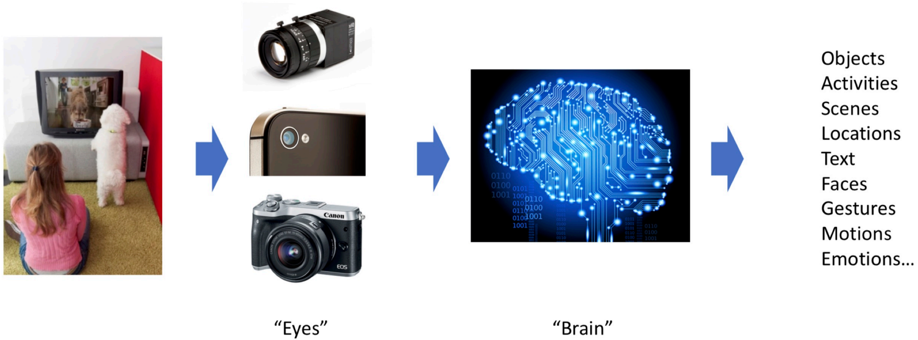
香港中文大學
The Chinese University of Hong Kong

# Hardware Friendly Computer Vision

Bei Yu
Department of Computer Science & Engineering
Chinese University of Hong Kong
byu@cse.cuhk.edu.hk

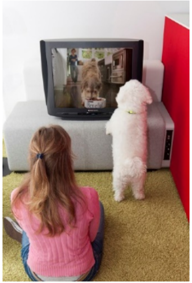
November 15, 2022

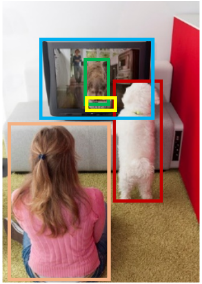
- Humans use their eyes and their brains to visually sense the world.
- Computers user their cameras and computation to visually sense the world

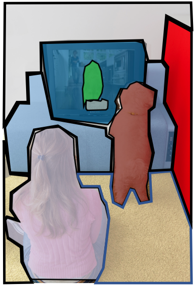



Jian Sun, “Introduction to Computer Vision and Deep Learning”.

# Few More Core Problems



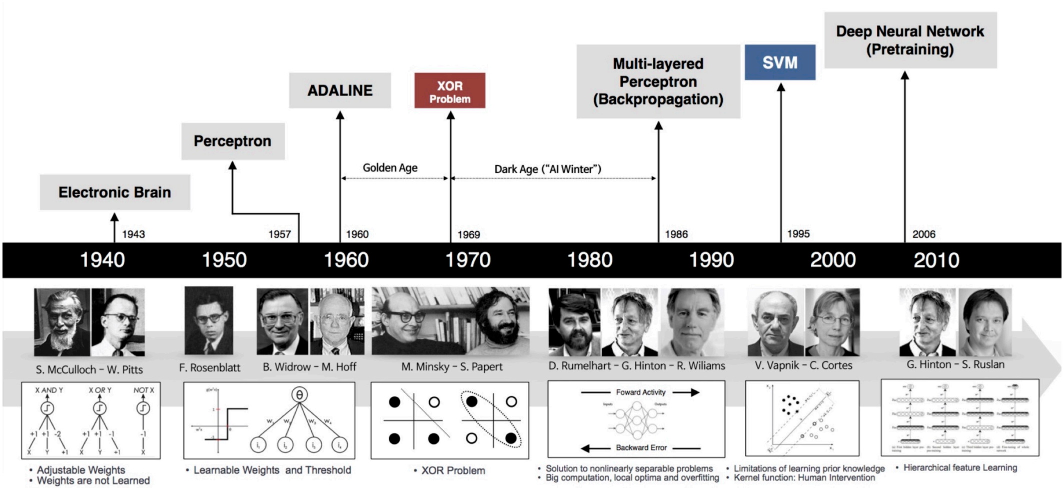




Classification | Detection | Segmentation | Sequence

Image → Region → Pixel | Video

# A Bit of History



# Winter of Neural Networks (mid 90′ – 2006)

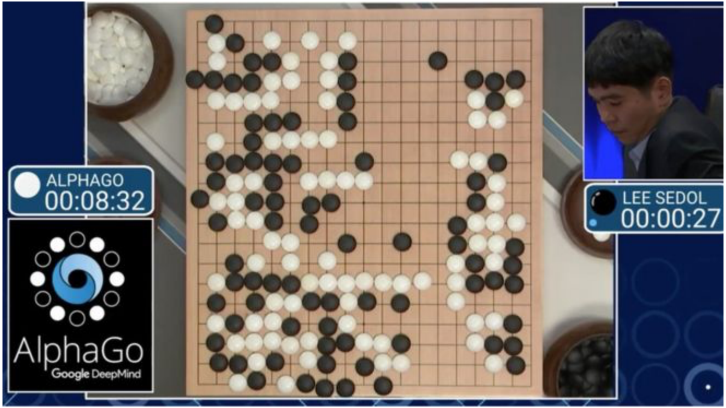
- The rises of SVM, Random forest
- No theory to play
- Lack of training data
- Benchmark is insensitive
- Difficulties in optimization
- Hard to reproduce results

**Curse**

"Deep neural networks are no good and could never be trained."

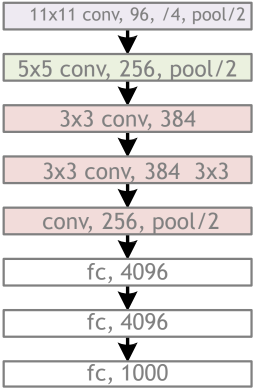
# Renaissance of Deep Learning (2006 – )

- A fast learning algorithm for deep belief nets. [Hinton et.al 1996]
- Data + Computing + Industry Competition
- NVidia's GPU, Google Brain (16,000 CPUs)
- Speech: Microsoft [2010], Google [2011], IBM
- Image: AlexNet, 8 layers [Krizhevsky et.al 2012] (26.2% -> 15.3%)



# Revolution of Depth

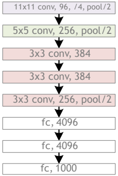
AlexNet, 8 layers (ILSVRC 2012)



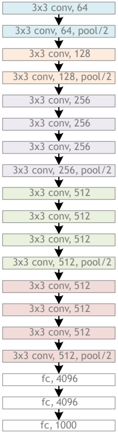

Slide Credit: He et al. (MSRA)

# Revolution of Depth

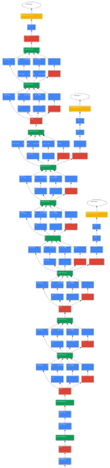
AlexNet, 8 layers (ILSVRC 2012)




VGG, 19 layers (ILSVRC 2014)




GoogleNet, 22 layers (ILSVRC 2014)



Slide Credit: He et al. (MSRA)

# Revolution of Depth

AlexNet, 8 layers (ILSVRC 2012)

VGG, 19 layers (ILSVRC 2014)

ResNet, 152 layers (ILSVRC 2015)

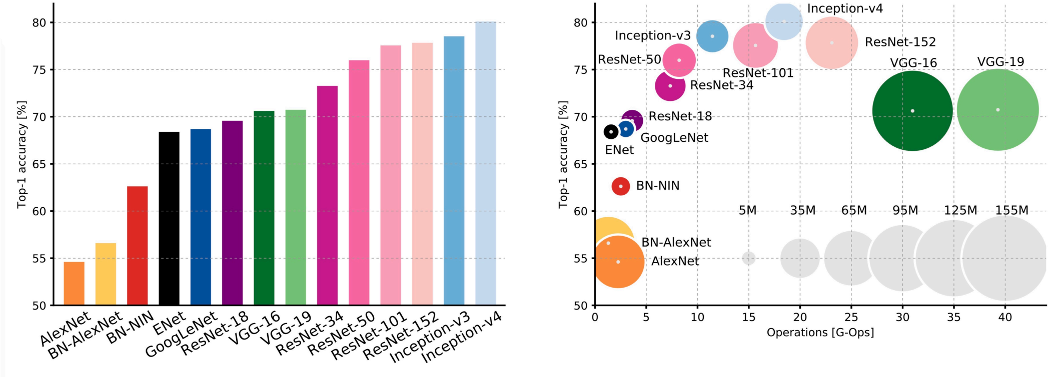
Slide Credit: He et al. (MSRA)

# Some Recent Classification Architectures

- AlexNet (Krizhevsky, Sutskever, and E. Hinton 2012) 233MB
- Network in Network (Lin, Q. Chen, and Yan 2013) 29MB
- VGG (Simonyan and Zisserman 2015) 549MB
- GoogleNet (Szegedy, Liu, et al. 2015) 51MB
- ResNet (K. He et al. 2016) 215MB
- Inception-ResNet (Szegedy, Vanhoucke, et al. 2016)
- DenseNet (Huang et al. 2017)
- Xception (Chollet 2017)
- MobileNetV2 (Sandler et al. 2018)
- ShuffleNet (Zhang, Zhou, et al. 2018)

# Some Recent Classification Architectures

- AlexNet (Krizhevsky, Sutskever, and E. Hinton 2012) 233MB
- Network in Network (Lin, Q. Chen, and Yan 2013) 29MB
- VGG (Simonyan and Zisserman 2015) 549MB
- GoogleNet (Szegedy, Liu, et al. 2015) 51MB
- ResNet (K. He et al. 2016) 215MB
- Inception-ResNet (Szegedy, Vanhoucke, et al. 2016) 23MB
- DenseNet (Huang et al. 2017) 80MB
- Xception (Chollet 2017) 22MB
- MobileNetV2 (Sandler et al. 2018) 14MB
- ShuffleNet (Zhang, Zhou, et al. 2018) 22MB



[1]

[1] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello (2017). "An analysis of deep neural network models for practical applications". In: *arXiv preprint*.

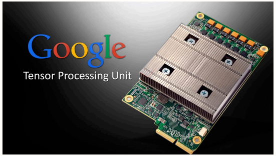
# When Machine Learning Meets Hardware

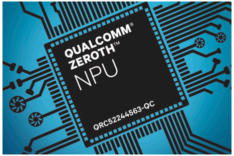
Convolution layer is one of the most expensive layers

- Computation pattern
- Emerging challenges

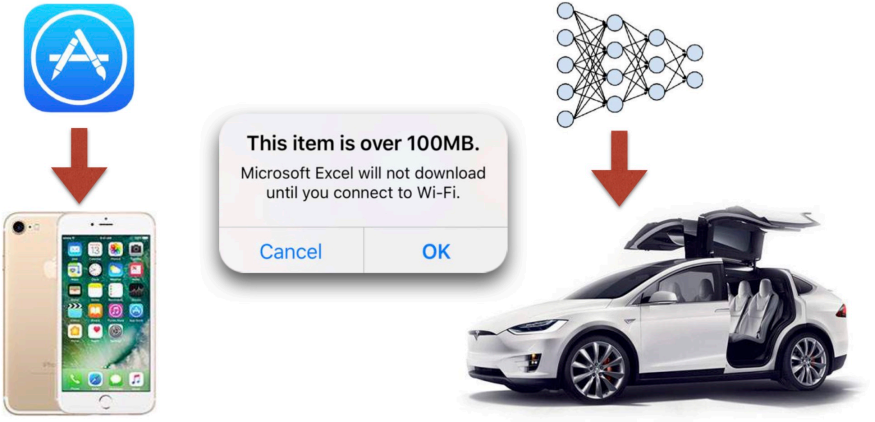
## More and more end-point devices with limited memory

- Cameras
- Smartphone
- Autonomous driving



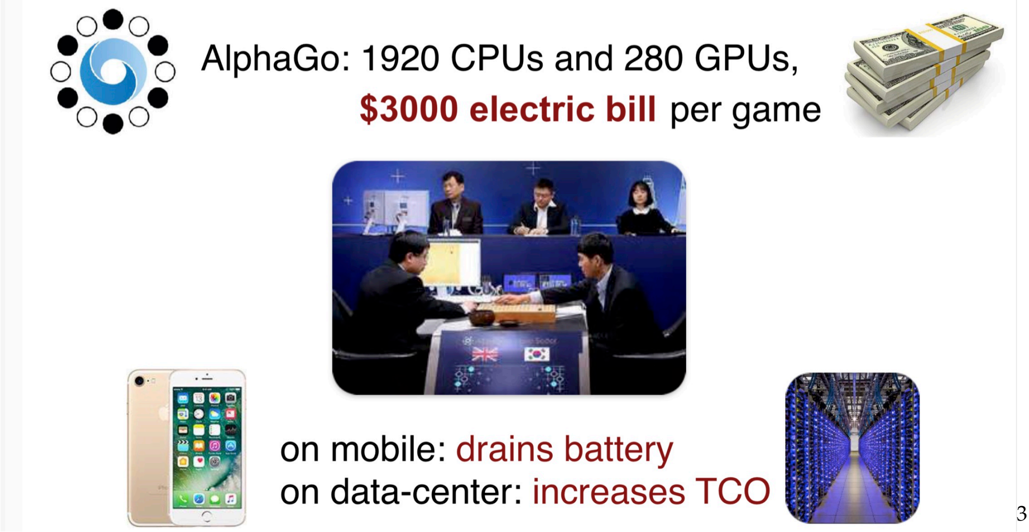
Hard to distribute large models through over-the-air update




[2]

[2]Song Han and William J. Dally (2018). “Bandwidth-efficient Deep Learning”. In: *Proc. DAC*, 147:1–147:6.

# 2nd Challenge: Energy Efficiency



[3]

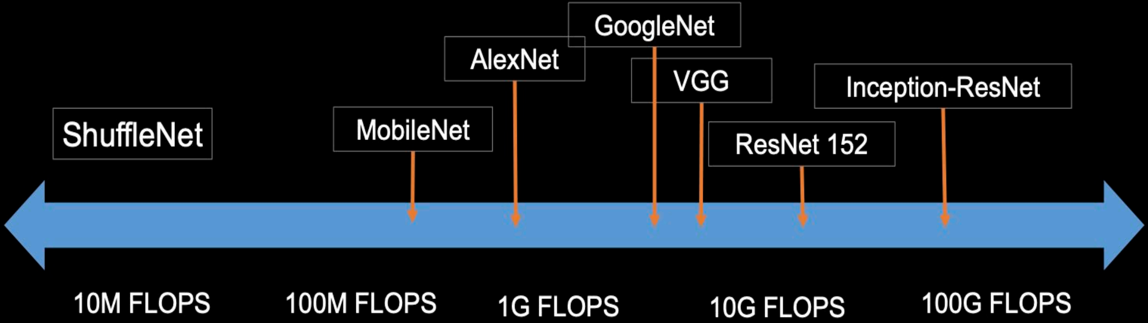
[3]Song Han and William J. Dally (2018). "Bandwidth-efficient Deep Learning". In: *Proc. DAC*, 147:1–147:6.

# Application Category

| Both | Datacenter | Edge |
|---|---|---|
| Intel, Nvidia, IBM, Xilinx, HiSilicon, Google, Baidu, Alibaba Group, Cambricon, DeePhi, Bitmain, Wave Computing | AMD, Microsoft, Apple, Tencent Cloud,Aliyun, Baidu Cloud, HUAWEI Cloud, Fujitsu, Nokia, Facebook, HPE, Thinkforce, Cerebras, Graphcore, Groq, SambaNova Systems, Adapteva, PEZY | Qualcomm, Samsung, STMicroelectronics, NXP, MediaTek, Rockchip, Amazon_AWS, ARM, Synopsys, Imagination, CEVA, Cadence, VeriSilicon, Videantis, Horizon Robotics, Chipintelli, Unisound, AISpeech, Rokid, KnuEdge, Tenstorrent, ThinCI, Koniku, Knowm, Mythic, Kalray, BrainChip, AImotive, DeepScale, Leepmind, Krtkl, NovuMind, REM, TERADEEP, DEEP VISION, KAIST DNPU, Kneron, Esperanto Technologies, Gyrfalcon Technology, GreenWaves Technology, Lightelligence, Lightmatter, ThinkSilicon, Innogrit, Kortiq, Hailo,Tachyum |

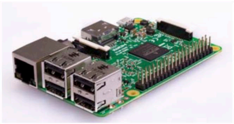
Source: `https://basicmi.github.io/Deep-Learning-Processor-List/`

# Computing Spectrum

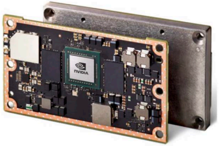




HERO3



CPU
(Raspberry Pi3)



GPU
(Jetson TX2)

FPGA
(UltraZed)

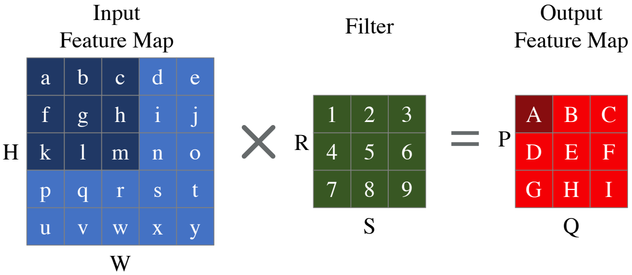

ASIC
(Movidius)

Flexibility

Power/Performance
Efficiency

# Outline

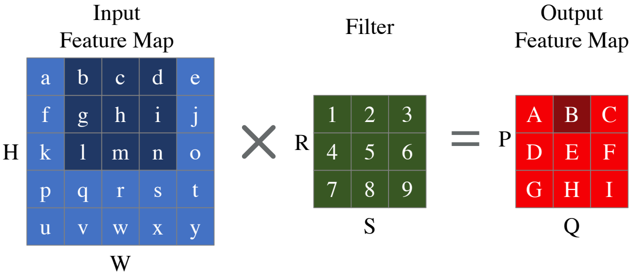
# Convolution Basis




$$
\begin{aligned}
A = & \, a \cdot 1 + b \cdot 2 + c \cdot 3 \\
& + f \cdot 4 + g \cdot 5 + h \cdot 6 \\
& + k \cdot 7 + l \cdot 8 + m \cdot 9
\end{aligned}
$$

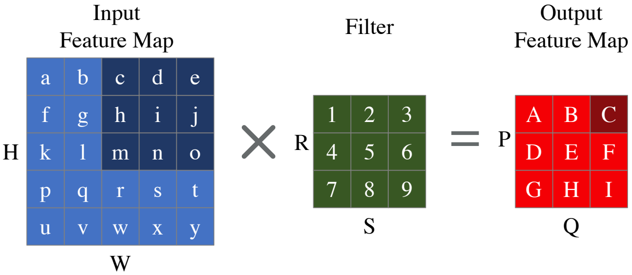
- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map

# 2D-Convolution



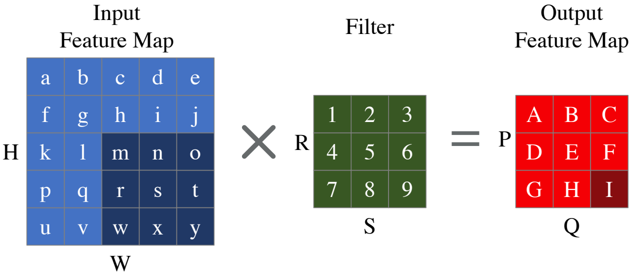

- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step

# 2D-Convolution



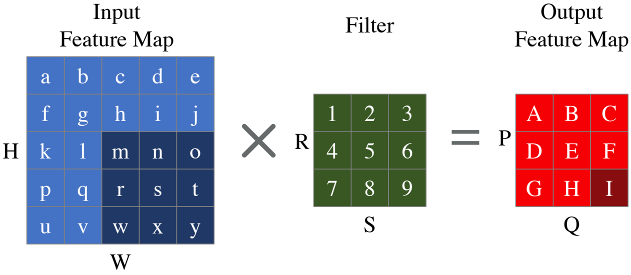

- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step

# 2D-Convolution



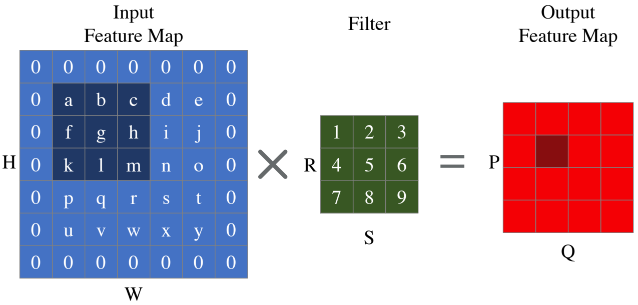

- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step

# 2D-Convolution




$$P = \frac{(H - R)}{\text{stride}} + 1;$$
$$Q = \frac{(W - S)}{\text{stride}} + 1.$$

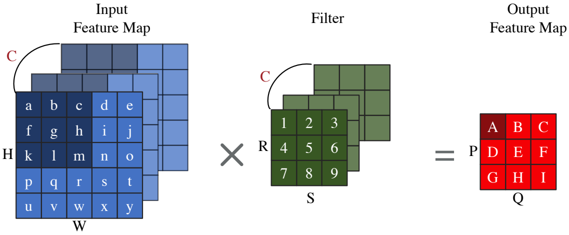
- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step



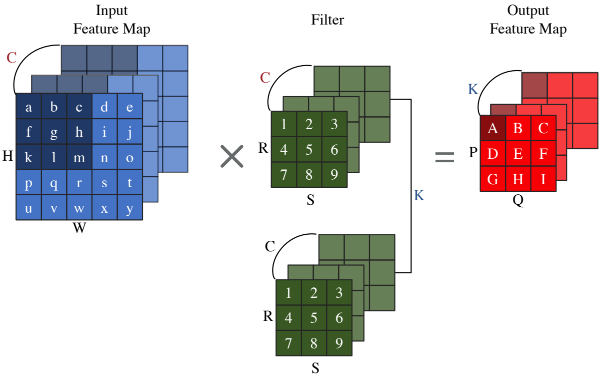

$$P = \frac{(H - R + 2 \cdot \text{pad})}{\text{stride}} + 1;$$
$$Q = \frac{(W - S + 2 \cdot \text{pad})}{\text{stride}} + 1.$$

- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step
- padding: # of zero rows/columns added

# 3D-Convolution

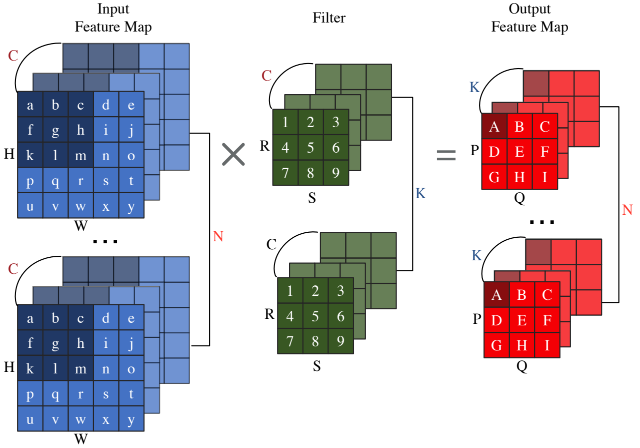



- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step
- padding: # of zero rows/columns added
- C: # of input channels




- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step
- padding: # of zero rows/columns added
- C: # of input channels
- K: # of output channels

# 3D-Convolution




- H: Height of input feature map
- W: Width of input feature map
- R: Height of filter
- S: Width of filter
- P: Height of output feature map
- Q: Width of output feature map
- stride: # of rows/columns traversed per step
- padding: # of zero rows/columns added
- C: # of input channels
- K: # of output channels
- N: Batch size

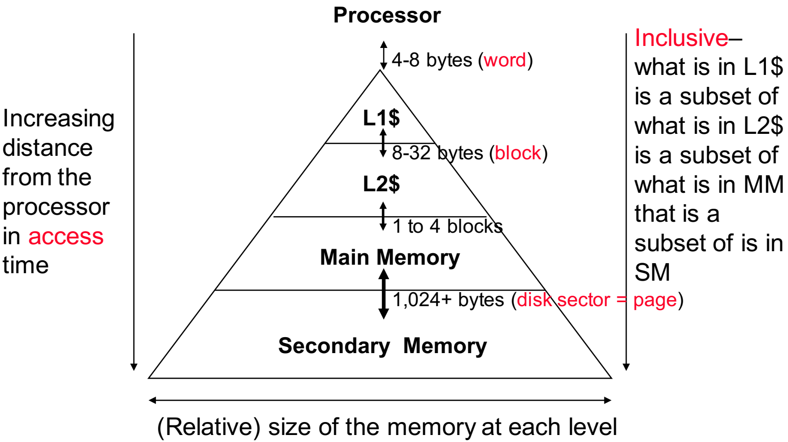
# Convolution 101

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 1 & 1 & 2 & 0 \\ 0 & 2 & 0 & 1 & 1 & 0 & 0 \\ 0 & 2 & 0 & 1 & 2 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 4 & 6 & 3 & 5 & 4 \\ 2 & 6 & 2 & 4 & 4 \\ 1 & 5 & 3 & 4 & 4 \\ 2 & 4 & 3 & 3 & 4 \\ 0 & 2 & 2 & 4 & 3 \end{bmatrix}$$

Direct convolution: No extra memory overhead

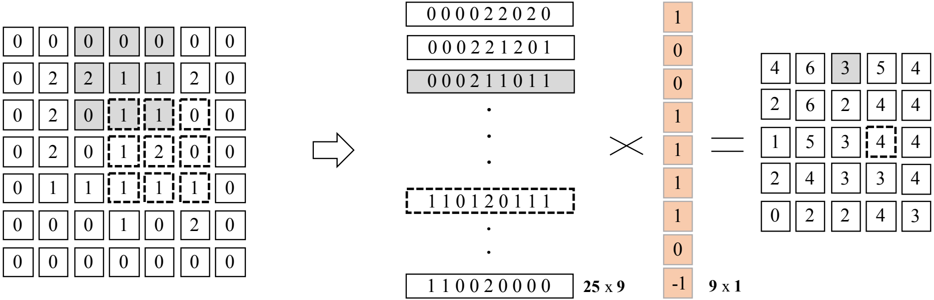
- Low performance
- Poor memory access pattern due to geometry-specific constraint
- Relatively short dot product

# Background: Memory System

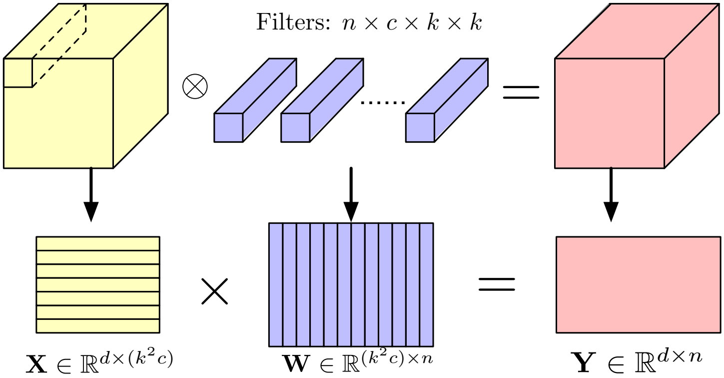



- Spatial locality
- Temporal Locality

# `Im2col` (Image2Column) 2D-Convolution



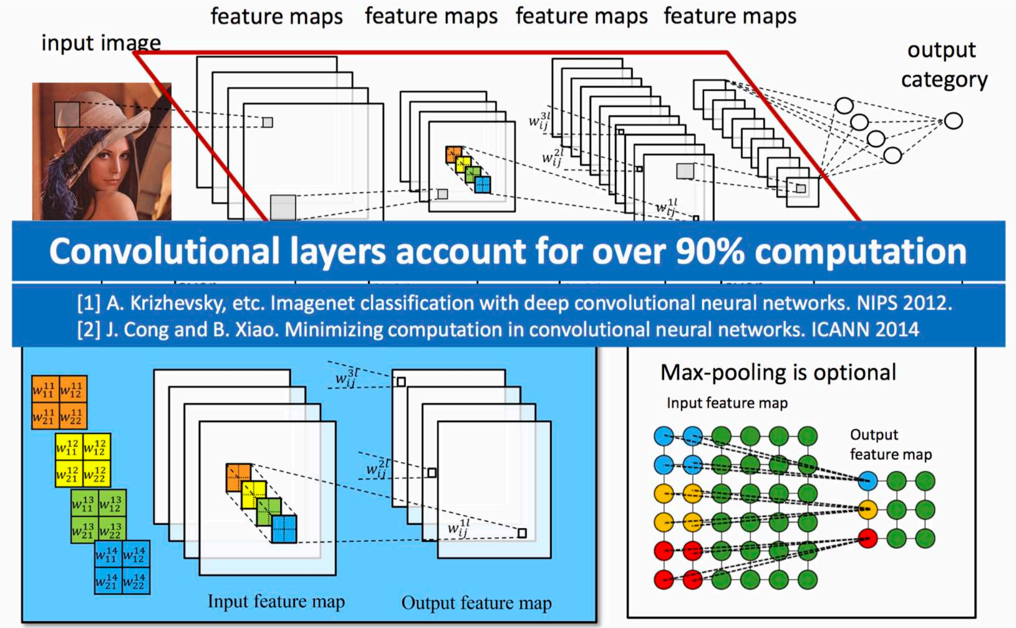

- Large extra memory overhead
- Good performance
- BLAS-friendly memory layout to enjoy SIMD/locality/parallelism
- Applicable for any convolution configuration on any platform



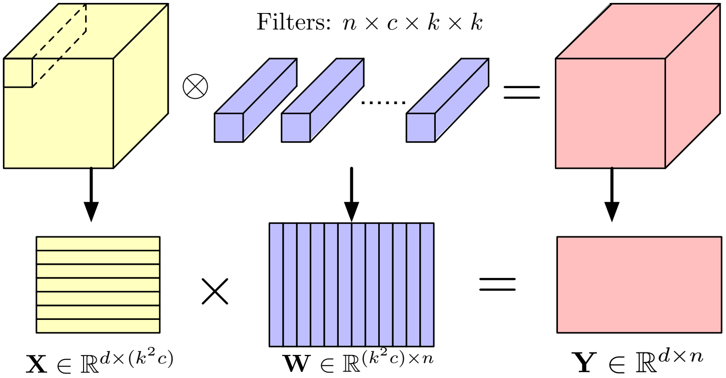

- Transform convolution to matrix multiplication
- Unified calculation for both convolution and fully-connected layers

# Algorithm Design Level

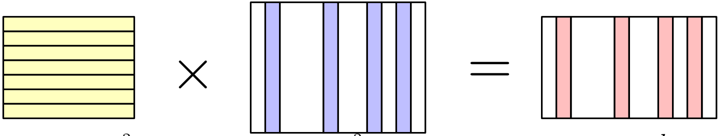
# Convolution Is the Bottleneck



# Im2col (Image2Column) Convolution




- Transform convolution to matrix multiplication
- Unified calculation for both convolution and fully-connected layers



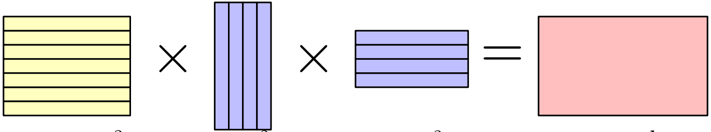
$\mathbf{X} \in \mathbb{R}^{d \times (k^2 c)}$ $\quad \mathbf{S} \in \mathbb{R}^{(k^2 c) \times n}$ $\quad \mathbf{Y} \in \mathbb{R}^{d \times n}$

## Sparse DNN

- *Sparsification*: weight pruning;
- *Compression*: compressed sparse format for storage;
- *Potential acceleration*: sparse matrix multiplication algorithm.

[4] Wei Wen et al. (2016). “Learning structured sparsity in deep neural networks”. In: *Proc. NIPS*, pp. 2074–2082.

[5] Yihui He, Xiangyu Zhang, and Jian Sun (2017). “Channel Pruning for Accelerating Very Deep Neural Networks”. In: *Proc. ICCV*.



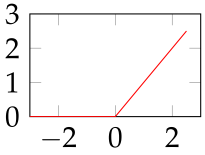
$\mathbf{X} \in \mathbb{R}^{d \times (k^2 c)}$ $\mathbf{U} \in \mathbb{R}^{(k^2 c) \times r}$ $\mathbf{V} \in \mathbb{R}^{1^2 r \times n}$ $\mathbf{Y} \in \mathbb{R}^{d \times n}$

## Low-rank DNN

- *Low-rank approximation*: matrix decomposition or tensor decomposition.
- *Compression and acceleration*: less storage required and less FLOP in computation.

[6] Xiangyu Zhang, Jianhua Zou, et al. (2015). "Efficient and accurate approximations of nonlinear convolutional networks". In: *Proc. CVPR*, pp. 1984–1992.

[7] Xiyu Yu et al. (2017). "On compressing deep models by low rank and sparse decomposition". In: *Proc. CVPR*, pp. 7370–7379.




ReLU

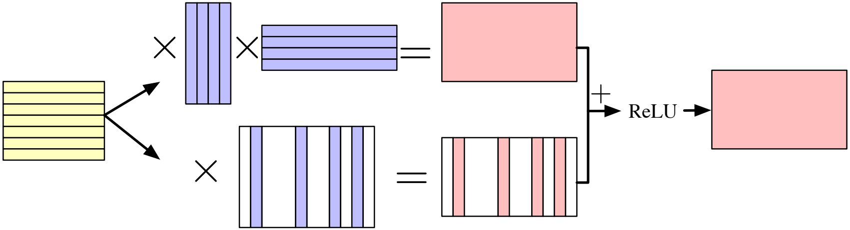
- Activation unit: `ReLU`
- Error more sensitive to positive response;
- Enlarge the solution space.

$$\min_{\boldsymbol{W}} \sum_{i=1}^{N} \|\boldsymbol{W}\boldsymbol{X}_i - \boldsymbol{Y}_i\|_F \rightarrow \min_{\boldsymbol{W}} \sum_{i=1}^{N} \|r(\boldsymbol{W}\boldsymbol{X}_i) - \boldsymbol{Y}_i\|_F$$

- $\boldsymbol{X}$: input feature map
- $\boldsymbol{Y}$: output feature map

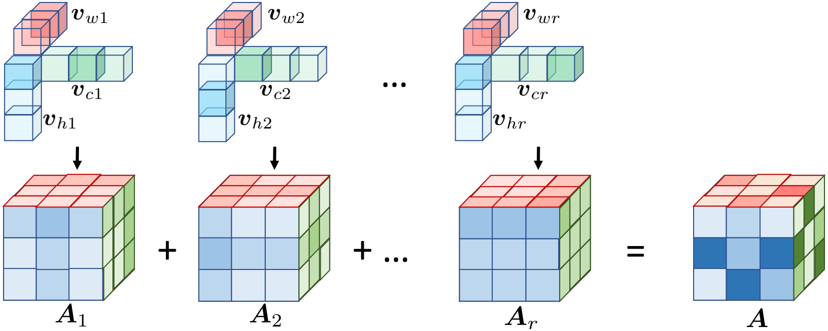
[8] Xiangyu Zhang, Jianhua Zou, et al. (2015). "Efficient and accurate approximations of nonlinear convolutional networks". In: *Proc. CVPR*, pp. 1984–1992.

# Our Idea: Unified Structure[9] (Best Student Paper Award)




- Simultaneous low-rank approximation and network sparsification;
- Non-linearity is taken into account.
- Acceleration is achieved with structured sparsity.

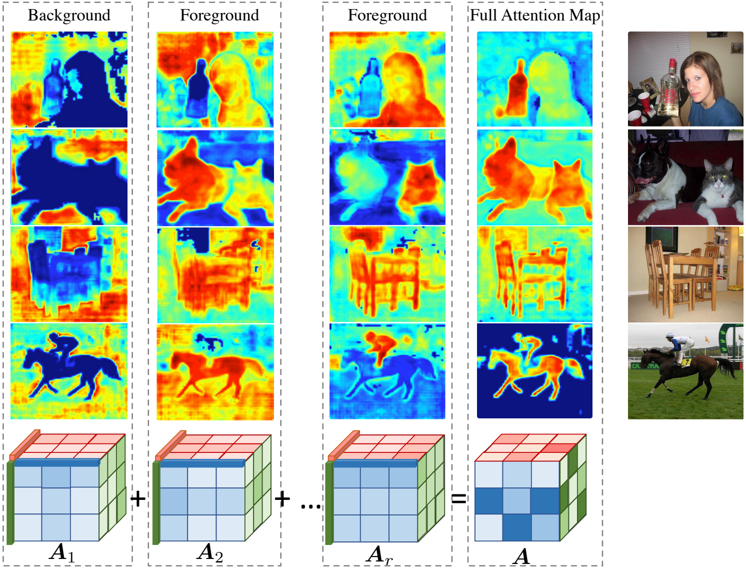
[9]Yuzhe Ma et al. (2019). "A Unified Approximation Framework for Non-Linear Deep Neural Networks". In: *Proc. ICTAI*.




Tensor Reconstruction Module (TRM). The pipeline of TRM consists of two main steps, sub-attention map generation and global context reconstruction. The processing from top to bottom (see ↓) indicates the sub-attention map generation from three dimensions (channel / height / width). The processing from left to right (see $A_1 + A_2 + \cdots + A_r = A$ ) denotes the global context reconstruction from low-rank to high-rank.

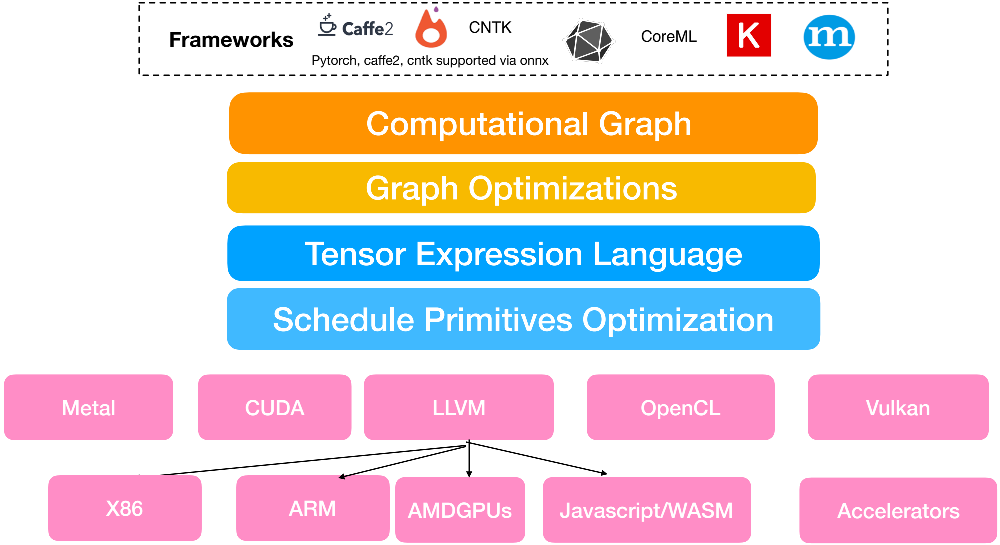
[10]Wanli Chen et al. (2020). "Tensor low-rank reconstruction for semantic segmentation". In: *Proc. ECCV*, pp. 52–69.

# Visualization of Tensor Decomposition in Segmentation



# Compilation Level

# Deep Learning Compiler Example: TVM[11]



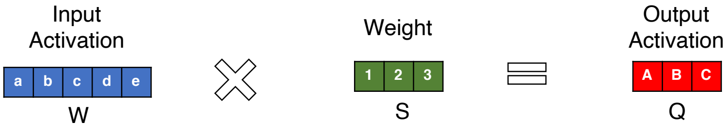

[11] Only SOTA on some AI chips; but NOT for x86 CPU (OpenVINO), GPU (TensorRT)

```c
#include <stdio.h>
#include <memory.h>
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>

double a_arr[1024][1024];
double b_arr[1024][1024];

int main()
{
    int N = 1024;
    for(int i=0;i<1024;i++) for(int j=0;j<1024;j++)
    {
        a_arr[i][j] = 1; b_arr[i][j] = 2;
    }
    double sum;

    struct timeval startTime,endTime;
    float Timeuse;
    gettimeofday(&startTime,NULL);

    // ===========================================
    //     following are the key operations
    for(int i=0; i<1024; i++){
        for(int j=0; j<1024; j++){
            sum += a_arr[j][i] * b_arr[j][i];
        }
    }
    // ===========================================

    gettimeofday(&endTime,NULL);
    Timeuse = 1000000 * (endTime.tv_sec-startTime.tv_sec) + (endTime.tv_usec-startTime.tv_us
    printf("============ Cache Optimization Demo ========== \n");
    printf("total timeuse = %.2f us \n",Timeuse);

    return 0;
}

```

```c
#include <stdio.h>
#include <memory.h>
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>

double a_arr[1024][1024];
double b_arr[1024][1024];

int main()
{
    int N = 1024;
    for(int i=0;i<1024;i++) for(int j=0;j<1024;j++)
    {
        a_arr[i][j] = 1; b_arr[i][j] = 2;
    }
    double sum;

    struct timeval startTime,endTime;
    float Timeuse;
    gettimeofday(&startTime,NULL);

    // ===========================================
    //     following are the key operations
    for(int i=0; i<1024; i++){
        for(int j=0; j<1024; j++){
            sum += a_arr[i][j] * b_arr[i][j];
        }
    }
    // ===========================================

    gettimeofday(&endTime,NULL);
    Timeuse = 1000000 * (endTime.tv_sec-startTime.tv_sec) + (endTime.tv_usec-startTime.tv_
    printf("============ Cache Optimization Demo ========== \n");
    printf("total timeuse = %.2f us \n",Timeuse);

    return 0;
}

```

Same complexity; same real runtime?

# 1D Convolution Example



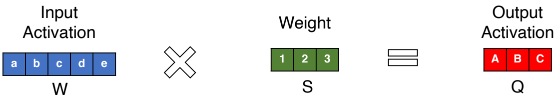

```
for(q=0; q<Q; q++){
  for (s=0; s<S; s++){
    OA[q] += IA[q+s] * W[s];
  }
}
```

**Output Stationary (OS) Dataflow**

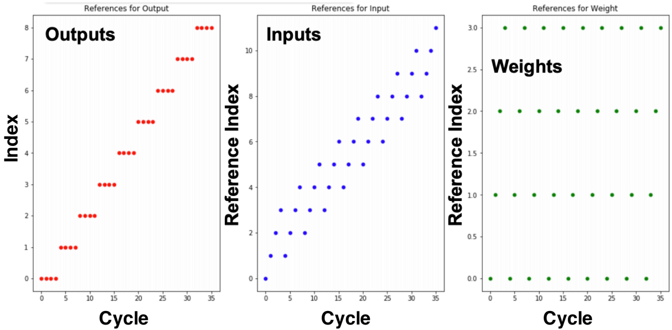
```
for (s=0; s<S; s++){
  for(q=0; q<Q; q++){
    OA[q] += IA[q+s] * W[s];
  }
}
```

**Weight Stationary (WS) Dataflow**

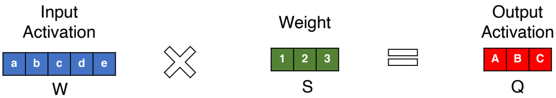
# Buffer Access Pattern 1: Output Stationary



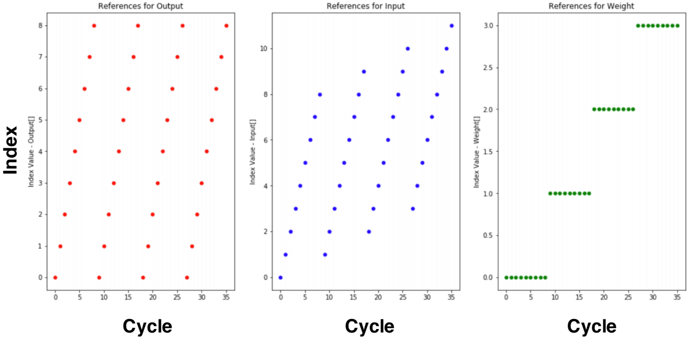

```
for(q=0; q<Q; q++){ // Q =9
  for (s=0; s<S; s++){ // S=4
    OA[q] += IA[q+s] * W[s];
  }
}
```



# Buffer Access Pattern 2: Weight Stationary

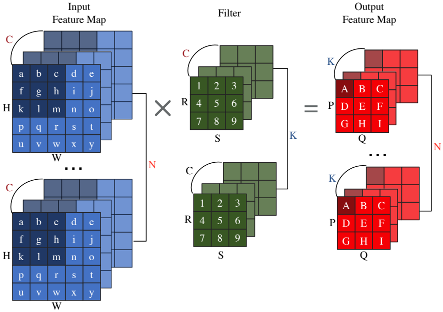



```
for (s=0; s<S; s++){// S=4
  for(q=0; q<Q; q++){// Q =9
    OA[q] += IA[q+s] * W[s];
  }
```



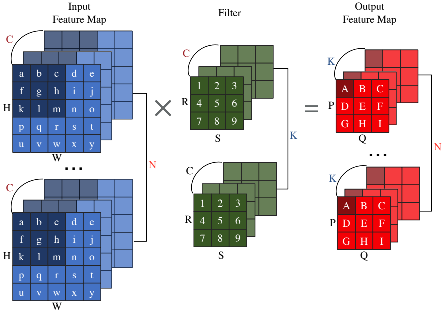
# Direct Convolution

```
for (n=0; n<N; n++) {
for (k=0; k<K; k++) {
for (p=0; p<P; p++) {
for (q=0; q<Q; q++) {
    OA[n][k][p][q]= 0;
    for (r=0; r<R; r++) {
    for (s=0; s<S; s++) {
    for (c=0; c<C; c++) {
        h = p * stride - pad + r;
        w = q * stride - pad + s;
        OA[n][k][p][q] += IA[n][c][h][w] * W[k][c][r][s];
} } } } } } }
```



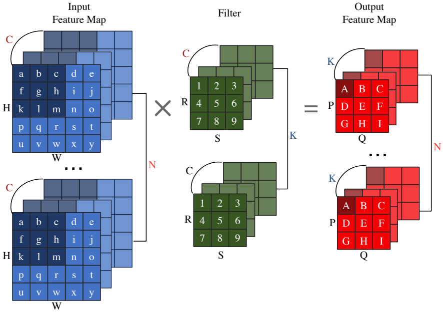
# Direct Convolution: Loop Ordering

```
for (n=0; n<N; n++) {
for (r=0; r<R; r++) {
for (s=0; s<S; s++) {
for (c=0; c<C; c++) {
for (k=0; k<K; k++) {
    float curr_w = W[r][s][c][k];
    for (p=0; p<P; p++) {
    for (q=0; q<Q; q++) {
        h = p * stride - pad + r;
        w = q * stride - pad + s;
        OA[n][k][p][q] += IA[n][c][h][w] * curr_w;
} } } } } } }
```



# Direct Convolution: Loop Ordering + Unrolling

```
for (n=0; n<N; n++) {
for (r=0; r<R; r++) {
for (s=0; s<S; s++) {
    spatial_for (c=0; c<C; c++) {
    spatial_for (k=0; k<K; k++) {
    float curr_w = W[r][s][c][k];
    for (p=0; p<P; p++) {
    for (q=0; q<Q; q++) {
        h = p * stride - pad + r;
        w = q * stride - pad + s;
        OA[n][k][p][q] += IA[n][c][h][w] * curr_w;
} } } } } } }
```



# Direct Convolution: Loop Ordering + Unrolling + Tiling

```
for (n=0; n<N; n++) {
for (r=0; r<R; r++) {
for (s=0; s<S; s++) {
for (c_t=0; c_t<C/16; c_t++) {
for (k_t=0; k_t<K/64; k_t++) {
spatial_for (c_s=0; c_s<16; c_s++) {
spatial_for (k_s=0; k_s<64; k_s++) {
    int curr_c = c_t * 16 + c_s;
    int curr_k = k_t * 64 + k_s;
    float curr_w = W[r][s][curr_c][curr_k];
    for (p=0; p<P; p++) for (q=0; q<Q; q++) {
        h = p * stride - pad + r; w = q * stride - pad + s;
        OA[n][curr_k][p][q] += IA[n][curr_c][h][w] * curr_w;
} } } } } }
```

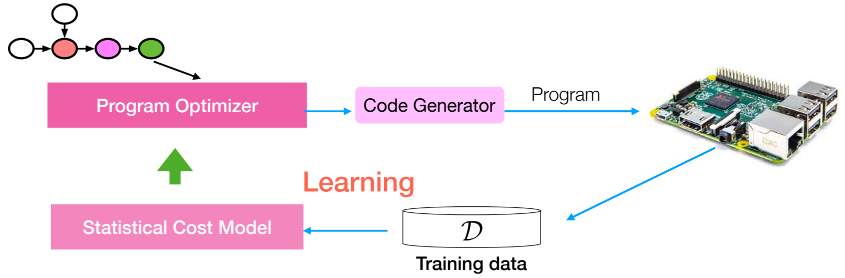
# Direct Convolution: Loop Ordering + Unrolling + Tiling

```
for (n=0; n<N; n++) {
for (r=0; r<R; r++) {
for (s=0; s<S; s++) {
for (c_t=0; c_t<C/16; c_t++) {
for (k_t=0; k_t<K/64; k_t++) {
spatial_for (c_s=0; c_s<16; c_s++) {
spatial_for (k_s=0; k_s<64; k_s++) {
    int curr_c = c_t * 16 + c_s;
    int curr_k = k_t * 64 + k_s;
    float curr_w = W[r][s][curr_c][curr_k];
    for (p=0; p<P; p++) for (q=0; q<Q; q++) {
        h = p * stride - pad + r; w = q * stride - pad + s;
        OA[n][curr_k][p][q] += IA[n][curr_c][h][w] * curr_w;
} } } } } }
```

## Questions:

- How many configurations we have?
- How to search for the BEST configuration?
- How to handle different backend devices?

## Layer-wise Optimization: Autotuning

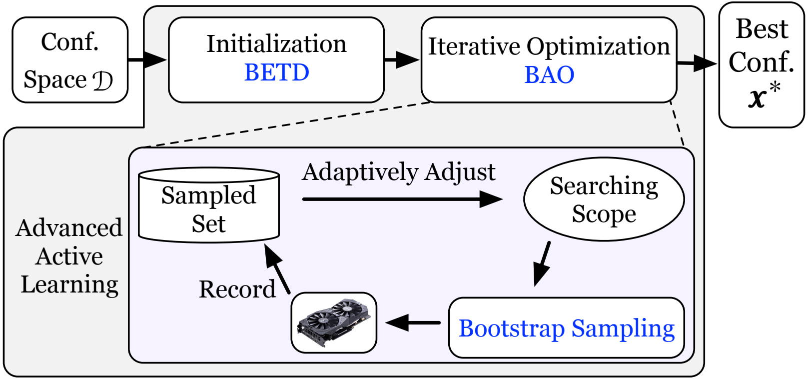



Tuning algorithms:

- Active learning.
- Transfer learning.
- Reinforcement learning.

# Active Learning driven TVM [DATE'21][12]

- Batch transductive experimental design
- Bootstrap-guided adaptive optimization



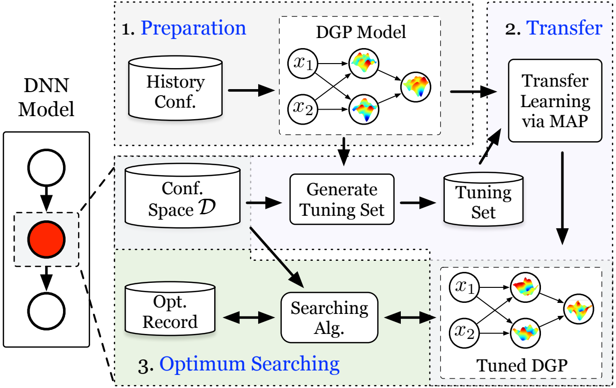

[12]Qi Sun, Chen Bai, Hao Geng, et al. (2021). "Deep neural network hardware deployment optimization via advanced active learning". In: *Proc. DATE*, pp. 1510–1515.

# Deep Gaussian Transfer Learning [ICCV'21][13]

① preparation: learn a deep Gaussian process model from historical data

② transfer: transfer knowledge of the DGP model to new tasks

③ optimal searching: guide the optimization of new tasks with the tuned DGP model




[13] Qi Sun, Chen Bai, Tinghuan Chen, et al. (2021). "Fast and Efficient DNN Deployment via Deep Gaussian Transfer Learning". In: *Proc. ICCV*.

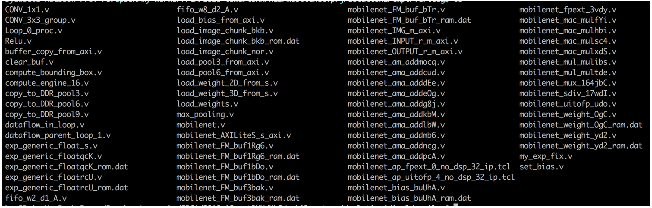
# Hardware Implementation

# Performance-Driven Neural Network Design

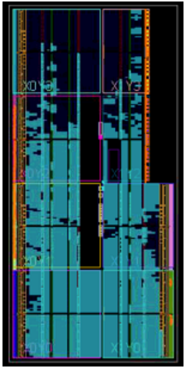
| | | |
|---|---|---|
| **Q 1** **Layer Type?** | ▪ Limited hardware resources, 5FPS speed requirement – **Light neural network** | ✓ **Depth-wise** and **Point-wise** convolutional layers |
| **Q 2** **Kernel Type?** | ▪ Hardware friendly – **More IP reuse** among layers – Less types of kernels, less IPs | ✓ **Two types** of convolutional kernels (3x3 and 1x1) |
| **Q 3** **# of layers?** | ▪ Computed from the latency constraint **(5 FPS, 200ms )** – **Maximum # of computations** | ✓ **8 ~ 10 layers** of convolutional layers, **12~14** with pooling |

# FPGA Design Flow

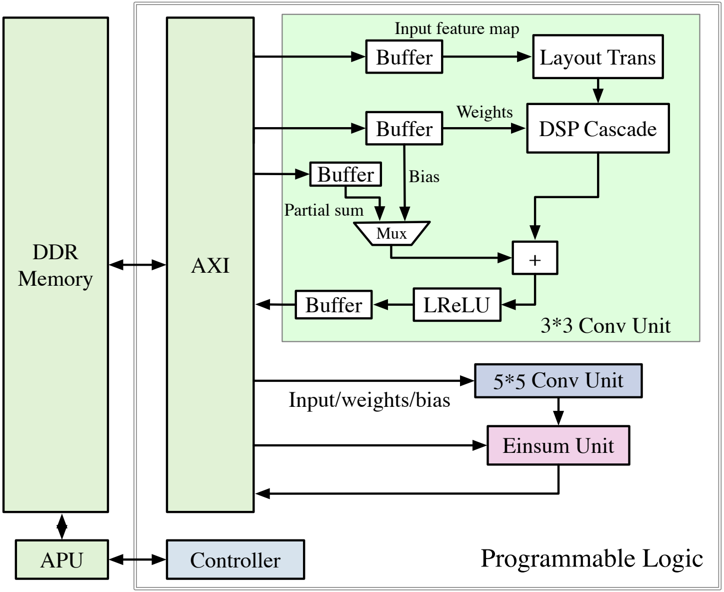
1. (optional) DNN Design in C/C++ (`example`)
2. Generate RTL (`example`) by tool Vivado HLS

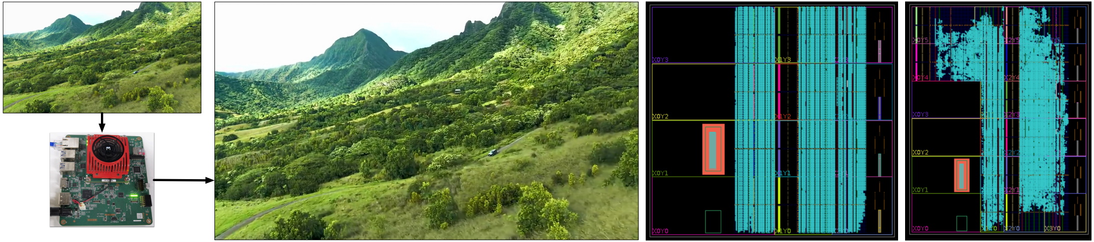



3. Generate bitstream by tool Vivado IDE
4. Load bitstream to FPGA board



# Architecture of our SR accelerator





(a) (b) (c)

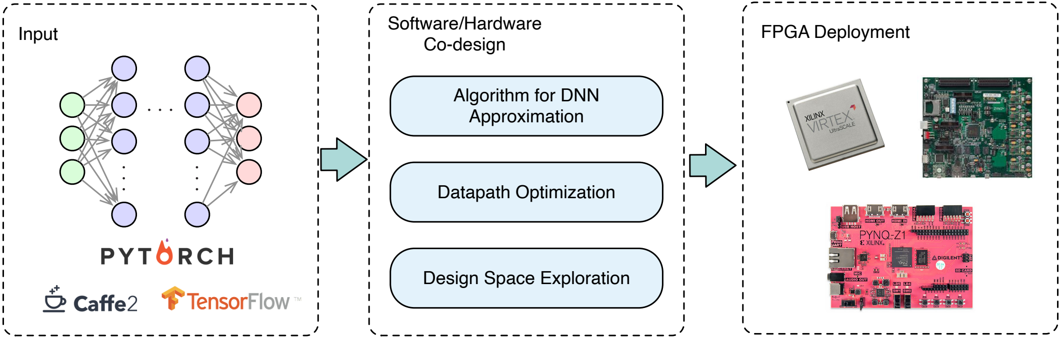
(a) 540p $\rightarrow$ 1080p SR running on KV260; (b) Physical implementation on KV260; (c) Physical implementation on ZCU104.

# Comparison

- Comparison with SOTA FPGA solutions and commercial GPU and ASIC
- 540p→1080p SR is based on KV260
- 720p→1440p is based on ZCU104

| Accelerator | 540p→1080p Latency (ms) | 540p→1080p FPS | 540p→1080p Price ($) | 720p→1440p Latency (ms) | 720p→1440p FPS | 720p→1440p Price ($) |
|---|---|---|---|---|---|---|
| NX GPU | 35.36 | 28.28 | 399 | 61.83 | 16.17 | **399** |
| Ascend 310 | 48.55 | 20.60 | 999 | 87.57 | 11.42 | 999 |
| Xilinx DPU | >37.31 | < 26.80 | 199 | >56.14 | < 17.81 | 1554 |
| DNNBuilder | >186.57 | < 5.36 | 199 | >82.92 | < 12.06 | 1554 |
| Ours | **27.94** | **35.79** | **199** | **39.74** | **25.16** | 1554 |

# Conclusion


Input
PYTORCH
Caffe2
TensorFlow
Software/Hardware
Co-design
Algorithm for DNN
Approximation
Datapath Optimization
Design Space Exploration
FPGA Deployment
XILINX
VIRTEX
UltraSCALE
PYNQ-Z1

# THANK YOU!