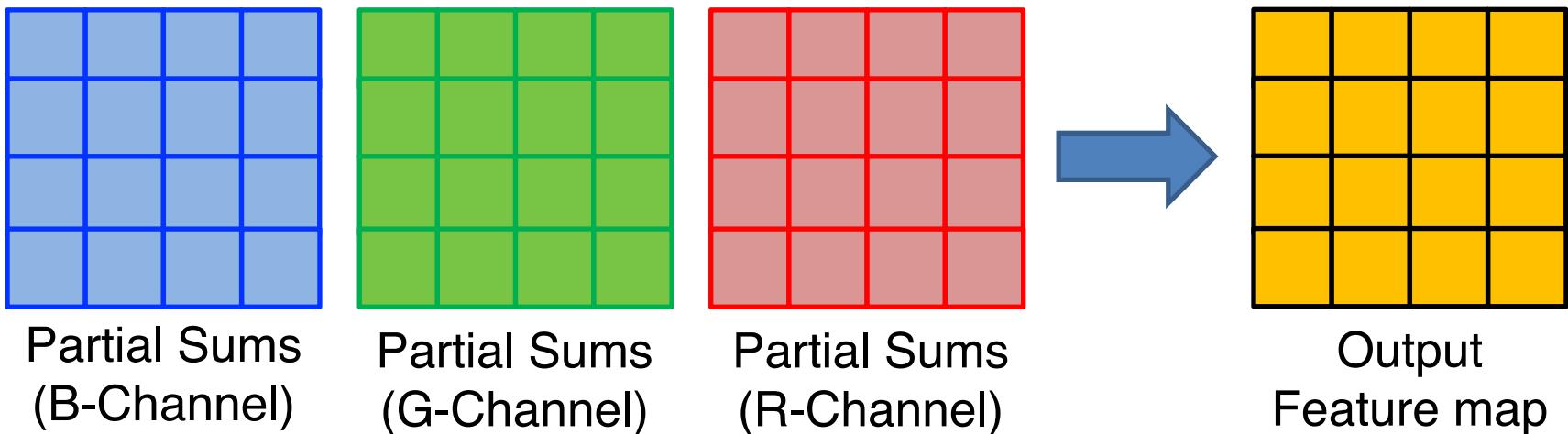
6) Repeat the same process (1-5) until the filter reaches the final pixel of input feature map

Convolutional Layer: Sliding Window Operation



7) Repeat the same process (1-6) for all the other channels

Convolutional Layer: Sliding Window Operation



8) Accumulate channel partial sums element-by-element to get output feature map

Convolutional Layer: Example

1	2	3
-2	0	-1
5	-2	4

0	1	0	1	0	1
2	4	3	1	0	0
5	2	7	2	1	5
4	1	8	4	2	8
5	0	1	5	8	3
0	0	0	5	2	6

**Multi-dimensional
2D Filters**

**Input featuremaps
(Input image)**

Convolutional Layer: Example

1	2	3
-2	0	-1
5	-2	4

0	1	0	1	0	1
2	4	3	1	0	0
5	2	7	2	1	5
4	1	8	4	2	8
5	0	1	5	8	3
0	0	0	5	2	6

Channel partial sum[0][0] =

$$\begin{aligned} & 1 \times 0 + 2 \times 1 + 3 \times 0 \\ & + (-2) \times 2 + 0 \times 4 + (-1) \times 3 \\ & + 5 \times 5 + (-2) \times 2 + 4 \times 7 \\ & = 44 \end{aligned}$$

44				

Convolutional Layer: Example

1	2	3
-2	0	-1
5	-2	4

0	1	0	1	0	1
2	4	3	1	0	0
5	2	7	2	1	5
4	1	8	4	2	8
5	0	1	5	8	3
0	0	0	5	2	6

Channel partial sum[0][1] =

$$\begin{aligned} & 1 \times 1 + 2 \times 0 + 3 \times 1 \\ & + (-2) \times 4 + 0 \times 3 + (-1) \times 1 \\ & + 5 \times 2 + (-2) \times 7 + 4 \times 2 \\ & = -1 \end{aligned}$$

44	-1			

Convolutional Layer: Example

44	-1	31	30
52	30	30	40
43	34	23	28
19	44	10	64

-4	5	6	2
7	3	2	5
0	-1	9	2
0	2	0	6

-9	3	7	1
6	2	8	1
3	-4	0	0
5	3	-2	-6

31	7	44	33
65	35	40	46
46	29	32	30
24	49	8	64

Decreased dimension?
(6x6 -> 4x4)

Output
Feature map

Convolutional Layer: Zero-padding

1	2	3
-2	0	-1
5	-2	4

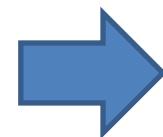
Filter

0	1	0	1	0	1		
2	4	3	1	0	0		
5	2	7	2	1	5		
4	1	8	4	2	8		
5	0	1	5	8	3		
0	0	0	5	2	6		

Zero-padded input feature map



Zero pixels



44	-1	31	30	-3	26
52	30	30	40	-10	30
43	34	23	28	41	4
19	44	10	64	10	-9
8	12	33	11	10	-9
0	15	15	27	14	6

Channel partial sum

Convolutional Layer: Computation

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
    for(m=0; m<M; m++) { // Weight Filters
        for(c=0; c<C; c++) { // IFMap/Weight Channels
            for(y=0; y<H; y++) { // Input feature map row
                for(x=0; x<W; x++) { // Input feature map column
                    for(j=0; j<R; j++) { // Weight filter row
                        for(i=0; i<R; i++) { // Weight filter column
                            O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```



Accumulation Multiplication

Massive independent multiplications
Massive accumulations



Massive parallelism!

Pooling Layer

- **Selecting Pixels using Pooling Window**

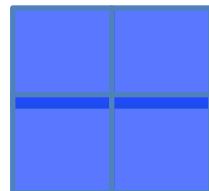
Ex) Max Pooling

31	7	44	33
65	35	40	46
46	29	32	30
24	49	8	64

Feature map



65	



Pooling Window

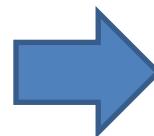
Pooling Layer

- **Selecting Pixels using Pooling Window**

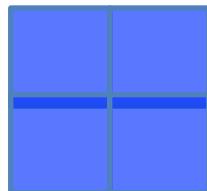
Ex) Max Pooling

31	7	44	33
65	35	40	46
46	29	32	30
24	49	8	64

Feature map



65	46



Pooling Window

Pooling Layer

- **Selecting Pixels using Pooling Window**

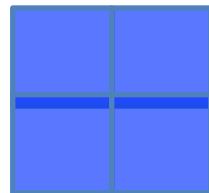
Ex) Max Pooling

31	7	44	33
65	35	40	46
46	29	32	30
24	49	8	64

Feature map



65	46
46	



Pooling Window

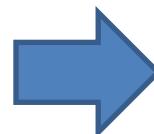
Pooling Layer

- Selecting Pixels using Pooling Window

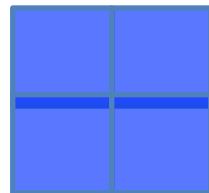
Ex) Max Pooling

31	7	44	33
65	35	40	46
46	29	32	30
24	49	8	64

Feature map



65	46
46	64

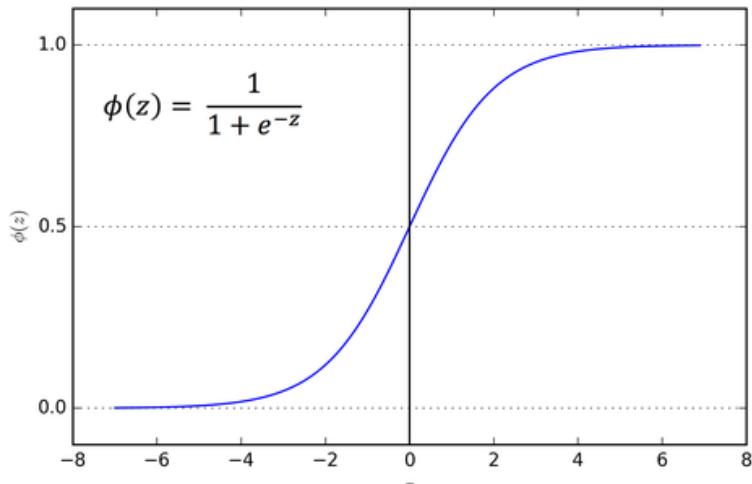


Pooling Window

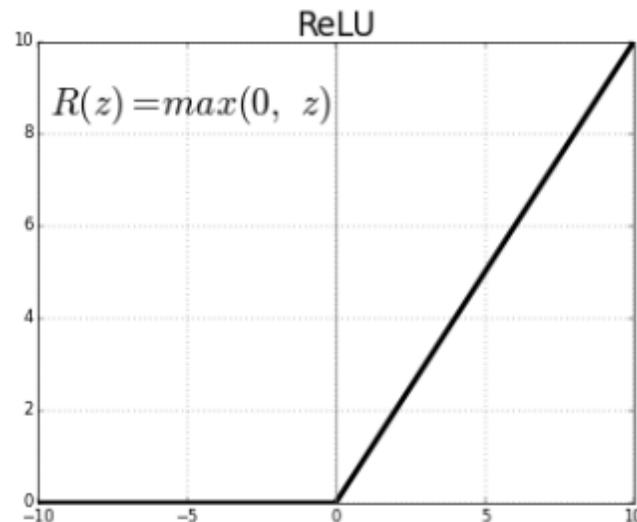
Reduces feature map dimension!

Activation Layer

- Applying a non-linear function



Sigmoid function

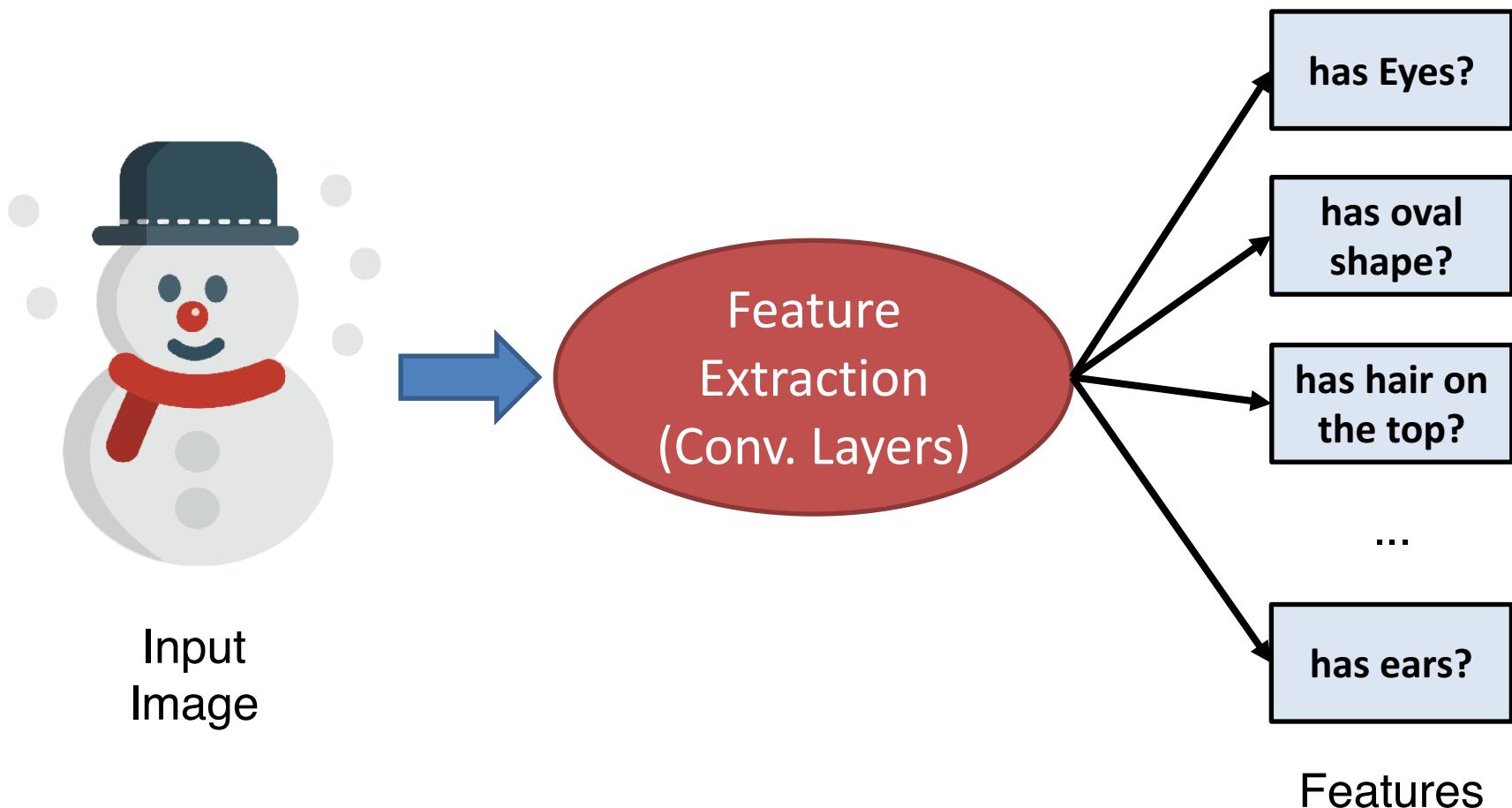


Rectified Linear Unit (ReLU) function

- Add non-linearity to neural networks
- Normalizes feature map values

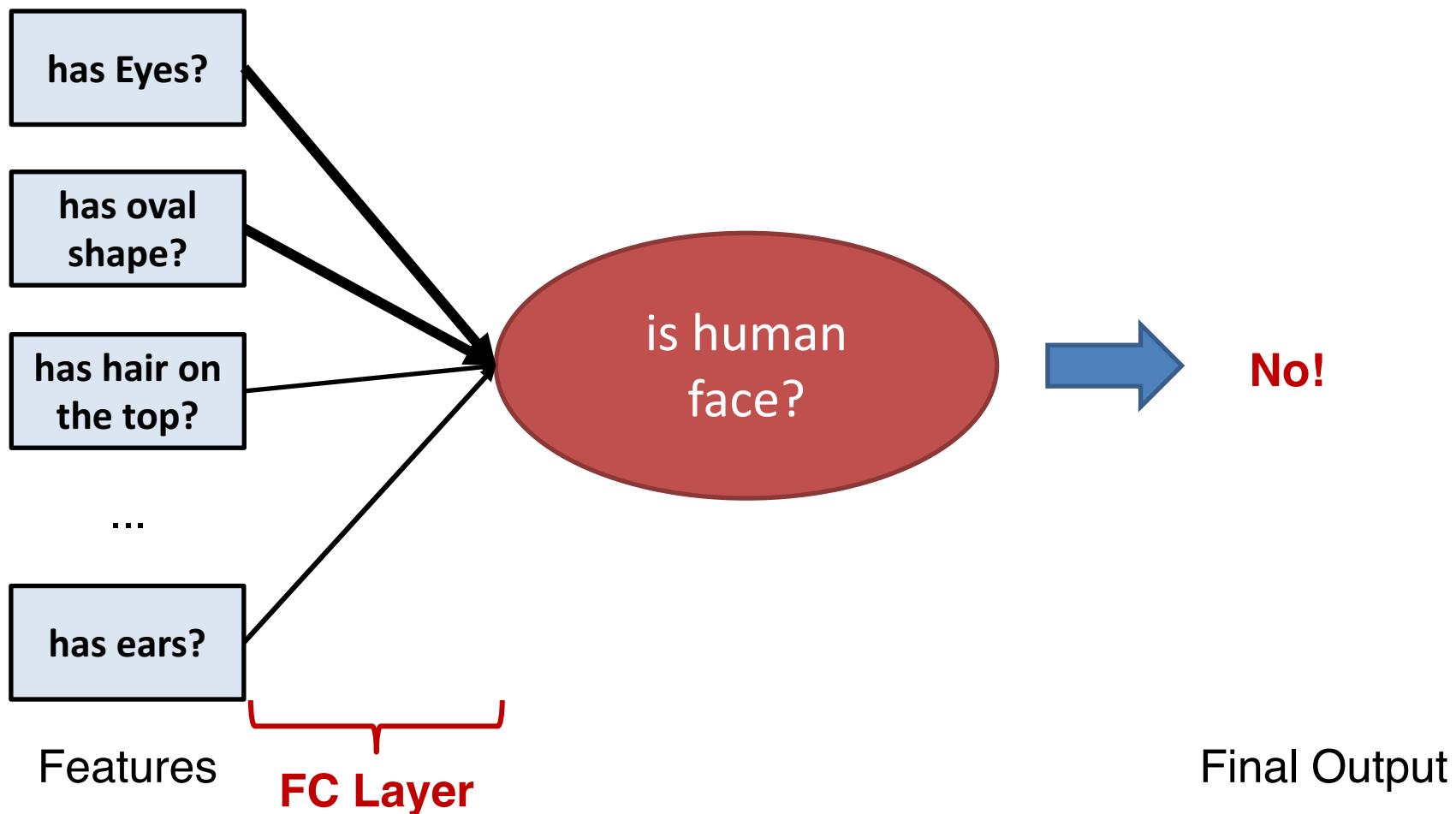
Fully-connected Layer

- Determining Output Using Gathered Features



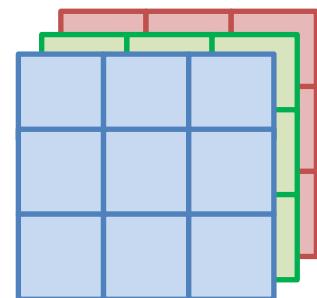
Fully-connected Layer

- Determining Output Using Gathered Features

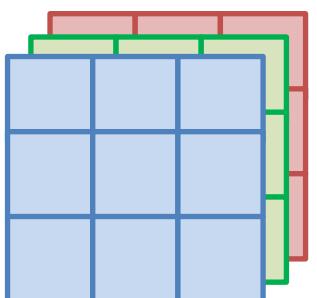


Fully-connected Layer: Computation

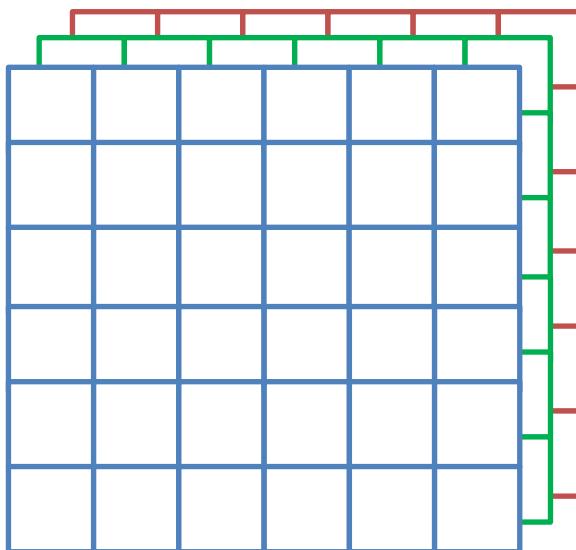
- **Convolutions with Multiple Filters**



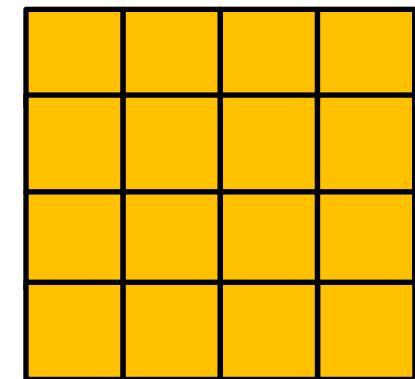
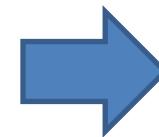
Filter 0



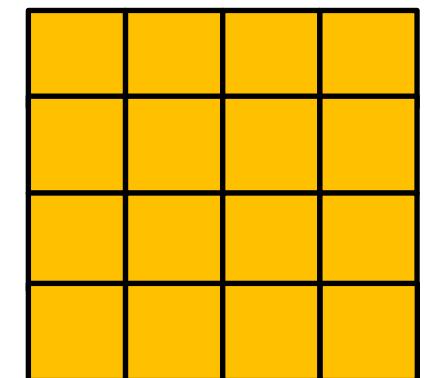
Filter 1



Input feature map



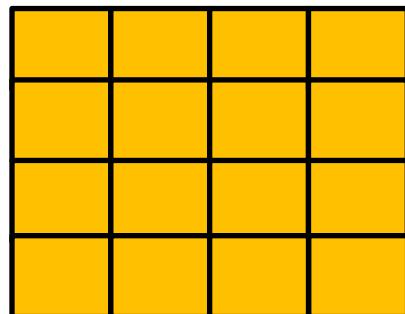
Output feature map 0



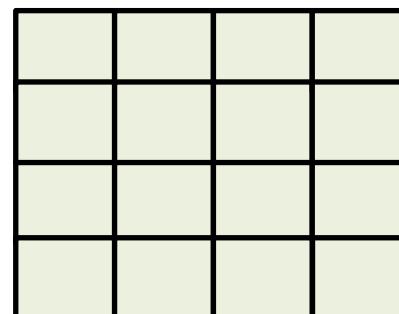
Output feature map 1

Fully-connected Layer: Computation

- Utilizing Each Feature to Determine Output



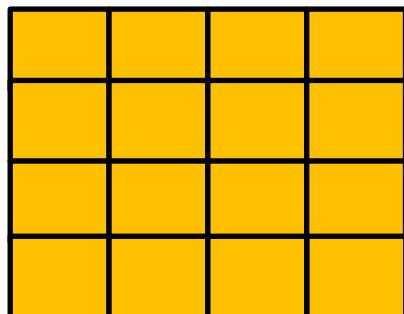
Output feature map 0



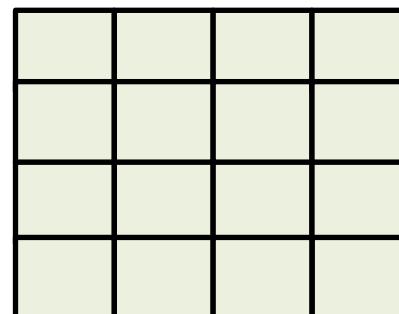
FC filter 0



Bias 0



Output feature map 1



FC filter 1



Bias 1



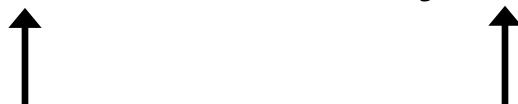
Final Value

Layers in CNN

- **Convolutional Layer**
 - Feature extraction
 - The most computation-dominant layer in CNNs
- **Pooling Layer**
 - Reduce the dimension of input/output feature map
- **Activation Layer**
 - Normalize input/output feature map values
- **Fully-connected Layer**

Revisiting Convolutional Layer

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
    for(m=0; m<M; m++) { // Weight Filters
        for(c=0; c<C; c++) { // IFMap/Weight Channels
            for(y=0; y<H; y++) { // Input feature map row
                for(x=0; x<W; x++) { // Input feature map column
                    for(j=0; j<R; j++) { // Weight filter row
                        for(i=0; i<R; i++) { // Weight filter column
                            O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```



Accumulation Multiplication

Massive parallelism!

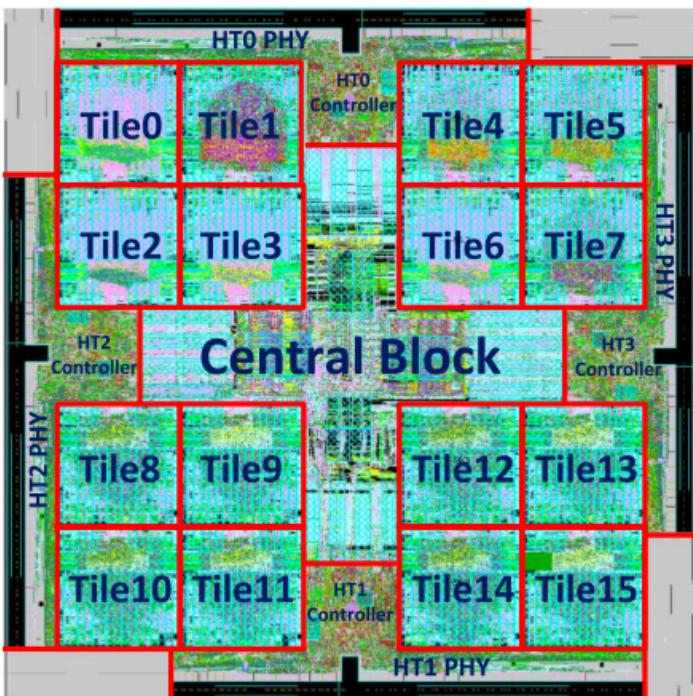


SIMD style parallel execution

Day 1 Agenda

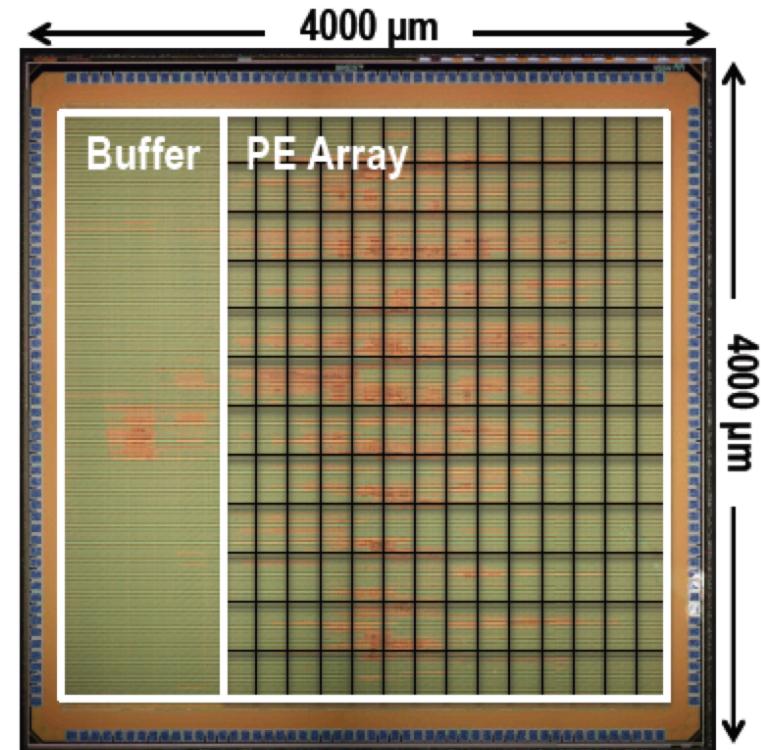
- **Convolutional Neural Networks (CNNs)**
 - Applications
 - CNN structure
 - Layer structure and computation
 - CNN accelerator structure overview
- **Bluespec System Verilog (BSV)**
 - BSV Overview
 - Basic Syntax
 - Combinational logic

CNN Accelerators



Dadiannao (MICRO 2014)

256 PEs (16 in each tile)



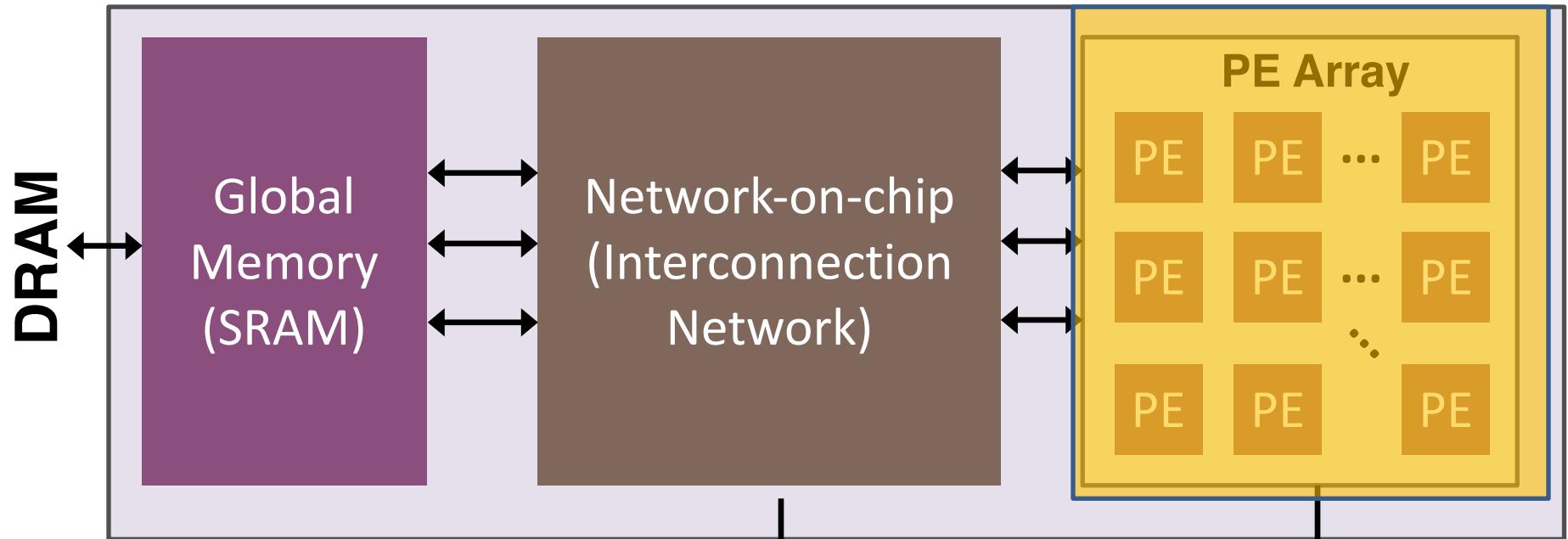
Eyeriss (ISCA 2016)

168 PEs

*PE: processing element

Spatial CNN Accelerator Structure

Focus of lab assignments



Multi-Bus: Eyeriss

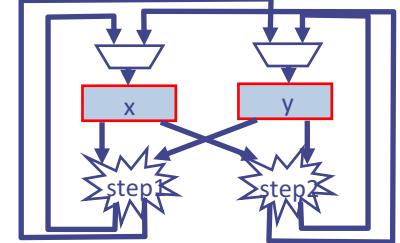
Mesh: Diannao, Dadiannao

Crossbar+Mesh: TrueNorth

Spatial processing over PEs

Day 1 Agenda

- **Convolutional Neural Networks (CNNs)**
 - Applications
 - CNN structure
 - Layer structure and computation
 - CNN accelerator structure overview
- **Bluespec System Verilog (BSV)**
 - BSV Overview
 - Basic Syntax
 - Combinational logic



Structure of BSV Code

```
module mkXYZ (XYZ_Interface);
```

```
Reg#(Int#(32)) x <- mkRegU;
```

```
Reg#(Int#(32)) y <- mkReg(0);
```

State

```
rule step1 ((x > y) && (y != 0));
```

```
    x <= y; y <= x;
```

```
endrule
```

```
rule step2 (( x <= y) && (y != 0));
```

```
    y <= y-x;
```

```
endrule
```

Internal behavior

```
method Action start(Int#(32) a, Int#(32) b) if (y==0);
```

```
    x <= a; y <= b;
```

```
endmethod
```

```
method Int#(32) result() if (y==0);
```

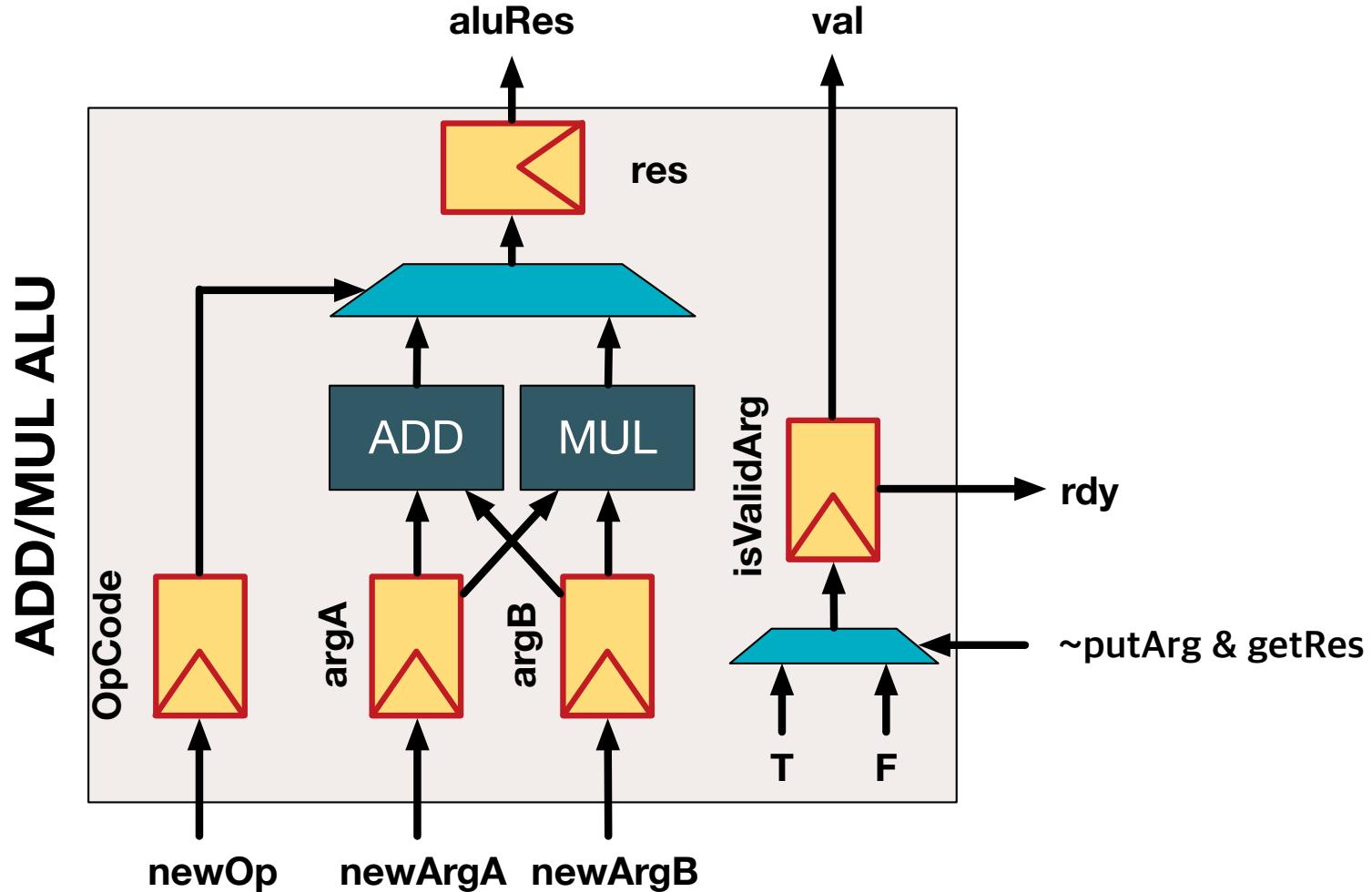
```
    return x;
```

```
endmethod
```

External interface

```
endmodule
```

Bluespec System Verilog Example (ALU)



Bluespec System Verilog Example (ALU)

```
typedef Bit#(32) Word;  
typedef enum {ADD, MUL} OpCode deriving (Bits, Eq);
```

User-defined types

```
interface ALU:  
    method Action putArguments(OpCode newOp,  
                               Word newArgA, Word newArgB);  
    method ActionValue#(Word) getResults;  
endinterface
```

Module interface definition

...

Only defines module interface signature
(name, return type, and arguments)

Bluespec System Verilog Example (ALU)

```
(* synthesize *)
```

← **Synthesis Boundary**

```
module mkALU(ALU);
  Reg#(Bool) isValidArgs <- mkReg(False);
  Reg#(OpCode) op <- mkRegU;
  Reg#(Word) argA <- mkRegU;
  Reg#(Word) argB <- mkRegU;
  Reg#(Word) res <- mkRegU;
```

} **Sub-modules (registers)**

```
rule doOperation (isValidArgs == True);
  if(op == ADD)
    res <= argA + argB;
  else
    res <= argA * argB;
endrule
```

} **Rule: define an atomic action**

```
...
```

Bluespec System Verilog Example (ALU)

```
method Action putArguments(OpCode newOp,  
Word newArgA, Word newArgB) if (isValidArgs == False);  
    isValidArgs <= True; op <= newOp;  
    argA <= newArgA; argB <= newArgB;  
endmethod
```

```
method ActionValue#(Word)  
getResults if (isValidArgs == True);  
isValidArgs <= False;  
return res;  
endmethod
```

```
endmodule
```

Actual interface implementation

Day 1 Agenda

- **Convolutional Neural Networks (CNNs)**
 - Applications
 - CNN structure
 - Layer structure and computation
 - CNN accelerator structure overview
- **Bluespec System Verilog (BSV)**
 - BSV Overview
 - Basic Syntax
 - Combinational logic
 - Sequential logic

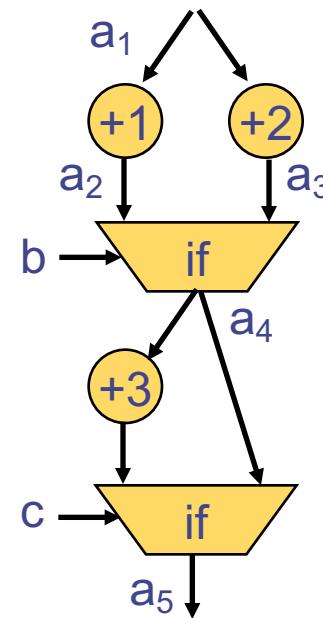
BSV Basic Syntax

- **Variable Declaration and Initialization**
 - Value assignment ('=')
 - Types
 - Type deduction ('let' statement)
 - Type value and real value
- **Calculating values (combinational logic)**
 - Conditional Statements
 - Arithmetic operations
 - Logical operations
 - Bit operators

Variable Assignment ('=')

An example involving conditionals

```
int a = 10;  
  
if (b) a = a + 1;  
else a = a + 2;  
  
if (c) a = a + 3;
```



Executes sequentially within a cycle

Variable Declaration and Initialization

- **Types in BSV**
 - Primitive types
 - **Bit#(Number of bits)** // All the signals on the circuit
 - **Bool** // Boolean value
 - Aggregation types
 - **Enum**
 - **Struct**
 - **Vector#(Number of elements, Type)**
 - Module interface

Variable Declaration and Initialization

- Types in BSV
 - Example

```
rule runExample;  
    Bit#(32) valA = 0;  
    Bit#(32) valB = 15;  
    Bool isValid = True; ] }  
Declaration of primitive types  
Corresponds to “wire” in Verilog
```

```
if(isValid == True)  
    $display("Value A = %d", valA);  
endrule
```

Variable Declaration and Initialization

- **Types in BSV**
 - Example - enumeration

```
typedef enum{Mon, Tue, Wed, Thr, Fri, Sat, Sun}  
Days deriving (Bits, Eq);
```



- 1) Type “Days” will be converted to “Bits” internally
- 2) Comparison among “Days” values will be available

Variable Declaration and Initialization

- **Types in BSV**
 - Example - struct

```
typedef struct{  
    Days day;  
    Bit#(32) value;  
} DailyBudget deriving (Bit, Eq);
```

```
DailyBudget budget1;  
budget1.day = Mon;  
budget1.value = 15000;
```



Access fields using ‘.’

Variable Declaration and Initialization

- **Types in BSV**
 - Example – vector1

```
Vector#(4, Bit#(32)) fourValues;
```

```
fourValues[0] = 1;  
fourValues[1] = 2;  
fourValues[2] = 3;  
fourValues[3] = 4;
```



Access each element using []

Variable Declaration and Initialization

- Types in BSV
 - Example – vector2

```
Vector#(2, Vector#(4, Bit#(32))) twoFourValues;
```

```
fourValues[0][0] = 1;  
fourValues[0][1] = 2;  
fourValues[0][2] = 3;  
fourValues[0][3] = 4;
```



Access multiple '[]' s to access elements

Variable Declaration and Initialization

- **Automatic Type Deduction using “let”**
 - “let” statement enables users to declare a variable without providing an exact type
 - Compiler deduces the type using other information (e.g., assigned value)
 - Example
- `let isValid = True; // Assigning Bool value; isValid is Bool`
- `let today = Fri; // Assigning Days value; today is Days`

Variable Declaration and Initialization

- **Type value and real value**
 - Integer literal assigned to a type is **a type value**
(e.g., **typedef** 32 WordLength;)
 - All the values based on **Bit#(n)**, which actually exists on the circuit as signals, are real values.
 - We cannot directly assign a type value to a real value
(e.g., **Bit#(32) len = WordLength; //Error!**)

Variable Declaration and Initialization

- **Type value and real value**
 - Type values are usually used as module/interface parameters
(e.g., **Reg#(Bit#(WordLength)**) wordReg <- mkRegU;)
 - We cannot directly assign a type value to a real value
(e.g., **Bit#(32) len = WordLength;** //Error!)
 - We can convert (1) type values to Integer values and
(2) Integer values to real values
(e.g., **Bit#(32) len = fromInteger(valueOf(WordLength));**
Integer type will be explained with “static elaboration”

Variable Declaration and Initialization

- **Type value and real value**



Module parameters

- Cannot be modified after defined
- Example: data bit-width, number of PEs, etc.
- To define another type value using existing type values, use special statements (e.g., TAdd#(T1, T2))

Conceptual numbers in a circuit (not a signal)

- Example: The index of a register array, iteration variable in a for-loop

Real Values in a circuit

- Represents values that exist either on a wire or memory element(register/FIFO)

BSV Basic Syntax

- **Variable Declaration and Initialization**
 - Value assignment ('=')
 - Types
 - Type deduction ('let' statement)
 - Type value and real value
- **Calculating values (combinational logic)**
 - Conditional Statements
 - Arithmetic operations
 - Logical operations
 - Bit operators

IF-statement

- If/elseif/ else/ endif

- Ex)

```
Bit#(16) valA = 12;  
if (valA == 0) begin  
    $display("valA is zero");  
end  
else if(          valA != 1) begin  
    $display("valA is neither zero nor one");  
end  
else begin  
    $display("valA is %d", valA);  
end
```

Arithmetic Operators

- Addition (+), subtraction (-), multiplication (*), and divisions (/)

– Ex)

```
Bit#(16) valA = 12; Bit#(16) valB = 2500;
```

```
Bit#(16) valC = 50000;
```

```
let valD = valA + valB;
```

```
let valE = valC - valB;
```

```
let valF = valB * valC;
```

```
let valG = valB / valA;
```

Logical Operators

- Comparators (`==`, `>`, `<`, `>=`, `<=`) and Operators (`&&`, `||`, `!`)
 - Ex)

```
Bit#(16) valA = 12; Bit#(16) valB = 2500;
```

```
Bit#(16) valC = 50000;
```

```
let valD = valA < valB;
```

```
let valE = valC == valB;
```

```
let valF = !valD;
```

```
let valG = valD && !valE;
```

Bit Operators

- Selection ([]), concatenation ({ }), truncation (truncate, truncateLSB), and extension (zeroExtend, signExtend)

– Ex)

```
Bit#(4) valA = 4'b1001; Bit#(4) valB = 4'b1100;  
let valC = {valA, valB};
```

```
Bit#(4) valD = truncate(valC);
```

“Let” statement doesn’t work with truncate/extension. Why?



Now we are ready to code combinational logic 😊

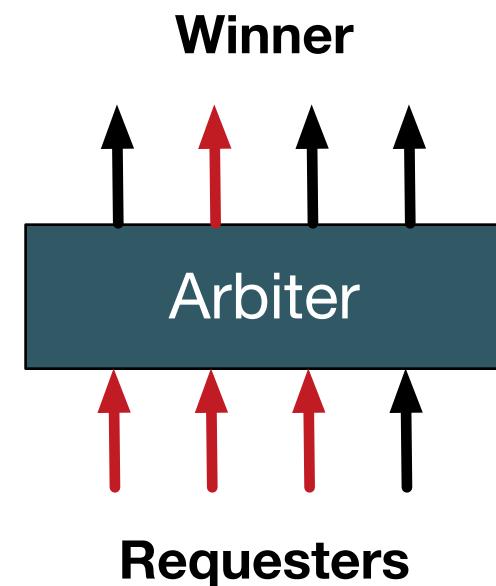
Day 1 Agenda

- **Convolutional Neural Networks (CNNs)**
 - Applications
 - CNN structure
 - Layer structure and computation
 - CNN accelerator structure overview
- **Bluespec System Verilog (BSV)**
 - BSV Overview
 - Basic Syntax
 - Combinational logic

[Lab] Combinational Logic Implementation

- **Arbiter**

- Arbiter is a hardware module that manages concurrent requests to one hardware resource (e.g., DRAM write requests to the same DRAM bank from multiple cores)



[Lab] Combinational Logic Implementation

- **Arbiter**

- Requirements on Arbiter
 - Low area and power
 - Scalability in area/power and performance (latency, max. clock cycle, etc.)
 - Fairness
- Arbiter designs
 - Priority arbiter
 - Round-robin arbiter
 - Matrix arbiter
 - ...

[Lab] Combinational Logic Implementation

- **Priority Arbiter**
 - **Arbitration policy**
 - Determine priority of each requester in design time
(e.g., priority: core 1 < core 2 < core 3 < core 4
when the arbiter receives requests from
core 1,3, and 4, the arbiter select core 1
as the winner)
 - Implement the priority using a combinational logic

[Lab] Combinational Logic Implementation

- Priority Arbiter
 - Spec
 - Implement a 4:1 priority arbiter logic using the following priority
 - Request 0 < Request 1 < Request 2 < Request 3
 - * A < B: A has higher priority than B
 - Module interface (I/O) is provided in the skeleton code
 - Hint: you can use **\$display**(" your print-out message") for debugging