MIT HAN LAB

Lecture 17
TinyEngine - Efficient Training and Inference on Microcontrollers

Song Han
songhan@mit.edu

Lecture Plan

Deploy Neural Networks on Microcontrollers with TinyEngine

Today we will:

1. Introduce what microcontrollers are and why they are essential.
   - Important characteristics and components of microcontrollers.
2. Reveal critical factors of deploying neural networks on microcontrollers.
   - Why deploying neural networks on microcontrollers is challenging?
   - Essential data layouts/formats of neural networks on microcontrollers.
3. Demonstrate the critical optimization techniques used in TinyEngine.
   - To improve computing speed and memory footprint.
   - How the optimization techniques work.
     - Loop unrolling, loop reordering, loop tiling, SIMD programming, Im2col convolution, in-place depth-wise convolution, NHWC for point-wise convolution and NCHW for depth-wise convolution, and Winograd convolution.

Lecture Plan

Deploy Neural Networks on Microcontrollers with TinyEngine

Today we will:

1. **Introduce what microcontrollers are and why they are essential.**
   - Important characteristics and components of microcontrollers.
2. Reveal critical factors of deploying neural networks on microcontrollers.
   - Why deploying neural networks on microcontrollers is challenging?
   - Essential data layouts/formats of neural networks on microcontrollers.
3. Demonstrate the critical optimization techniques used in TinyEngine.
   - To improve computing speed and memory footprint.
   - How the optimization techniques work.
     - Loop unrolling, loop reordering, loop tiling, SIMD programming, Im2col convolution, in-place depth-wise convolution, NHWC for point-wise convolution and NCHW for depth-wise convolution, and Winograd convolution.

Section 1: Introduction to Microcontroller

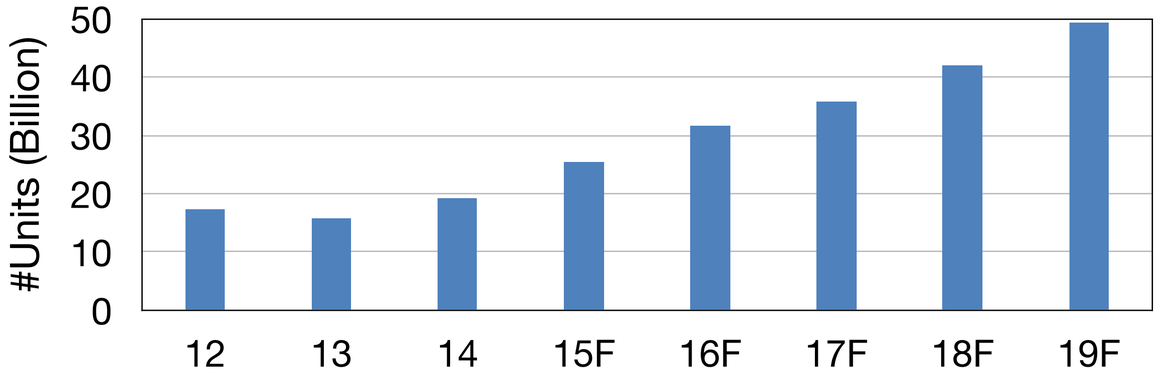
What Microcontrollers Are and Why They Are Essential?

The Era of AIoT on Microcontrollers

Microcontroller unit (MCU)

- **Low-cost, low-power**

- **Rapid growth**

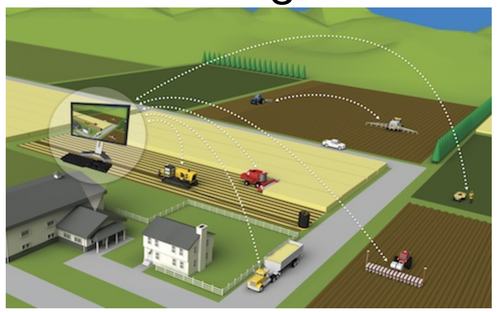


- **Wide applications**

Smart Retail

Personalized Healthcare

Precision Agriculture



Smart Home

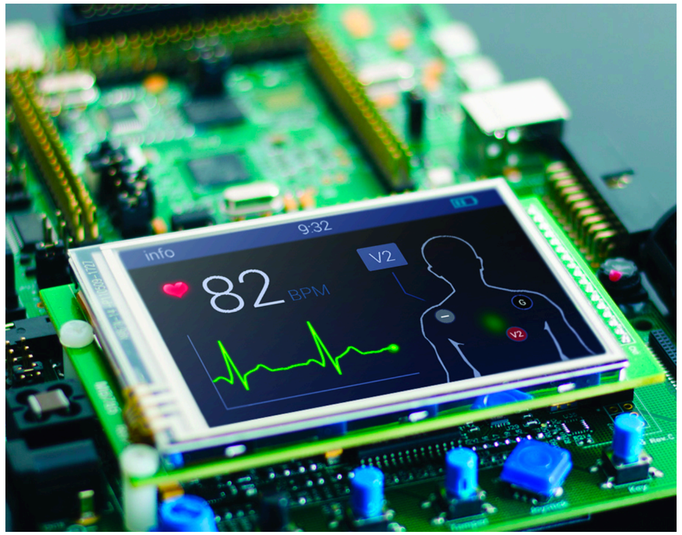
...

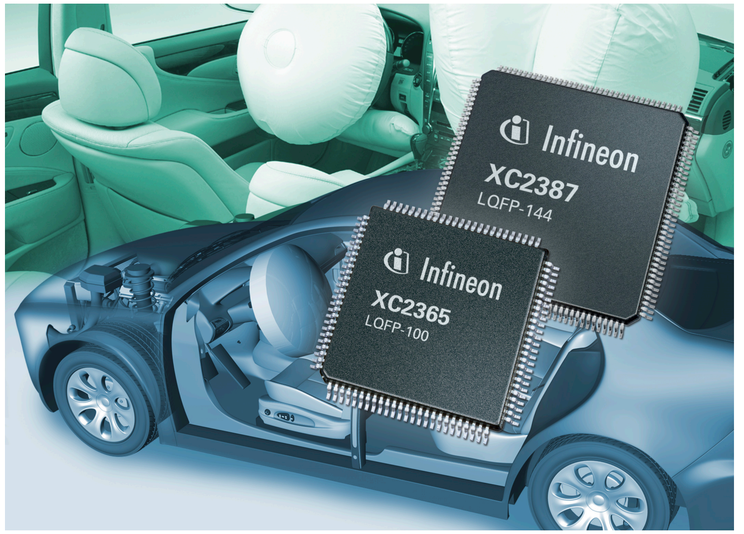
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning [Lin et al., NeurIPS 2021]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Microcontrollers are Ubiquitous

Microcontroller unit (MCU)

- Microcontrollers are ubiquitous in our daily lives and can be found in:
  - Vehicles
  - Robots
  - Office machines
  - Medical devices
  - Mobile radio transceivers
  - Vending machines
  - Home appliances



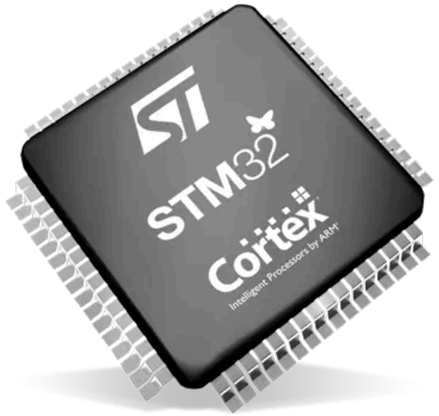


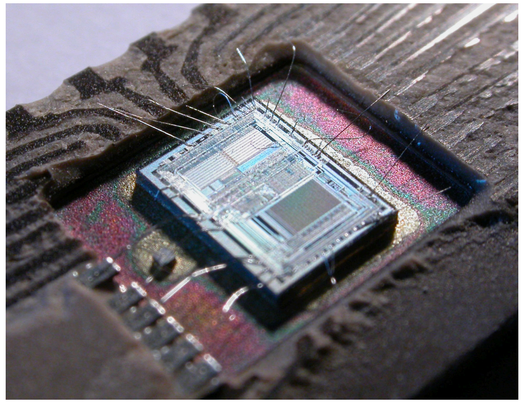
- There are several dozen microcontroller architectures and vendors:
  - STMicroelectronics: STM8 (8-bit), ST10 (16-bit), STM32 (32-bit)
  - Texas Instruments: TI MSP430 (16-bit), MSP432 (32-bit), C2000 (32-bit)
  - Microchip Technology: Atmel AVR (8-bit), AVR32 (32-bit), and AT91SAM (32-bit)

"microcontroller (MCU)" [Link], Microcontroller [Link], Infineon's microcontroller [Link], Amazon Alexa [Link], Caliber Interconnect [Link]

Introduction to Microcontroller

- Microcontrollers are essentially simple miniature personal computers designed to control small features of a larger component.
  - Without an operating system (OS)
- Advantages of Microcontrollers:
  - Cost-effective
  - Low power
  - Small chip area
- Disadvantages of Microcontrollers:
  - Low computational capability
  - Small memory/storage space
  - Limited instruction set

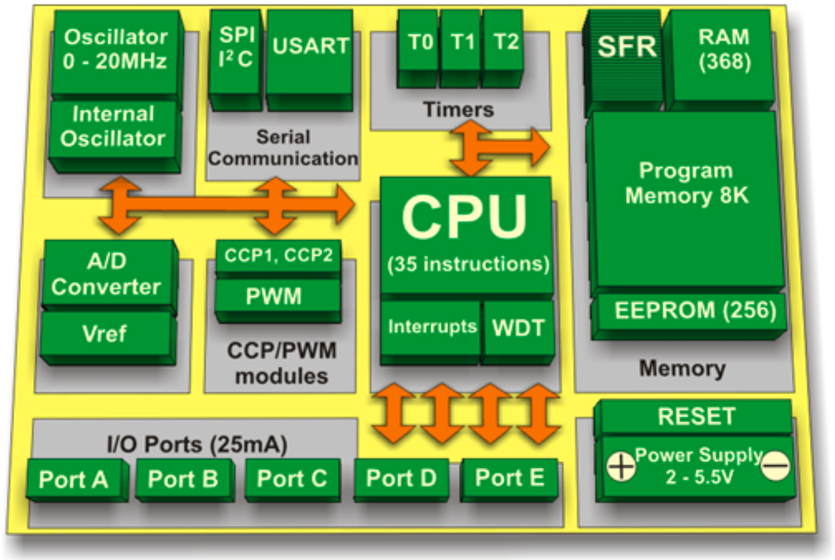


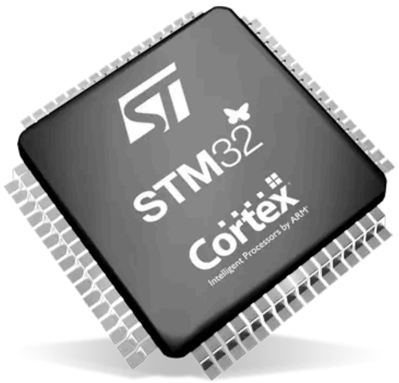


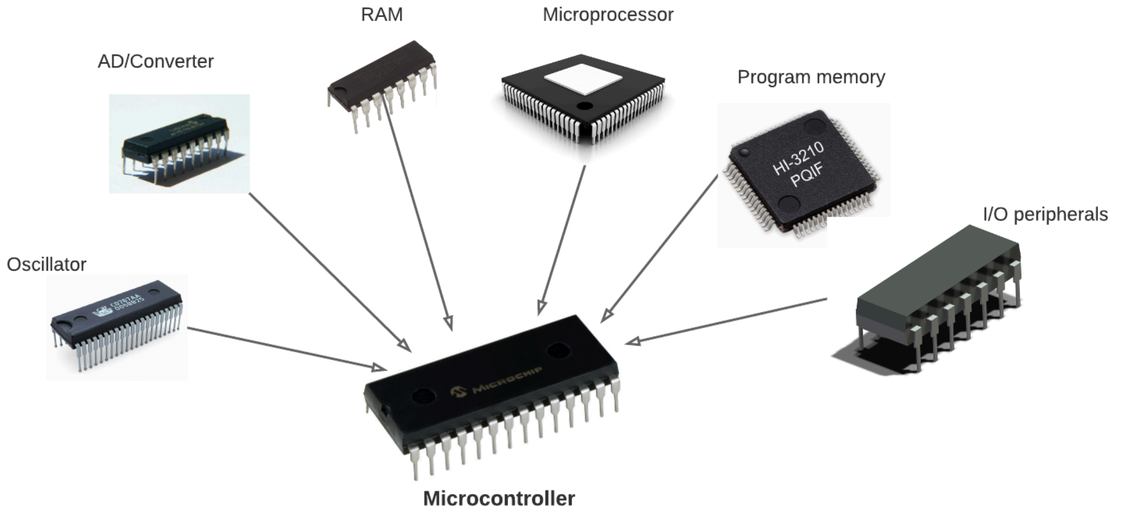
Microcontroller [Link], Basics of Microcontrollers [Link], STMicroelectronics STM32 [Link], GD32F103CBT6 [Link]

Basic Structure of Microcontrollers

- A microcontroller is commonly with the following features:
  - Central processing units (CPUs)
  - Volatile memory, e.g., SRAM
  - Nonvolatile memory, e.g., ROM, Flash memory
  - Serial input/output, e.g., serial ports (UARTs)
  - Peripherals, e.g., timers, event counters and watchdog
  - Analog-to-digital converters (ADCs)
  - Digital-to-analog converters (DACs)







Microcontroller [Link], PIC Microcontroller [Link], Introduction to Microcontrollers [Link]

Microcontroller vs. Laptop

Arm Cortex-M7-core-based STM32F746 MCU vs. Apple MacBook Pro (M1 Ultra)

	# CPU Core	Max. CPU Clock Rate	# GPU Cores	# Neural Engine Cores	L1 Cache Size	L2 Cache Size	L3 Cache Size	Memory Capacity	Storage Capacity	Operating System
STM32F746 MCU	1	216MHz	N/A	N/A	8KB	N/A	N/A	320KB	1MB	N/A
Apple MacBook Pro (M1 Ultra)	20	3200MHz	64	32	320KB	48MB	96MB	64GB	8TB	macOS
Difference Ratio	**20x**	**15x**	**-**	**-**	**40x**	**-**	**-**	**210000x**	**8400000x**	**-**

STM32 F4 microcontroller [Link], Apple M1 [Link], Apple MacBook Pro [Link]

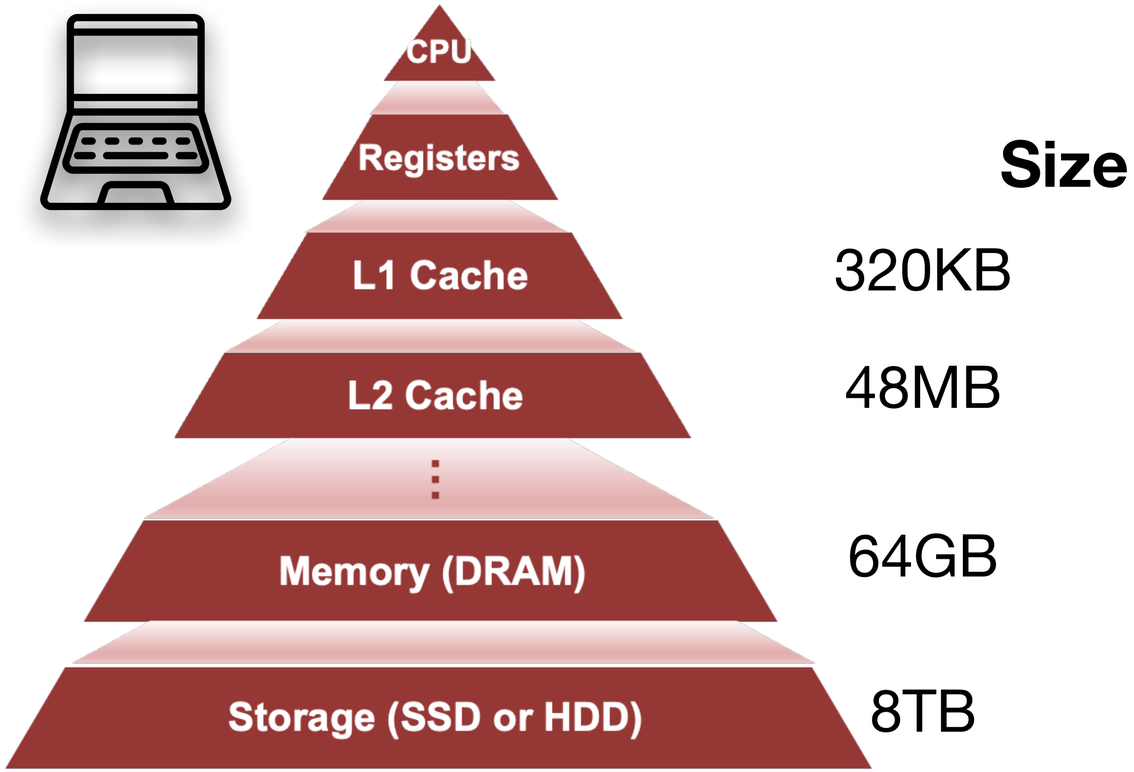
Memory Hierarchy

E.g., laptops, personal computers

Latency Numbers Every Programmer Should Know [Link]

Memory Hierarchy

Personal Computer vs. Microcontroller

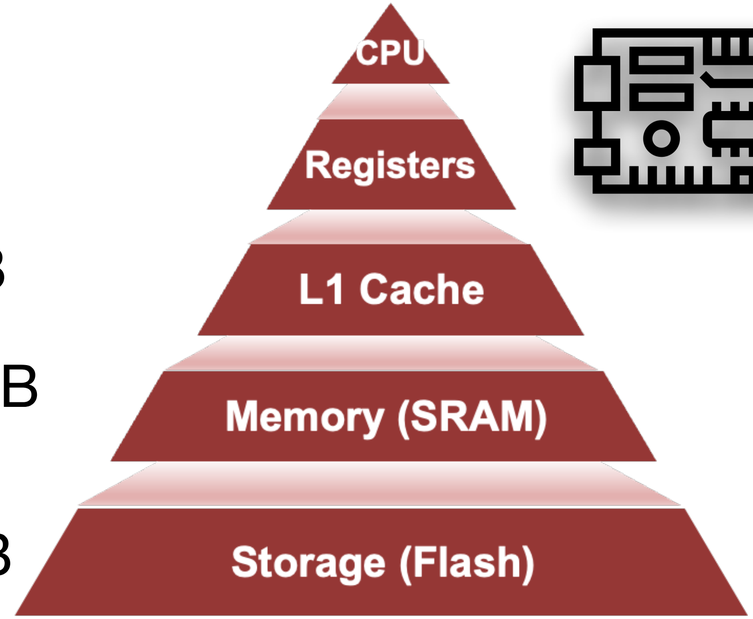


Latency Numbers Every Programmer Should Know [Link]

Memory Hierarchy

Personal Computer vs. Microcontroller

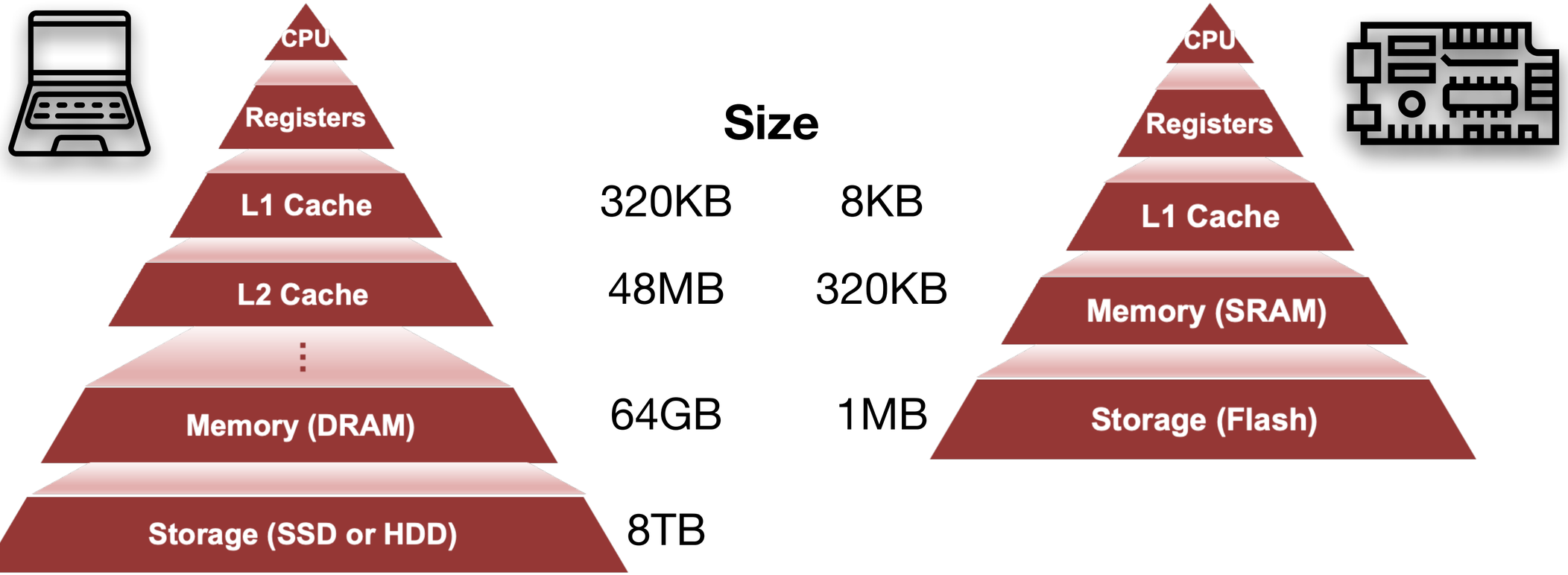
	Size	
L1 Cache	320KB	8KB
L2 Cache	48MB	320KB
Memory (DRAM)	64GB	1MB
Storage (SSD or HDD)	8TB	



Latency Numbers Every Programmer Should Know [Link]

Memory Hierarchy

Personal Computer vs. Microcontroller



How to effectively utilize the L1 cache is one of the most critical factors to boost MCUs' performance.

Latency Numbers Every Programmer Should Know [Link]

Lecture Plan

Deploy Neural Networks on Microcontrollers with TinyEngine

Today we will:

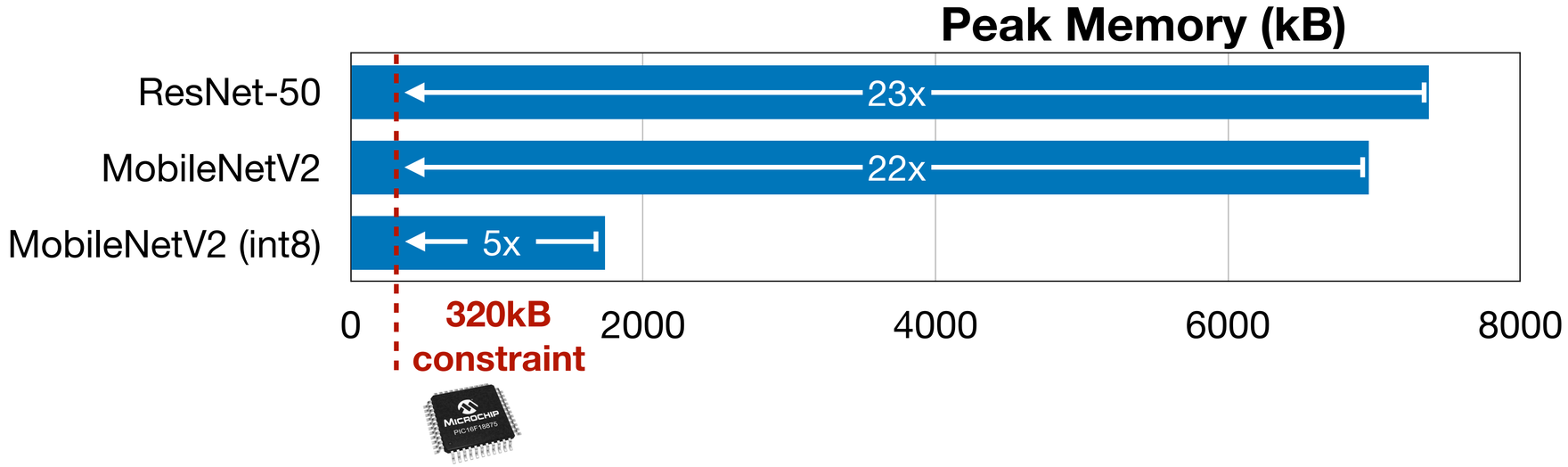
1. Introduce what microcontrollers are and why they are essential.
   - Important characteristics and components of microcontrollers.
2. **Reveal critical factors of deploying neural networks on microcontrollers.**
   - Why deploying neural networks on microcontrollers is challenging?
   - Essential data layouts/formats of neural networks on microcontrollers.
3. Demonstrate the critical optimization techniques used in TinyEngine.
   - To improve computing speed and memory footprint.
   - How the optimization techniques work.
     - Loop unrolling, loop reordering, loop tiling, SIMD programming, Im2col convolution, in-place depth-wise convolution, NHWC for point-wise convolution and NCHW for depth-wise convolution, and Winograd convolution.

Section 2: Neural Networks on Microcontrollers

Why Deploying Neural Networks on Microcontrollers is Challenging?

Challenge: Memory Too Small to Hold DNN

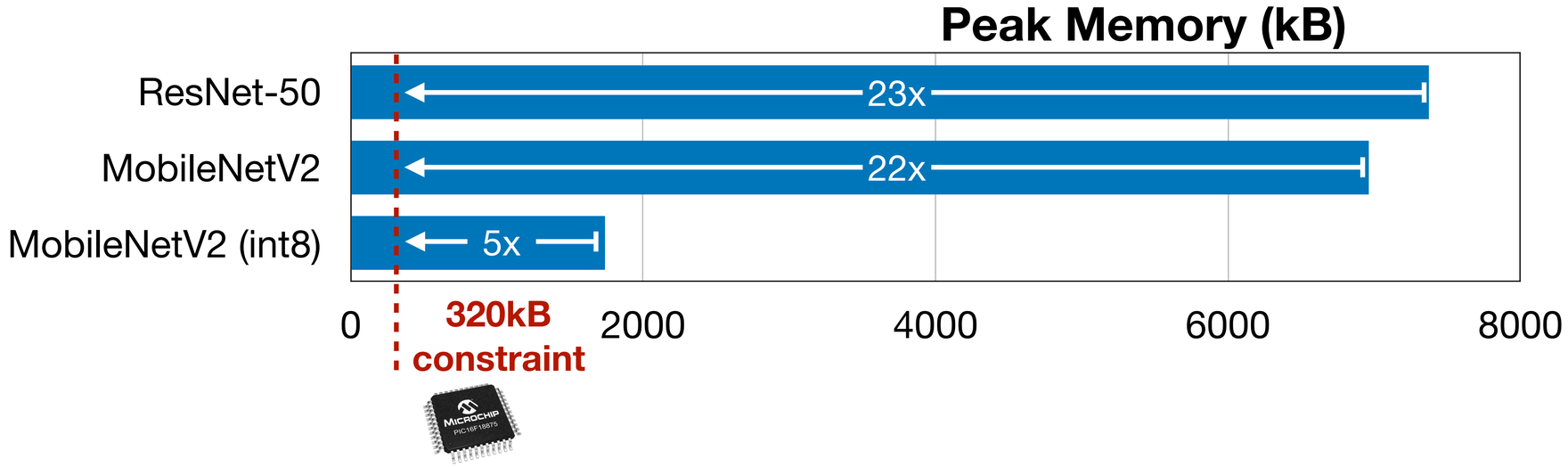
MCUNet: Tiny Deep Learning on IoT Devices

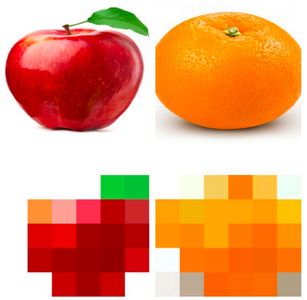


MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Challenge: Memory Too Small to Hold DNN

MCUNet: Tiny Deep Learning on IoT Devices

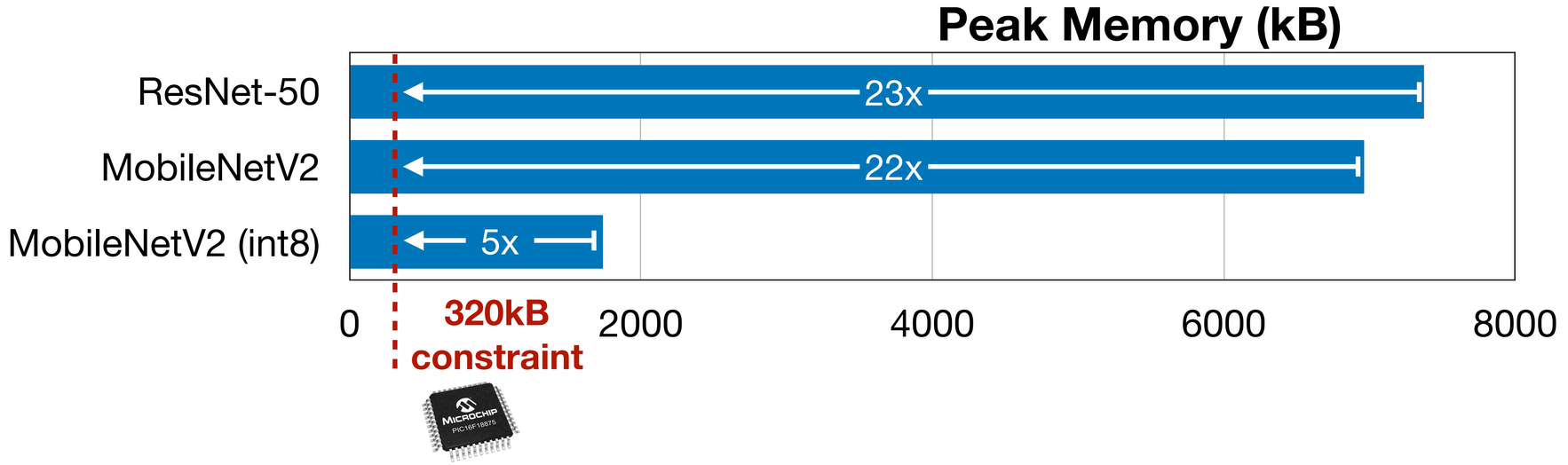


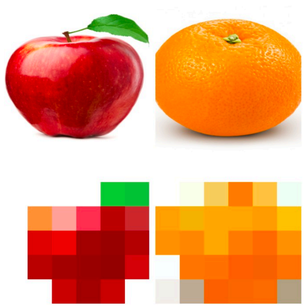


MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Challenge: Memory Too Small to Hold DNN

MCUNet: Tiny Deep Learning on IoT Devices



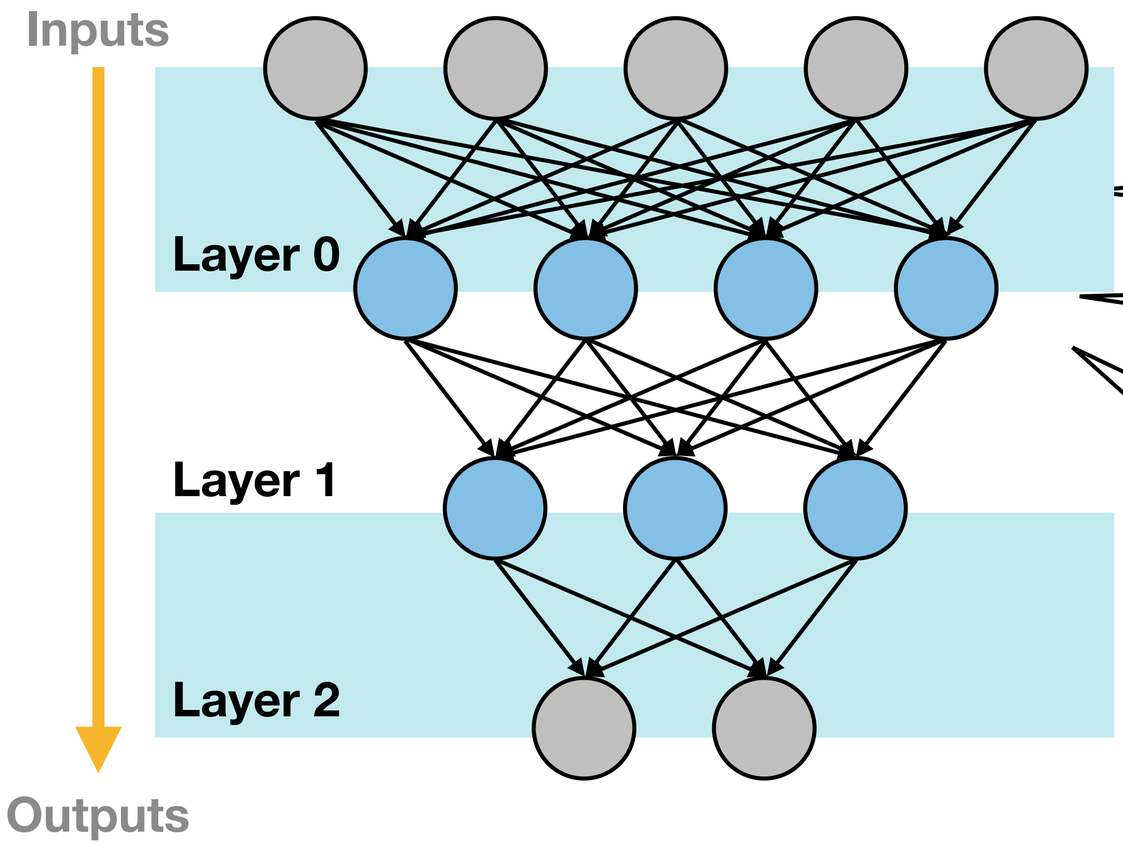


MCUNet

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

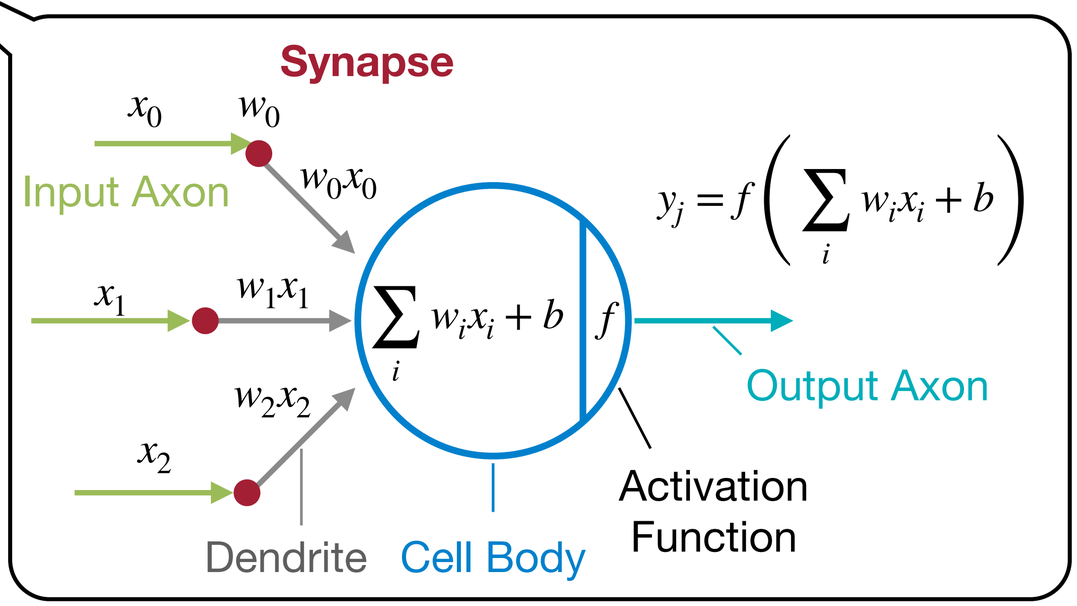
Primary Data Types in Neural Networks

Activations and Weights in MCUs



Synapses / Parameters / Weights (in Flash)

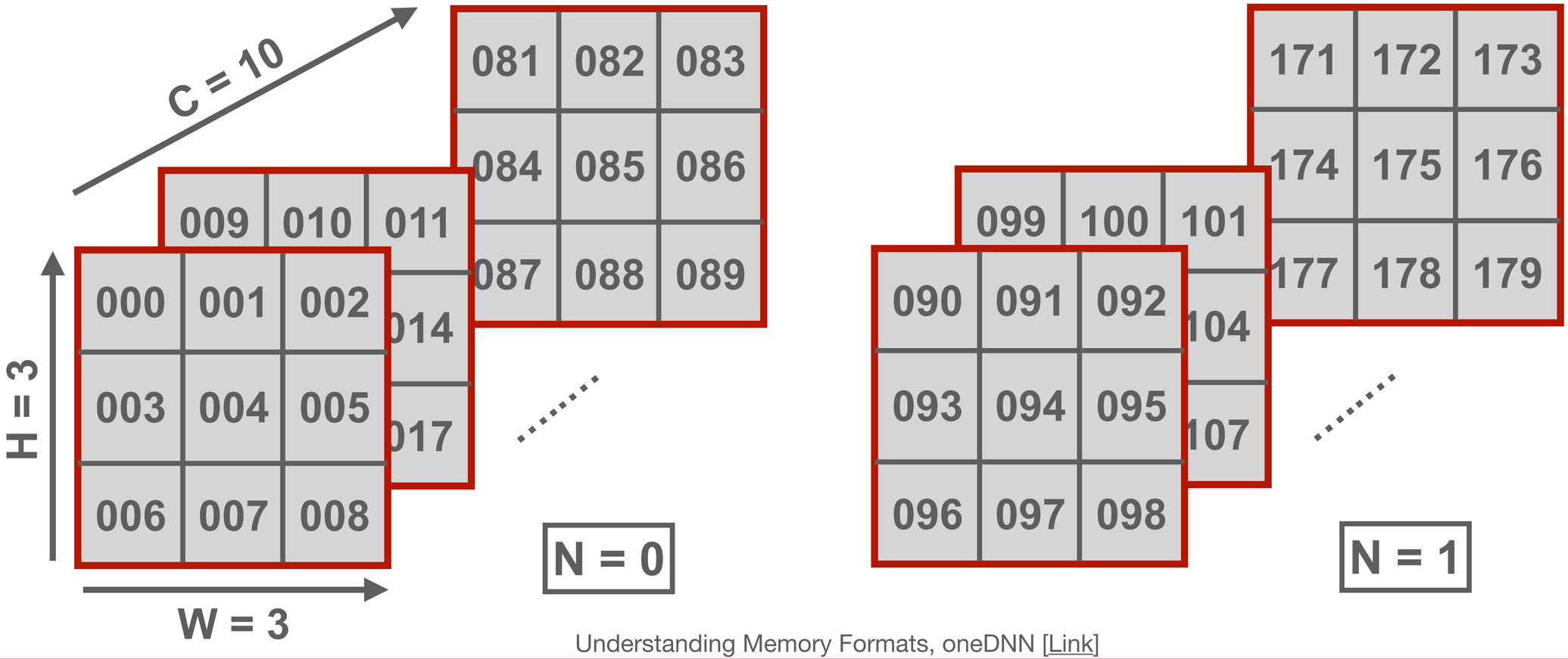
Neurons / Features / Activations (in SRAM)



Primary Data Layouts in Neural Networks

NCHW, NHWC, and CHWN

- Convolutions (both activations and weights) typically operate on four-dimensional tensors:
  - **N** feature maps/kernels of **C** channels of **H x W** spatial domain.
  - Now, consider 4D data with N = 2, 10 channels, and 3 x 3 spatial domain.

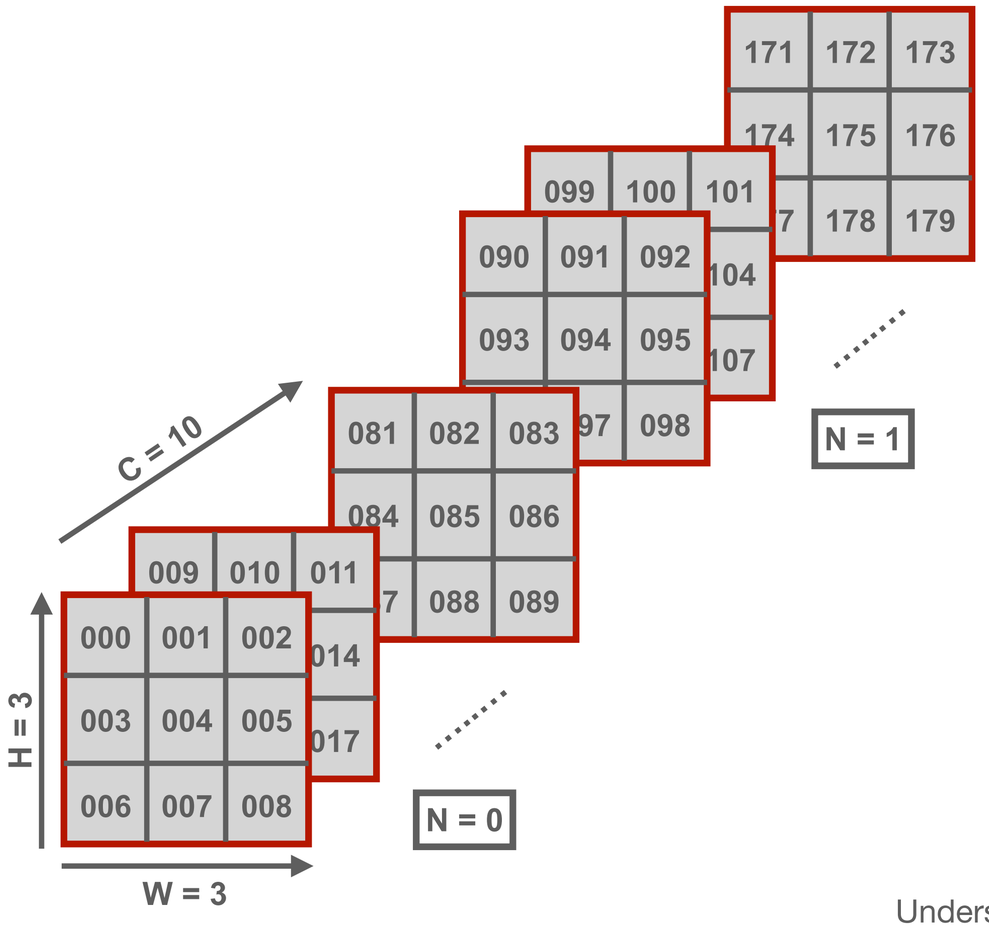


Understanding Memory Formats, oneDNN [Link]

Primary Data Layouts in Neural Networks

NCHW, NHWC, and CHWN

- Convolutions (both activations and weights) typically operate on four-dimensional tensors:
  - N feature maps/kernels of C channels of H x W spatial domain.



NCHW: (used by default in Caffe)

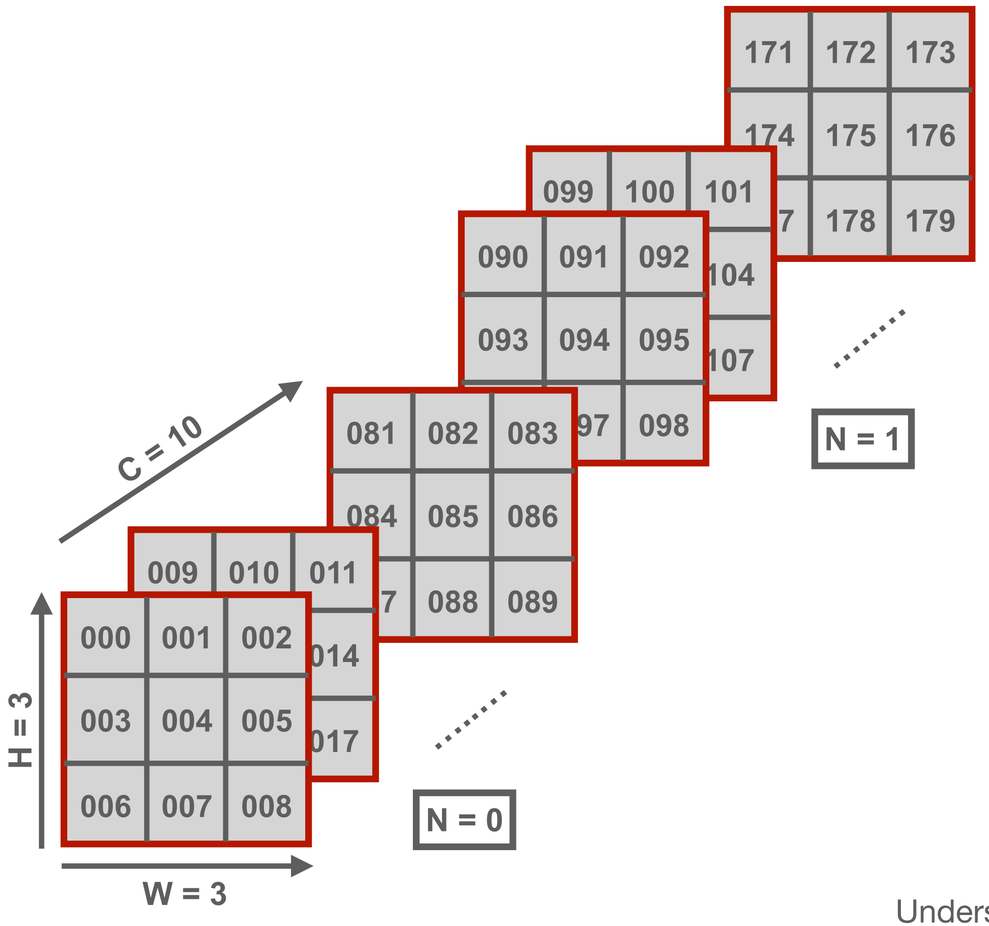
000	001	002	003	…	009	010	…	089	090	091	…	177	178	179

Understanding Memory Formats, oneDNN [Link]

Primary Data Layouts in Neural Networks

NCHW, NHWC, and CHWN

- Convolutions (both activations and weights) typically operate on four-dimensional tensors:
  - N feature maps/kernels of C channels of H x W spatial domain.



NCHW: (used by default in Caffe)

000	001	002	003	…	009	010	…	089	090	091	…	177	178	179

NHWC: (used by default in TensorFlow)

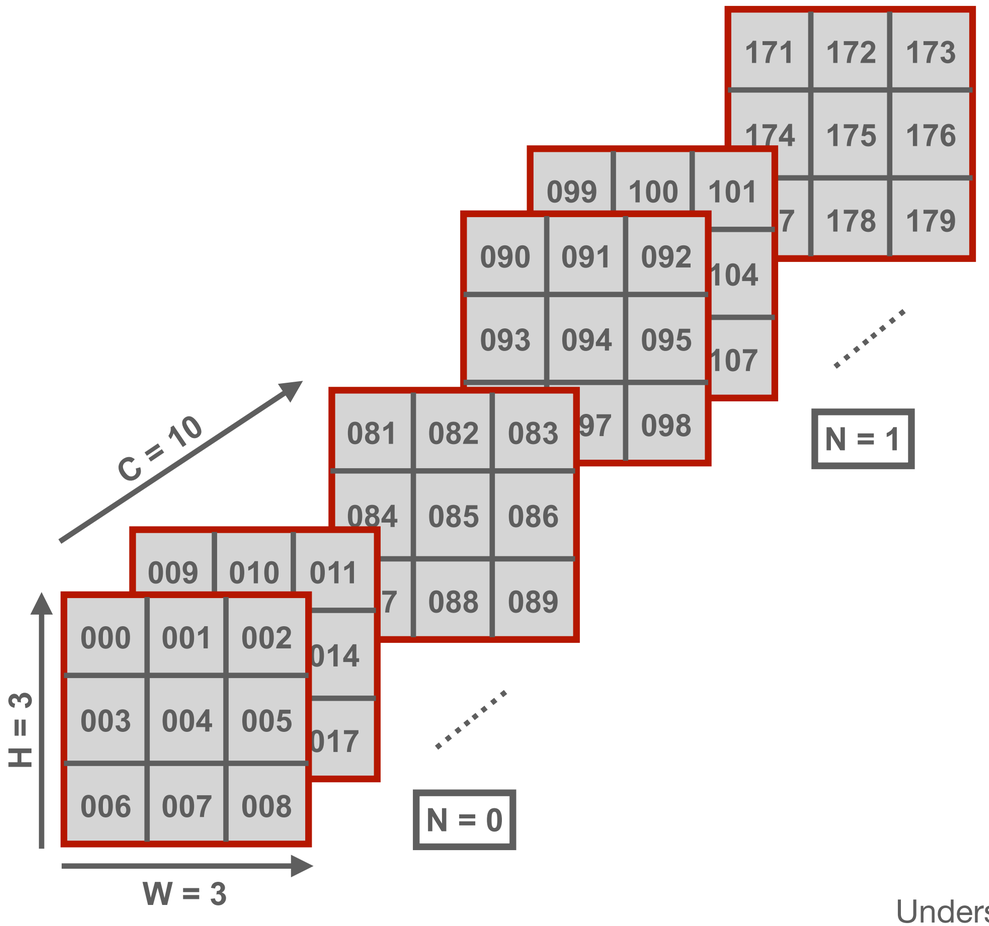
000	009	018	…	081	001	010	…	089	090	099	…	163	170	179

Understanding Memory Formats, oneDNN [Link]

Primary Data Layouts in Neural Networks

NCHW, NHWC, and CHWN

- Convolutions (both activations and weights) typically operate on four-dimensional tensors:
  - N feature maps/kernels of C channels of H x W spatial domain.



NCHW: (used by default in Caffe)

000	001	002	003	…	009	010	…	089	090	091	…	177	178	179

NHWC: (used by default in TensorFlow)

000	009	018	…	081	001	010	…	089	090	099	…	163	170	179

CHWN:

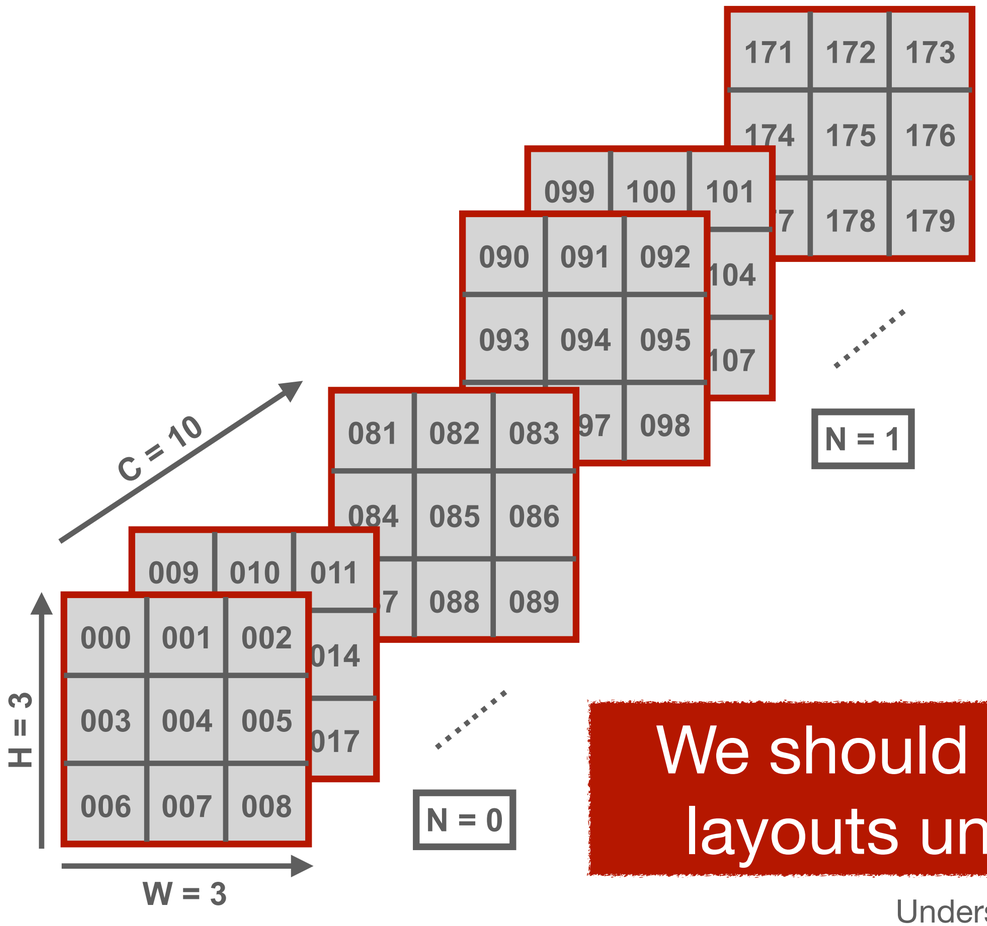
000	090	001	091	…	008	098	009	099	…	177	088	178	089	179

Understanding Memory Formats, oneDNN [Link]

Primary Data Layouts in Neural Networks

NCHW, NHWC, and CHWN

- Convolutions (both activations and weights) typically operate on four-dimensional tensors:
  - N feature maps/kernels of C channels of H x W spatial domain.



NCHW: (used by default in Caffe)

000	001	002	003	…	009	010	…	089	090	091	…	177	178	179

NHWC: (used by default in TensorFlow)

000	009	018	…	081	001	010	…	089	090	099	…	163	170	179

CHWN:

000	090	001	091	…	008	098	009	099	…	177	088	178	089	179

We should utilize different channel data layouts under various circumstances.

Understanding Memory Formats, oneDNN [Link]

Lecture Plan

Deploy Neural Networks on Microcontrollers with TinyEngine

Today we will:

1. Introduce what microcontrollers are and why they are essential.
   - Important characteristics and components of microcontrollers.
2. Reveal critical factors of deploying neural networks on microcontrollers.
   - Why deploying neural networks on microcontrollers is challenging?
   - Essential data layouts/formats of neural networks on microcontrollers.
3. **Demonstrate the critical optimization techniques used in TinyEngine.**
   - To improve computing speed and memory footprint.
   - How the optimization techniques work.
     - Loop unrolling, loop reordering, loop tiling, SIMD programming, Im2col convolution, in-place depth-wise convolution, NHWC for point-wise convolution and NCHW for depth-wise convolution, and Winograd convolution.

Section 3: Optimization Techniques in TinyEngine

TinyEngine: Memory-efficient and High-performance Neural Network Library for Microcontrollers

Optimization Techniques in TinyEngine

To enhance computing speed and reduce memory usage

- Loop unrolling:
  - A loop transformation technique that optimizes a program's execution speed at the expense of its binary size.
- Loop reordering:
  - A loop transformation technique that optimizes a program's execution speed by reordering the sequence of loops.
- Loop tiling:
  - A loop transformation technique that reduces memory access by partitioning a loop's iteration space into smaller chunks or blocks.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Image to Column (Im2col) convolution:
  - Rearranges input data to directly utilize matrix multiplication kernels.
- In-place depth-wise convolution:
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
  - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
  - Reduce the number of multiplications to enhance computing speed for convolution.

Optimization Techniques in TinyEngine

To enhance computing speed and reduce memory usage

- **Loop unrolling:**
  - A loop transformation technique that optimizes a program's execution speed at the expense of its binary size.
- Loop reordering:
  - A loop transformation technique that optimizes a program's execution speed by reordering the sequence of loops.
- Loop tiling:
  - A loop transformation technique that reduces memory access by partitioning a loop's iteration space into smaller chunks or blocks.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Image to Column (Im2col) convolution:
  - Rearranges input data to directly utilize matrix multiplication kernels.
- In-place depth-wise convolution:
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
  - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
  - Reduce the number of multiplications to enhance computing speed for convolution.

Loop Unrolling

Reduce branching overheads

- Overheads of loop control
  - Arithmetic operations for pointers (e.g., i, j, k)
  - End of loop test (e.g., k < N)
  - Branch prediction

- Reducing the overheads by loop unrolling
  - Replicate the loop body a number of times
  - Tradeoff between binary size and reduced overheads

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

Simple example: Matrix multiplication

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Loop Unrolling

Reduce branching overheads

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

e.g., unroll by 4

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N, 4): # step 1->4
            C[i][j] += A[i][k] * B[k][j]
            C[i][j] += A[i][k+1] * B[k+1][j]
            C[i][j] += A[i][k+2] * B[k+2][j]
            C[i][j] += A[I][k+3] * B[k+3][j]
```

- Arithmetic operations for pointers: N^3 -> $1/4N^3$
- Number of loop tests: N^3 -> $1/4N^3$
- Code size of the most inner loop: 1 -> 4

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Optimization Techniques in TinyEngine

To enhance computing speed and reduce memory usage

- Loop unrolling:
  - A loop transformation technique that optimizes a program's execution speed at the expense of its binary size.
- **Loop reordering:**
  - A loop transformation technique that optimizes a program's execution speed by reordering the sequence of loops.
- Loop tiling:
  - A loop transformation technique that reduces memory access by partitioning a loop's iteration space into smaller chunks or blocks.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Image to Column (Im2col) convolution:
  - Rearranges input data to directly utilize matrix multiplication kernels.
- In-place depth-wise convolution:
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
  - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
  - Reduce the number of multiplications to enhance computing speed for convolution.

Loop Reordering

Improve data locality

- Improve data locality of caches
  - Data movement (cache miss) is much more expense
  - Chuck of memory is fetched at a time (cache line)

- Reduce cache miss by loop reordering
  - Change the order of loop iteration variables
  - e.g., i, j, k -> i, k, j

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

* Assume stored in row-major order

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

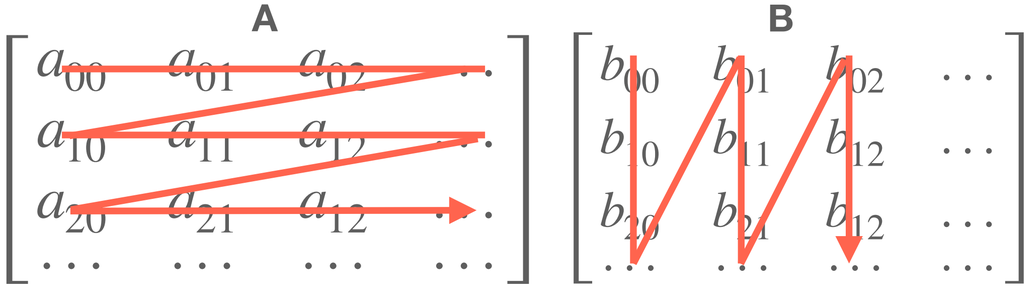
Loop Reordering

Improve data locality

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

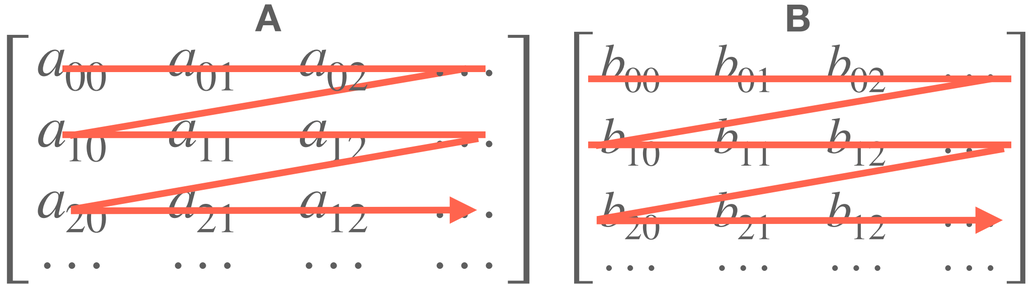
Reorder (i, j, k) as (i, k, j)

```
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```



Good data locality Poor data locality!

* Assume stored in row-major order



Good data locality

* Assume stored in row-major order

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Optimization Techniques in TinyEngine

To enhance computing speed and reduce memory usage

- Loop unrolling:
  - A loop transformation technique that optimizes a program's execution speed at the expense of its binary size.
- Loop reordering:
  - A loop transformation technique that optimizes a program's execution speed by reordering the sequence of loops.
- **Loop tiling:**
  - A loop transformation technique that reduces memory access by partitioning a loop's iteration space into smaller chunks or blocks.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Image to Column (Im2col) convolution:
  - Rearranges input data to directly utilize matrix multiplication kernels.
- In-place depth-wise convolution:
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
  - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
  - Reduce the number of multiplications to enhance computing speed for convolution.

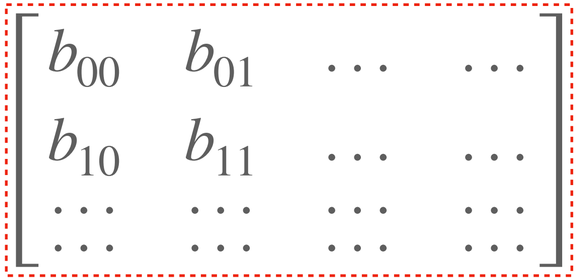
Loop Tiling

Reduce cache miss

- What if data is much larger than cache size?
  - Data in cache will be evicted before reuse -> cache miss ↑
  - e.g., B is much larger than cache size

- How loop tiling reduces cache miss?
  - Partition loop iteration space
  - Fit accessed elements in the loop into cache size
  - Ensure data stays in the cache until it is reused

```
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```



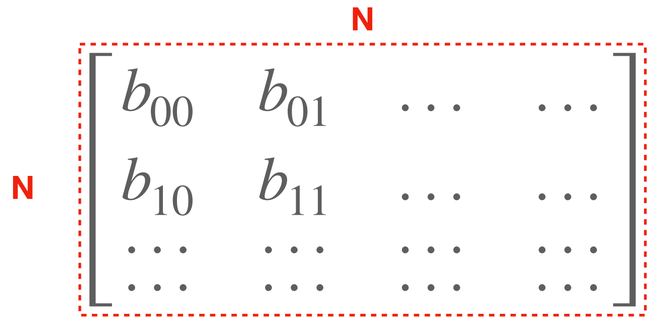
Accessed elements in B = N^2

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Loop Tiling

Reduce cache miss

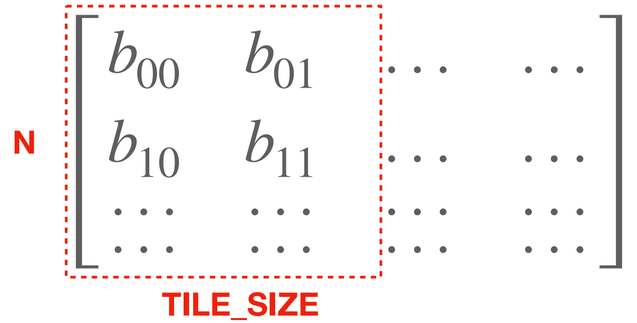
```
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```



Accessed elements in B: N^2

Tile the loop of j

```
T_j = TILE_SIZE
→ for j_t in range(0, N, T_j):
    for i in range(0, N):
        for k in range(0, N):
            → for j in range(j_t, j_t + T_j):
                C[i][j] += A[i][k] * B[k][j]
```



Accessed elements in B: N x TILE_SIZE

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Loop Tiling

Reduce cache miss

```
Tj = TILE_SIZE
for j_t in range(0, N, Tj):
    for i in range(0, N):
        for k in range(0, N):
            for j in range(j_t, j_t + Tj):
                C[i][j] += A[i][k] * B[k][j]
```

Tile the loop of k

```
Tj = Tk = TILE_SIZE
→ for k_t in range(0, N, Tk):
    for j_t in range(0, N, Tj):
        for i in range(0, N):
        → for k in range(k_t, k_t + Tk):
            for j in range(j_t, j_t + Tj):
                C[i][j] += A[i][k] * B[k][j]
```

Accessed elements in B

N

$$\begin{bmatrix} b_{00} & b_{01} & \cdots & \cdots \\ b_{10} & b_{11} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

TILE_SIZE

$$\begin{bmatrix} b_{00} & b_{01} & \cdots & \cdots \\ b_{10} & b_{11} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

TILE_SIZE

N x TILE_SIZE → TILE_SIZE²

Accessed elements in A

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots \\ a_{10} & a_{11} & a_{12} & \cdots \\ a_{20} & a_{21} & a_{12} & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots \\ a_{10} & a_{11} & a_{12} & \cdots \\ a_{20} & a_{21} & a_{12} & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

N² → N x TILE_SIZE

Loop Tiling

Reduce cache miss

```
T_j = T_k = TILE_SIZE
for k_t in range(0, N, T_k):
    for j_t in range(0, N, T_j):
        for i in range(0, N):
            for k in range(k_t, k_t + T_k):
                for j in range(j_t, j_t + T_j):
                    C[i][j] += A[i][k] * B[k][j]
```

Tile the loop of i

```
T_j = T_k = T_i = TILE_SIZE
→ for i_t in range(0, N, T_i):
    for k_t in range(0, N, T_k):
        for j_t in range(0, N, T_j):
        → for i in range(i_t, i_t + T_i):
            for k in range(k_t, k_t + T_k):
                for j in range(j_t, j_t + T_j):
                    C[i][j] += A[i][k] * B[k][j]
```

Accessed elements in A

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots \\ a_{10} & a_{11} & a_{12} & \cdots \\ a_{20} & a_{21} & a_{12} & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

TILE_SIZE

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots \\ a_{10} & a_{11} & a_{12} & \cdots \\ a_{20} & a_{21} & a_{12} & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

TILE_SIZE

N x TILE_SIZE → TILE_SIZE²

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Loop Tiling

Reduce cache miss

```
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

Loop tiling

```
T_j = T_k = T_i = TILE_SIZE
for i_t in range(0, N, T_i):
    for k_t in range(0, N, T_k):
        for j_t in range(0, N, T_j):
            for i in range(i_t, i_t + T_i):
                for k in range(k_t, k_t + T_k):
                    for j in range(j_t, j_t + T_j):
                        C[i][j] += A[i][k] * B[k][j]
```

Accessed elements in A: $N^2 \rightarrow TILE\_SIZE^2$

Accessed elements in B: $N^2 \rightarrow TILE\_SIZE^2$

Accessed elements in C: $N^2 \rightarrow TILE\_SIZE^2$

- The tile size can be determined according to the cache size
- Fitting accessed data into cache -> reuse data in cache -> cache miss ↓

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Loop Tiling

Multilevel tiling

```
Tj = Tk = Ti = TILE_SIZE
for i_t in range(0, N, Ti):
    for k_t in range(0, N, Tk):
        for j_t in range(0, N, Tj):
            for i in range(i_t, i_t + Ti):
                for k in range(k_t, k_t + Tk):
                    for j in range(j_t, j_t + Tj):
                        C[i][j] += A[i][k] * b[k][j]
```

B: $TILE\_SIZE^2$

B: N x TILE_SIZE (cache miss if we have large N!)

Tiling for multi-level caches

```
T2j =TILE2_SIZE
Tj = Tk = Ti = TILE_SIZE
→ for j_t2 in range(0, N, T2j):
    for i_t in range(0, N, Ti):
        for k_t in range(0, N, Tk):
→           for j_t1 in range(j_t2, j_t2 + T2j, Tj):
                for i in range(i_t, i_t + Ti):
                    for k in range(k_t, k_t + Tk):
                        for j in range(j_t, j_t + Tj):
                            C[i][j] += A[i][k] * b[k][j]
```

B: $TILE\_SIZE^2$ → L1 Cache

B: TILE2_SIZE x TILE_SIZE → L2 Cache

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Optimization Techniques in TinyEngine

To enhance computing speed and reduce memory usage

- Loop unrolling:
  - A loop transformation technique that optimizes a program's execution speed at the expense of its binary size.
- Loop reordering:
  - A loop transformation technique that optimizes a program's execution speed by reordering the sequence of loops.
- Loop tiling:
  - A loop transformation technique that reduces memory access by partitioning a loop's iteration space into smaller chunks or blocks.
- **SIMD (single instruction, multiple data) programming:**
  - Performs the same operation on multiple data points simultaneously.
- Image to Column (Im2col) convolution:
  - Rearranges input data to directly utilize matrix multiplication kernels.
- In-place depth-wise convolution:
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
  - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
  - Reduce the number of multiplications to enhance computing speed for convolution.

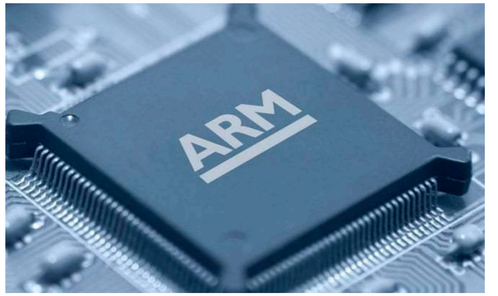
Introduction to Instruction Set

Instruction set architecture (ISA)

- An instruction set architecture acts as an interface between the software and the hardware.
  - Part of the abstract model of a computer defining how the CPU is controlled by the software.
  - Specify both what the processor is capable of doing as well as how it gets done.
  - Provide the only way through which a user is able to interact with the hardware.
- Examples of an instruction set:
  - ADD - Add two numbers together.
  - COMPARE - Compare numbers.
  - JUMP - Jump to a designated RAM address.
  - JUMP IF - Conditional statement that jumps to a designated RAM address.
  - LOAD - Load information from RAM to the CPU.
  - STORE - Store information to RAM.
  - IN/OUT - I/O for a device, e.g., monitor.

Instruction set [Link]; Instruction set architecture [Link]; Apple [Link]; Intel Skylake [Link]; AMD Ryzen [Link]; RISC-V [Link]

Introduction to Instruction Set



Instruction set architecture (ISA)

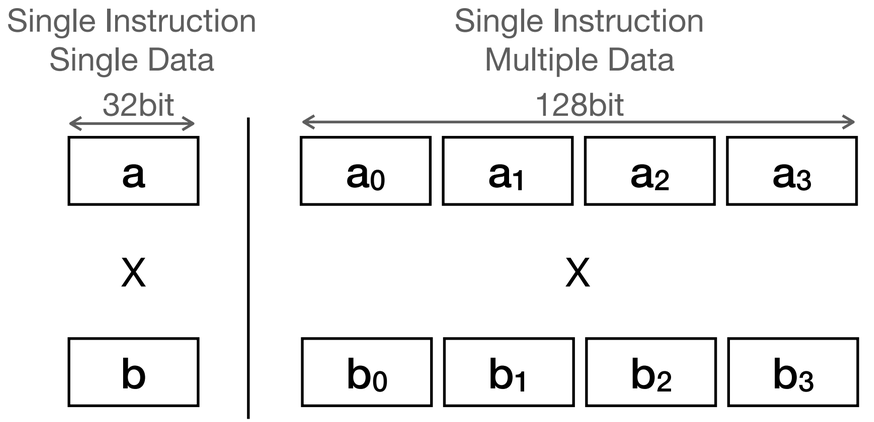
- **Complex Instruction Set Computer (CISC)**
  - Has many specialized instructions, some of which may only be rarely used in practical programs.
  - E.g., Intel x86
- **Reduced Instruction set Computer (RISC)**
  - Simplifies the processor by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines.
  - E.g., Arm, RISC-V
- **Example: For performing an ADD operation**
  - $C = A + B$
  - CISC may only need one instruction: $add\ a, b, c$
  - RISC may need four instructions: $load\ a, reg1$; $load\ b, reg2$; $add\ reg1 + reg2 = reg3$; $store\ reg3, c$

Instruction set architecture [Link]; CISC - Complex Instruction Set Computer [Link]; RISC - Reduced Instruction Set Computer [Link]

SIMD Programming

- SIMD (Single Instruction Multiple Data)
  - Perform operations on data vectors with a single instruction
  - Available on mostly all current processors

- With SIMD programming, we can
  - Exploit data-level parallelism in loops
  - Achieve speedup by quantization (e.g., int8, fp16)



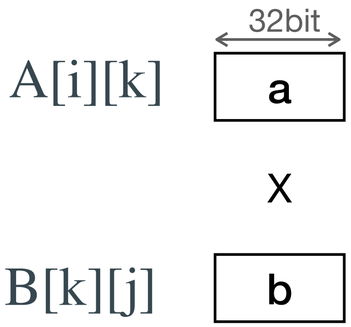
Difference between a conventional and SIMD instructions

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

SIMD Programming

```
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

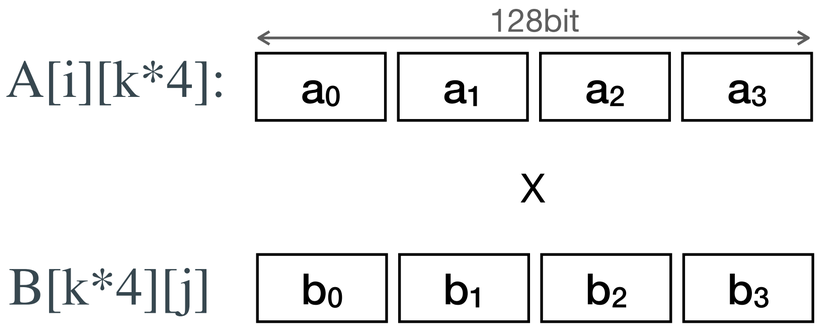
Arithmetic operations: N^3



128-bit SIMD instructions

```
for i in range(0, N):
    for k in range(0, N/4):
        for j in range(0, N):
            C[i][j] += dot_vec4(A[i][k*4], B[k*4][j])
```

Arithmetic operations: $N^3/4$



MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Optimization Techniques in TinyEngine

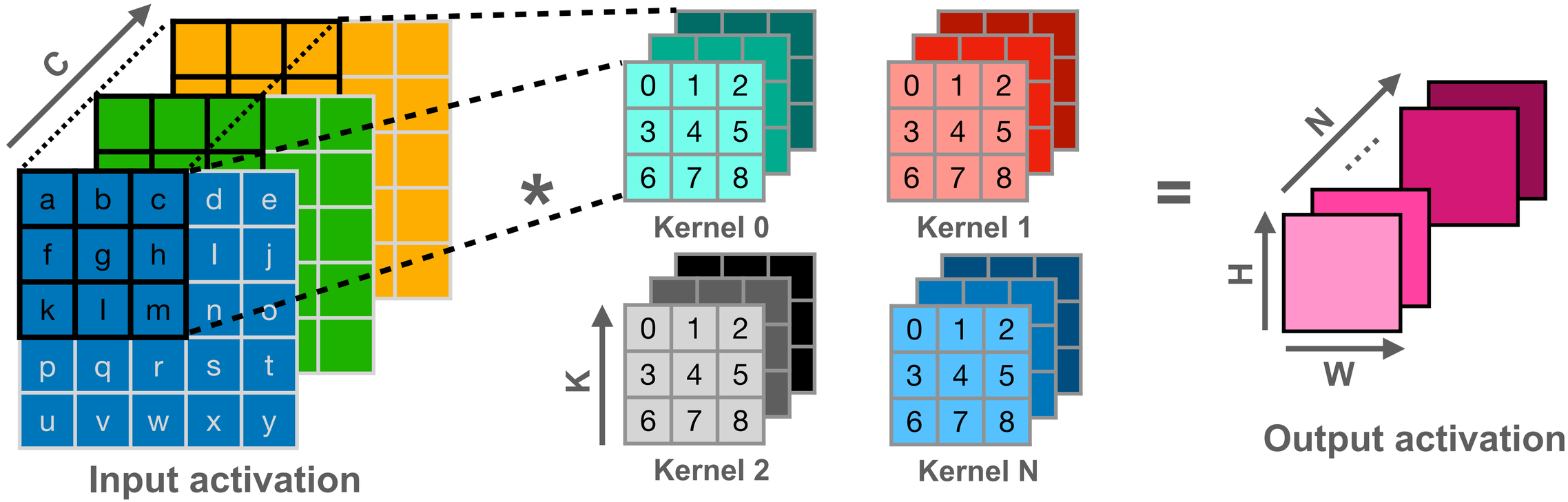
To enhance computing speed and reduce memory usage

- Loop unrolling:
  - A loop transformation technique that optimizes a program's execution speed at the expense of its binary size.
- Loop reordering:
  - A loop transformation technique that optimizes a program's execution speed by reordering the sequence of loops.
- Loop tiling:
  - A loop transformation technique that reduces memory access by partitioning a loop's iteration space into smaller chunks or blocks.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- **Image to Column (Im2col) convolution:**
  - Rearranges input data to directly utilize matrix multiplication kernels.
- In-place depth-wise convolution:
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
  - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
  - Reduce the number of multiplications to enhance computing speed for convolution.

Im2col Convolution

Image to Column Convolution

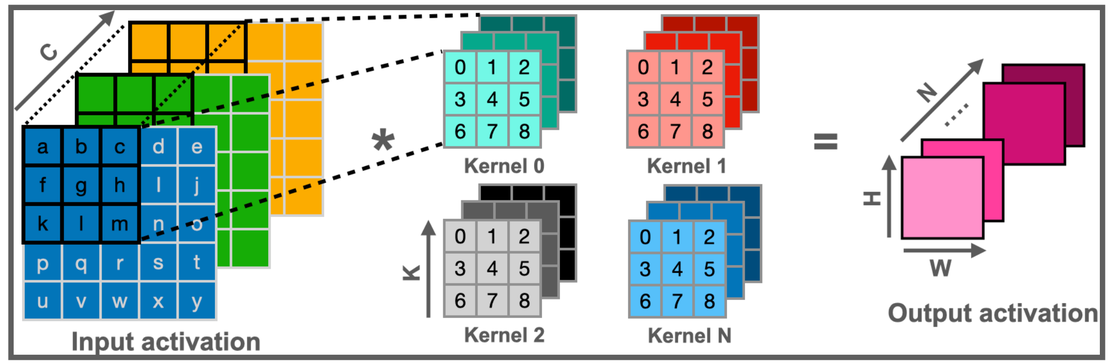
- Im2col is a technique to implement convolution with Generalized Matrix Multiplication (GEMM).

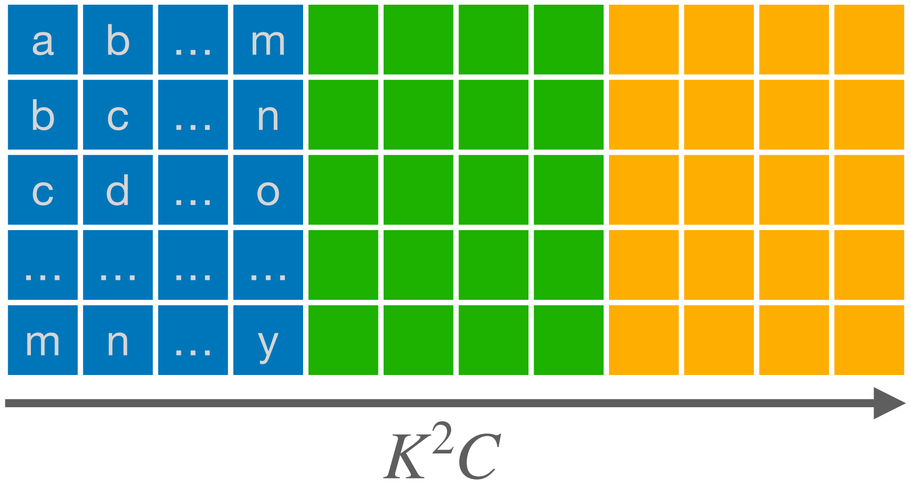


Anatomy of a High-Speed Convolution [Link]
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

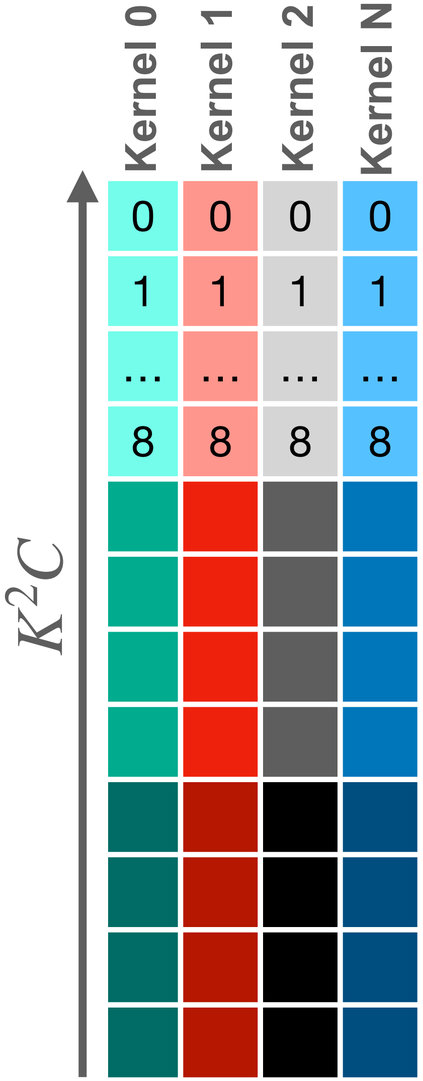
Im2col Convolution

Image to Column Convolution

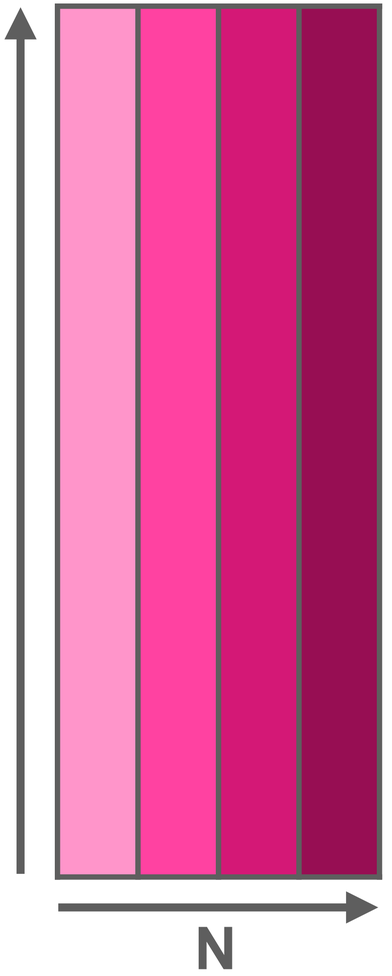




*



=



Im2col Convolution

Image to Column Convolution

- Im2col is a technique to convert the image in a form such that Generalized Matrix Multiplication (GEMM) calls for dot products.
- Pro:
  - Utilize GEMM for convolution
- Con:
  - Require additional memory space.
  - The implicit GEMM can solve the additional memory problem.
    - A variant of direct convolution, and operates directly on the input weight and activation tensors. [Link]

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Optimization Techniques in TinyEngine

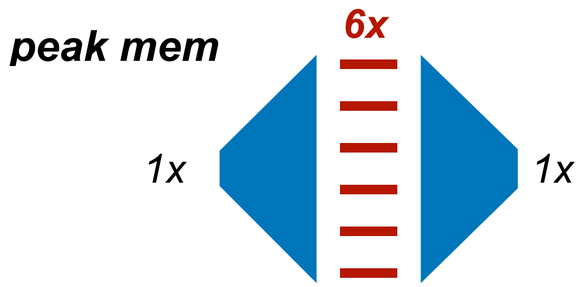
To enhance computing speed and reduce memory usage

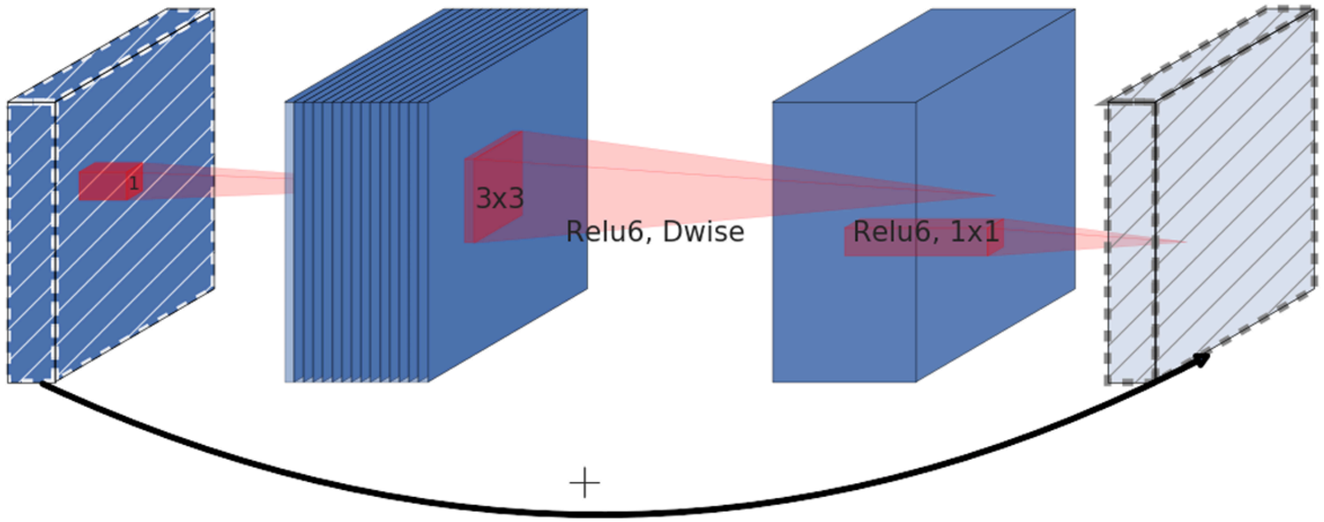
- Loop unrolling:
  - A loop transformation technique that optimizes a program's execution speed at the expense of its binary size.
- Loop reordering:
  - A loop transformation technique that optimizes a program's execution speed by reordering the sequence of loops.
- Loop tiling:
  - A loop transformation technique that reduces memory access by partitioning a loop's iteration space into smaller chunks or blocks.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Image to Column (Im2col) convolution:
  - Rearranges input data to directly utilize matrix multiplication kernels.
- **In-place depth-wise convolution:**
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
  - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
  - Reduce the number of multiplications to enhance computing speed for convolution.

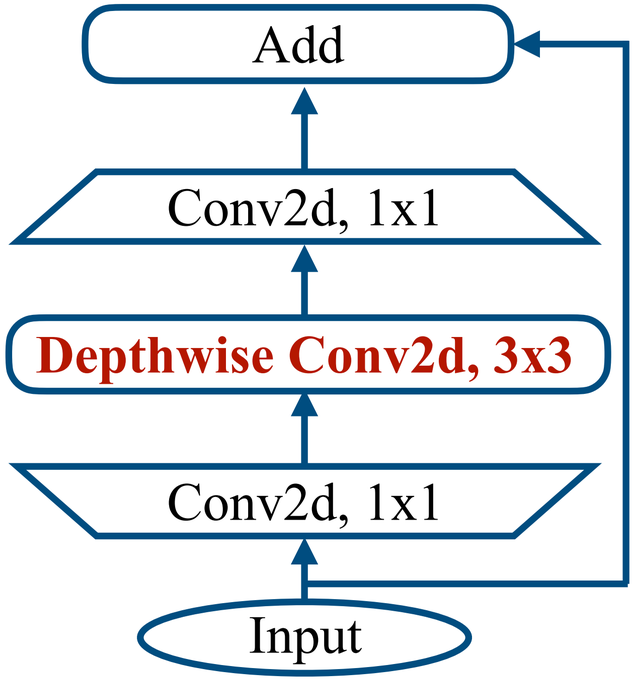
In-place Depth-wise Convolution

Inverted Residual Block

- Many popular neural network models, such as MobileNetV2, have “inverted residual blocks” with depth-wise convolutions which reduce model size and FLOPs, but significantly increase peak memory (3-6x).



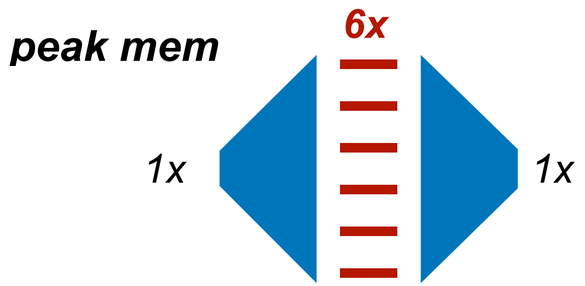




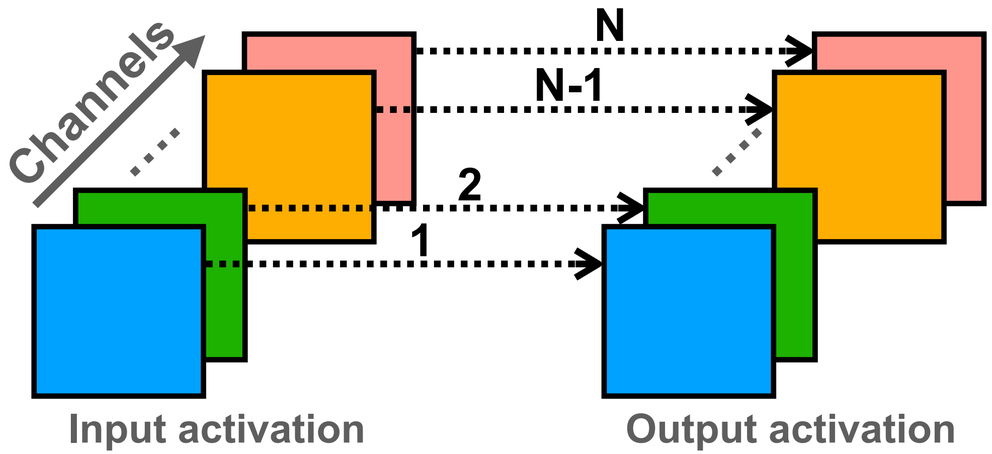
MobileNetV2: Inverted Residuals and Linear Bottlenecks [Sandler et al., CVPR 2018]
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

In-place Depth-wise Convolution

Reduce peak memory



- To reduce the peak memory of depth-wise convolution, we utilize the “in-place” updating policy with a temporary buffer.



General depth-wise convolution

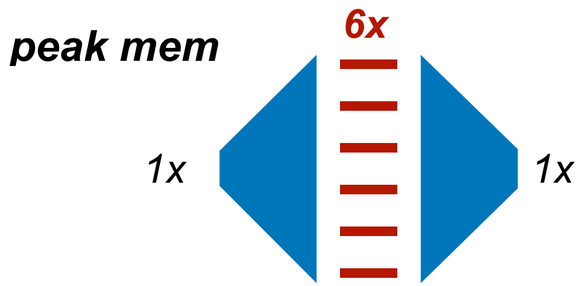
Peak Memory: 2xCxHxW

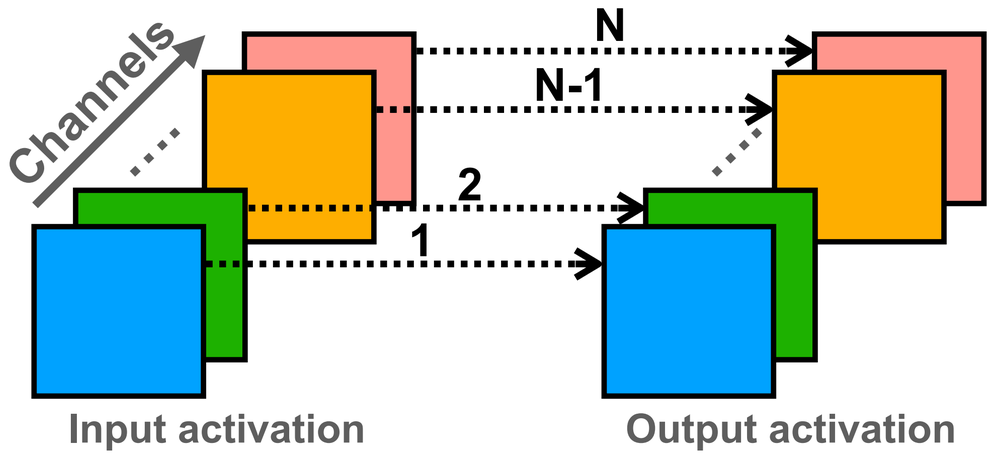
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

In-place Depth-wise Convolution

Reduce peak memory

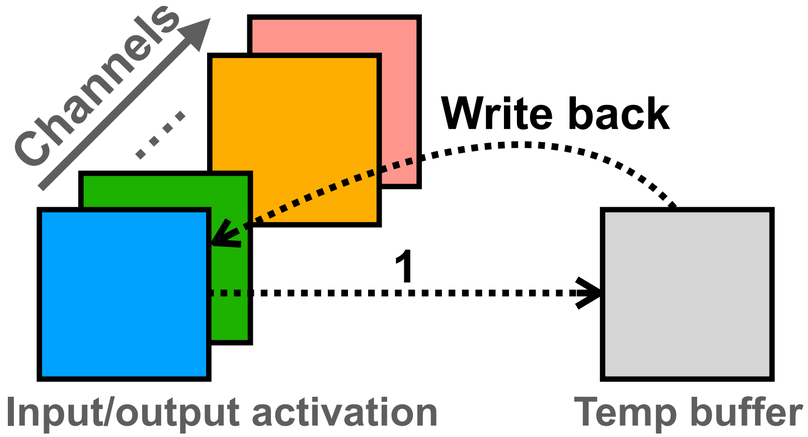
- To reduce the peak memory of depth-wise convolution, we utilize the “in-place” updating policy with a temporary buffer.





General depth-wise convolution

Peak Memory: 2xCxHxW



In-place depth-wise convolution

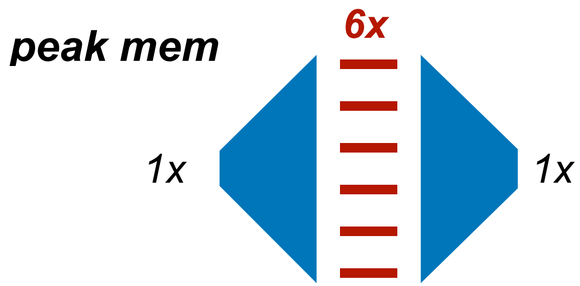
Peak Memory: (1+C)xHxW

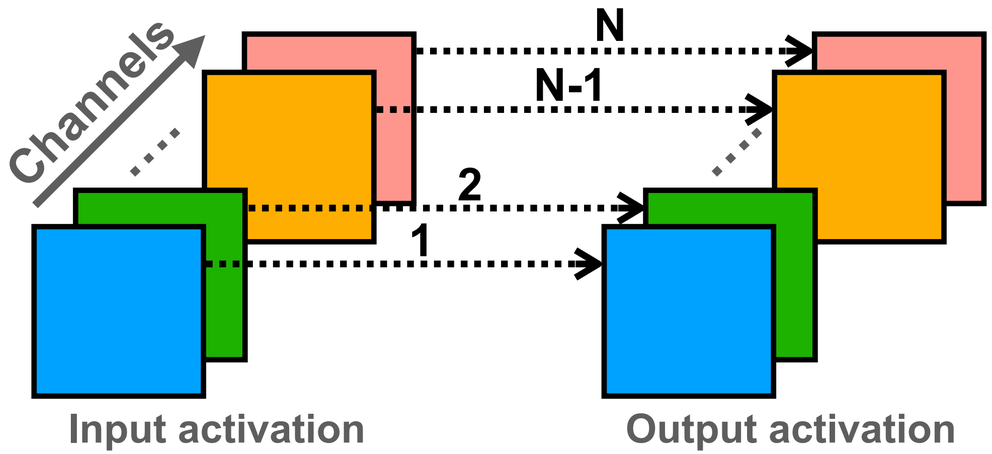
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

In-place Depth-wise Convolution

Reduce peak memory

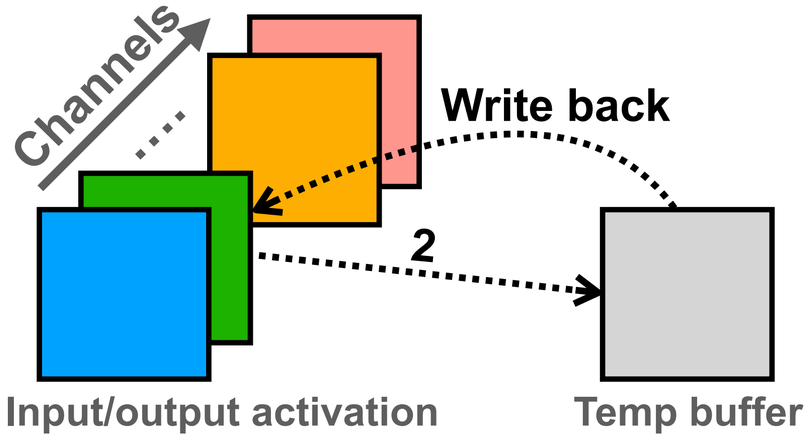
- To reduce the peak memory of depth-wise convolution, we utilize the “in-place” updating policy with a temporary buffer.





General depth-wise convolution

Peak Memory: 2xCxHxW



In-place depth-wise convolution

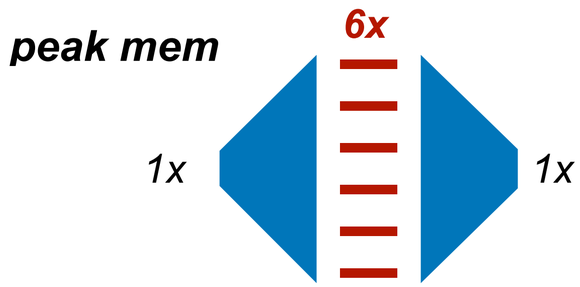
Peak Memory: (1+C)xHxW

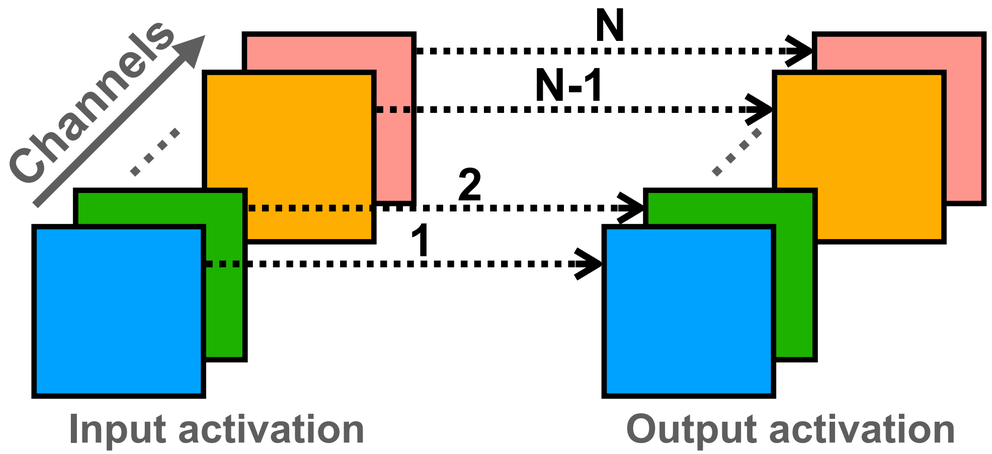
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

In-place Depth-wise Convolution

Reduce peak memory

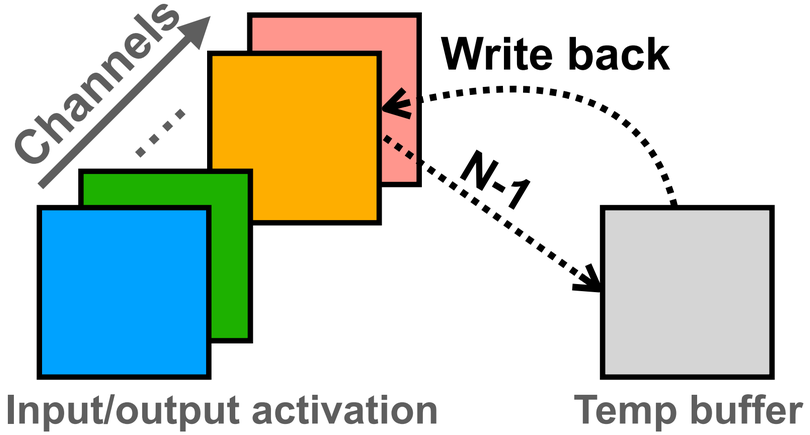
- To reduce the peak memory of depth-wise convolution, we utilize the “in-place” updating policy with a temporary buffer.





General depth-wise convolution

Peak Memory: 2xCxHxW



In-place depth-wise convolution

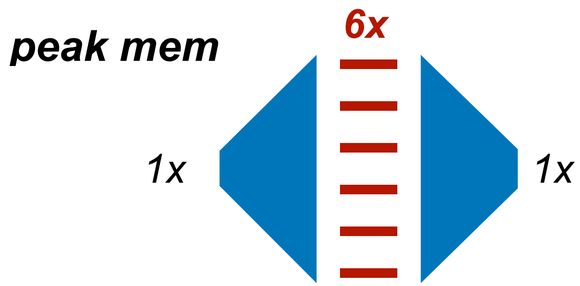
Peak Memory: (1+C)xHxW

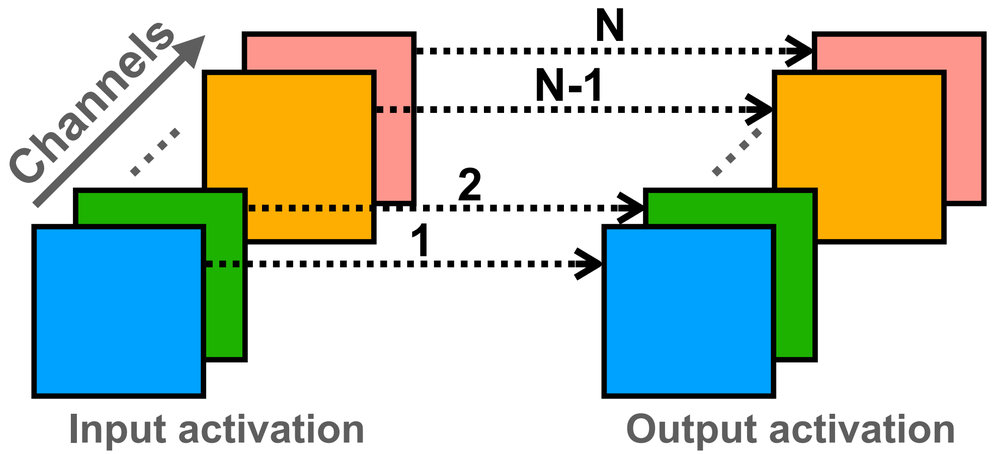
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

In-place Depth-wise Convolution

Reduce peak memory

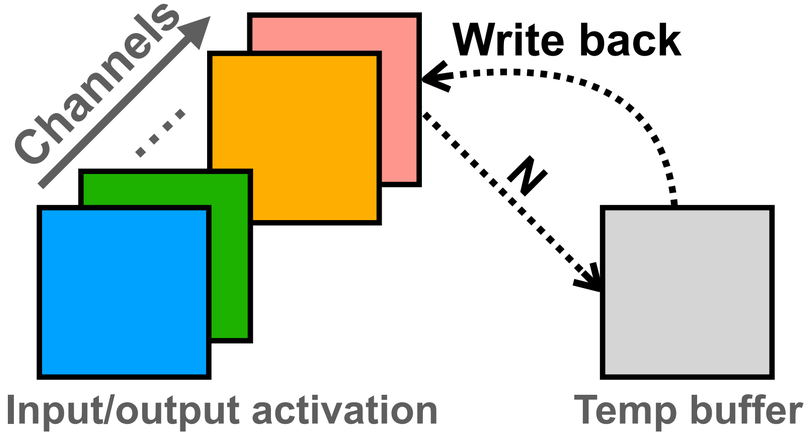
- To reduce the peak memory of depth-wise convolution, we utilize the “in-place” updating policy with a temporary buffer.





General depth-wise convolution

Peak Memory: 2xCxHxW



In-place depth-wise convolution

Peak Memory: (1+C)xHxW

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

Optimization Techniques in TinyEngine

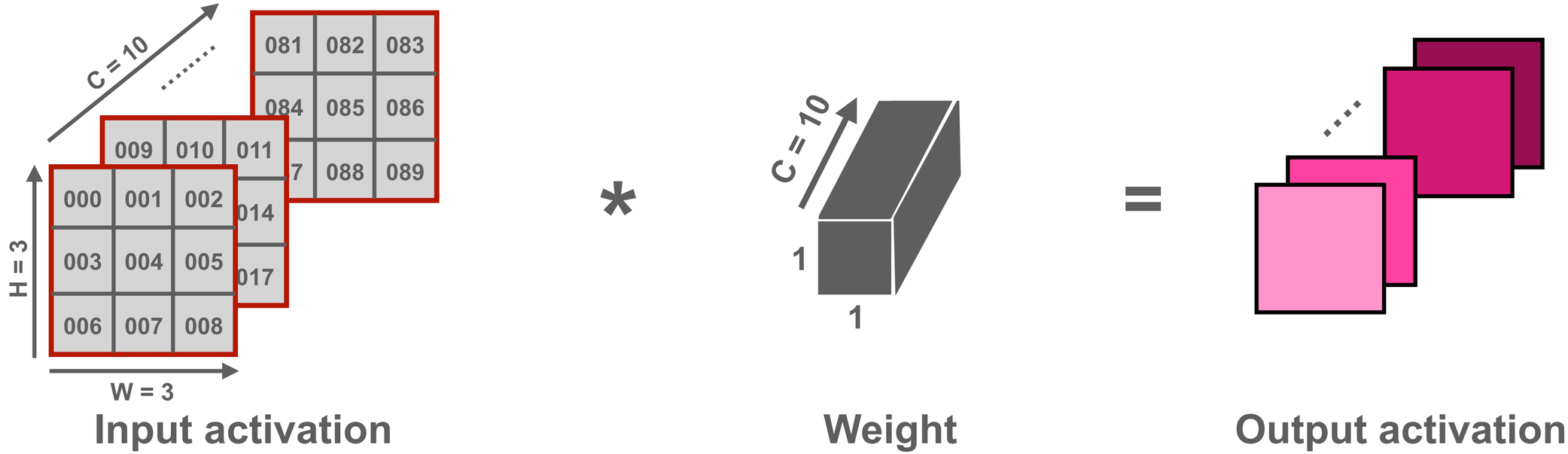
To enhance computing speed and reduce memory usage

- Loop unrolling:
  - A loop transformation technique that optimizes a program's execution speed at the expense of its binary size.
- Loop reordering:
  - A loop transformation technique that optimizes a program's execution speed by reordering the sequence of loops.
- Loop tiling:
  - A loop transformation technique that reduces memory access by partitioning a loop's iteration space into smaller chunks or blocks.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Image to Column (Im2col) convolution:
  - Rearranges input data to directly utilize matrix multiplication kernels.
- In-place depth-wise convolution:
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- **NHWC for point-wise convolution, and NCHW for depth-wise convolution:**
  - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
  - Reduce the number of multiplications to enhance computing speed for convolution.

How to Choose the Appropriate Data Layout

Use NHWC for Point-wise Convolution

- TinyEngine adopts the NHWC data layout for point-wise convolution.
  - Generally, NHWC would have better locality than NCHW for point-wise convolution due to more sequential access.



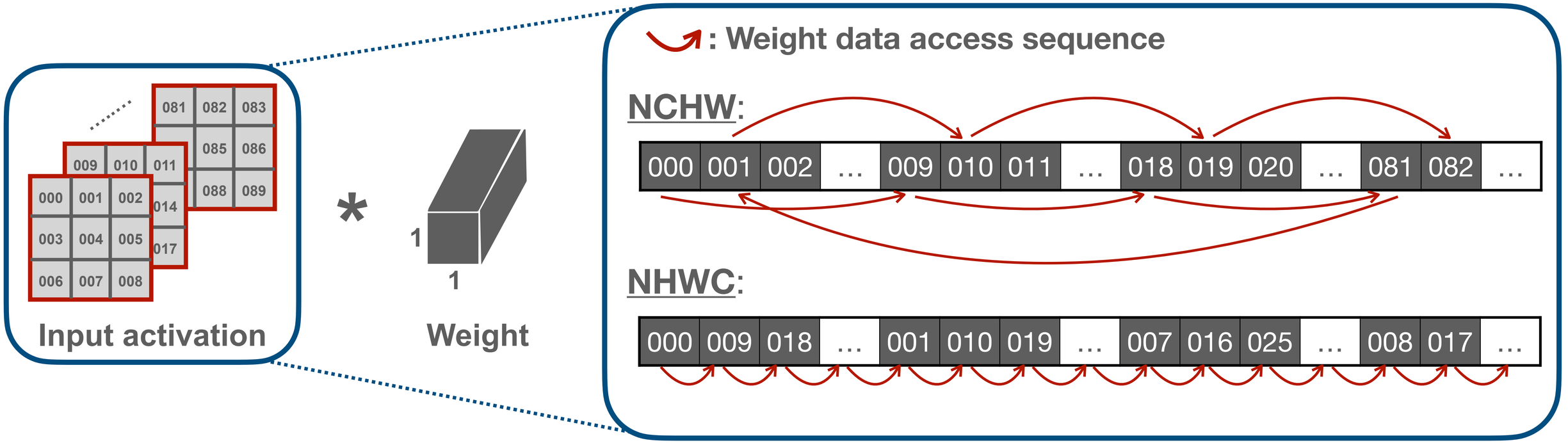
Input activation | Weight | Output activation

MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

How to Choose the Appropriate Data Layout

Use NHWC for Point-wise Convolution

- TinyEngine adopts the NHWC data layout for point-wise convolution.
  - Generally, NHWC would have better locality than NCHW for point-wise convolution due to more sequential access.

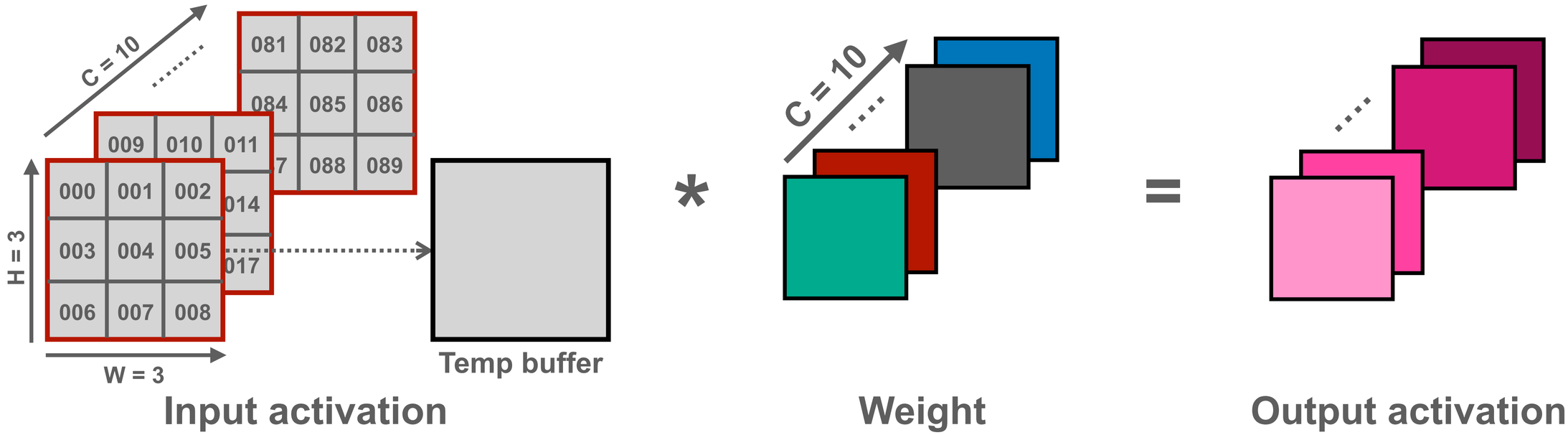


MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

How to Choose the Appropriate Data Layout

Use NCHW for Depth-wise Convolution

- TinyEngine adopts the in-place depth-wise convolution.
  - That is, we will access activation and conduct depth-wise convolution in the NCHW sequence.
  - NCHW would have better locality than NHWC for depth-wise convolutiondue to more sequential access.

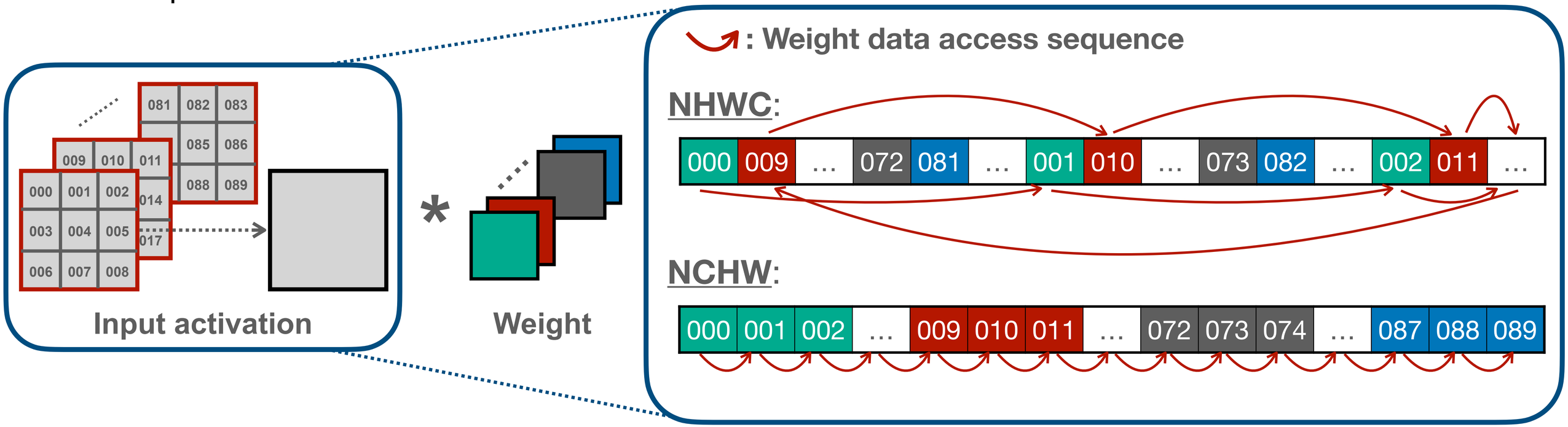


MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

How to Choose the Appropriate Data Layout

Use NCHW for Depth-wise Convolution

- TinyEngine adopts the in-place depth-wise convolution.
  - That is, we will access activation and conduct depth-wise convolution in the NCHW sequence.
  - NCHW would have better locality than NHWC for depth-wise convolutiondue to more sequential access.



MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

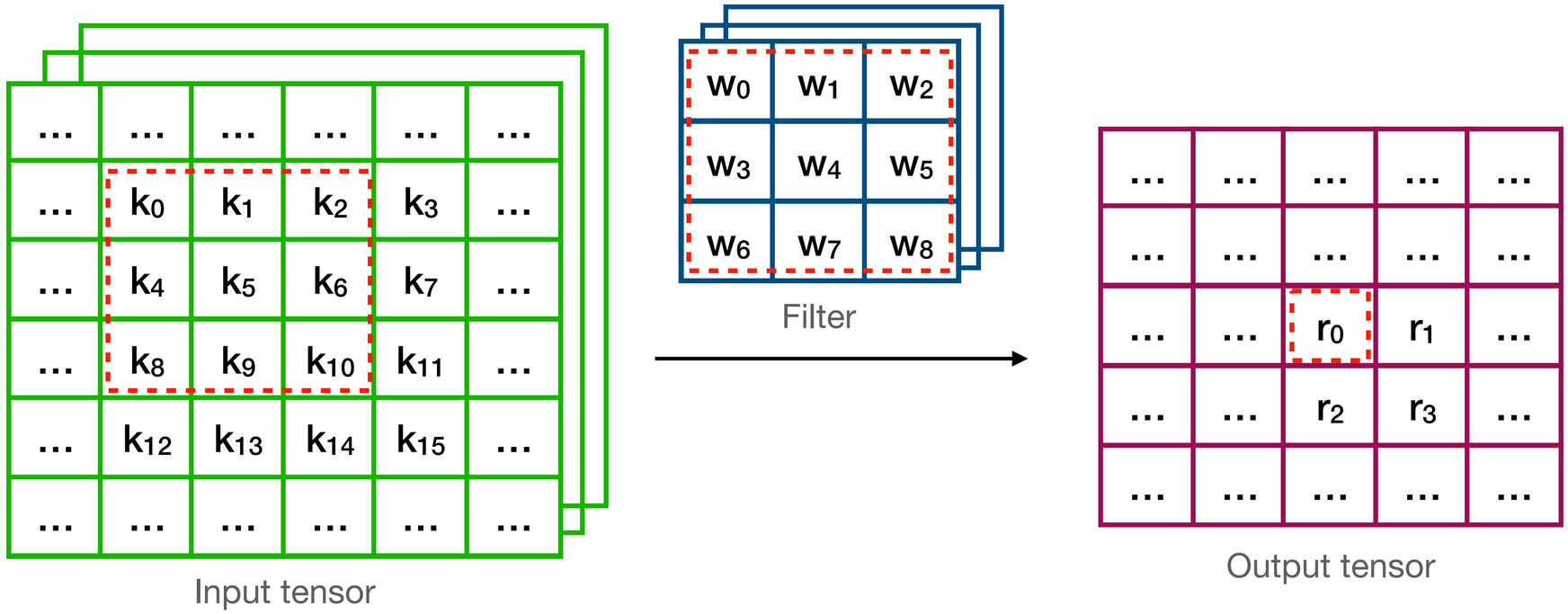
Optimization Techniques in TinyEngine

To enhance computing speed and reduce memory usage

- Loop unrolling:
  - A loop transformation technique that optimizes a program's execution speed at the expense of its binary size.
- Loop reordering:
  - A loop transformation technique that optimizes a program's execution speed by reordering the sequence of loops.
- Loop tiling:
  - A loop transformation technique that reduces memory access by partitioning a loop's iteration space into smaller chunks or blocks.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Image to Column (Im2col) convolution:
  - Rearranges input data to directly utilize matrix multiplication kernels.
- In-place depth-wise convolution:
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
  - Exploit the appropriate data layout for different types of convolution.
- **Winograd convolution:**
  - Reduce the number of multiplications to enhance computing speed for convolution.

Winograd Convolution

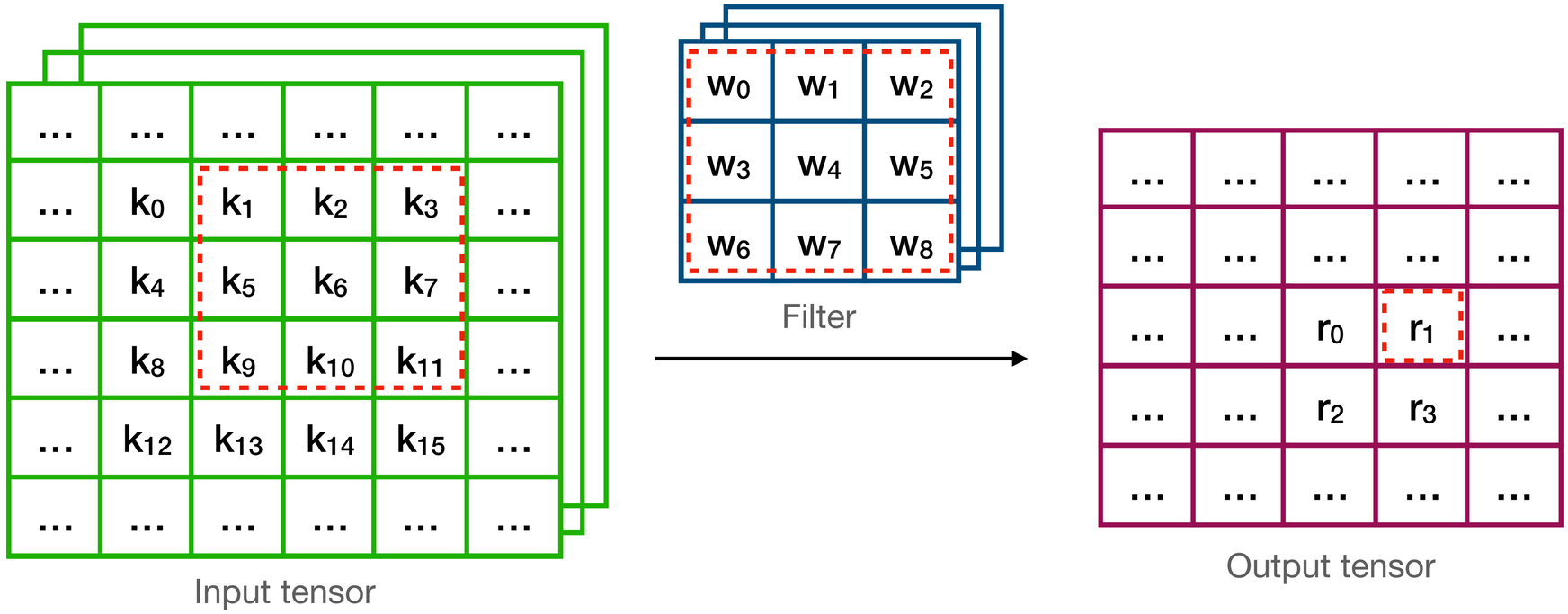
- Direct convolution need 9xCx4 MACs for 4 outputs



"Even Faster CNNs: Exploring the New Class of Winograd Algorithms," a Presentation from Arm

Winograd Convolution

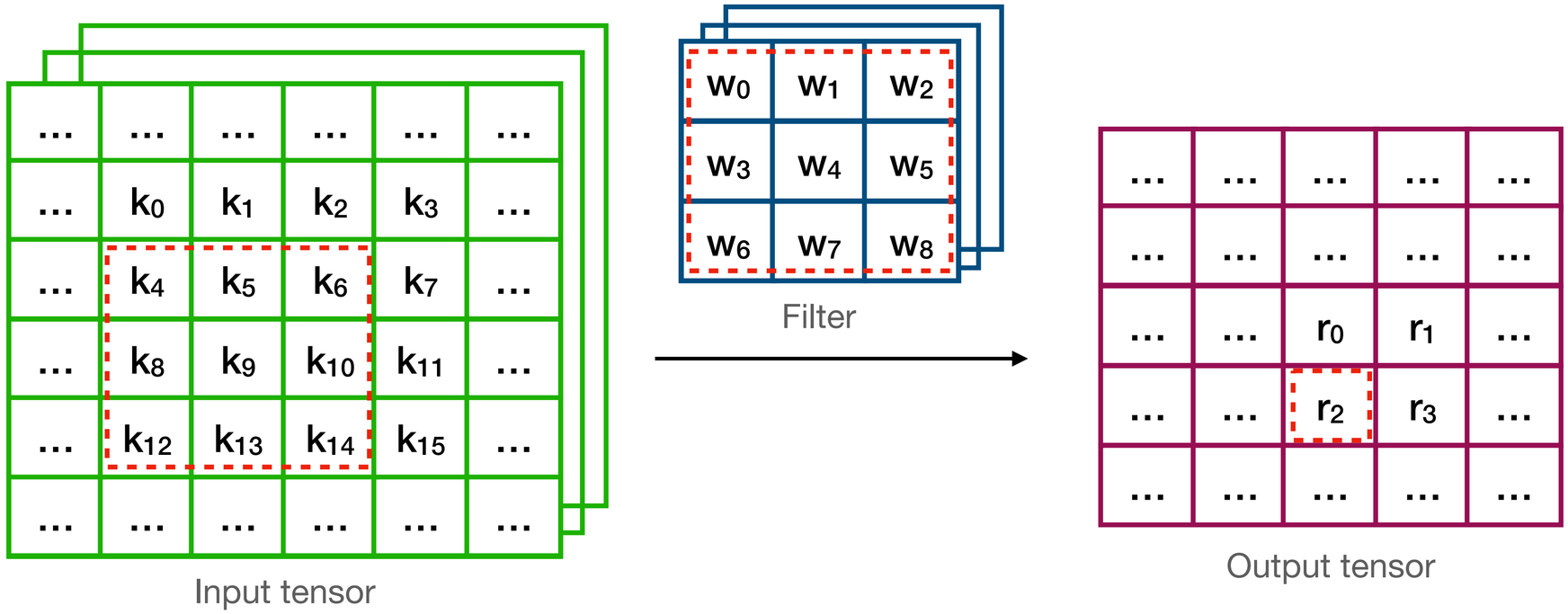
- Direct convolution need 9xCx4 MACs for 4 outputs



"Even Faster CNNs: Exploring the New Class of Winograd Algorithms," a Presentation from Arm

Winograd Convolution

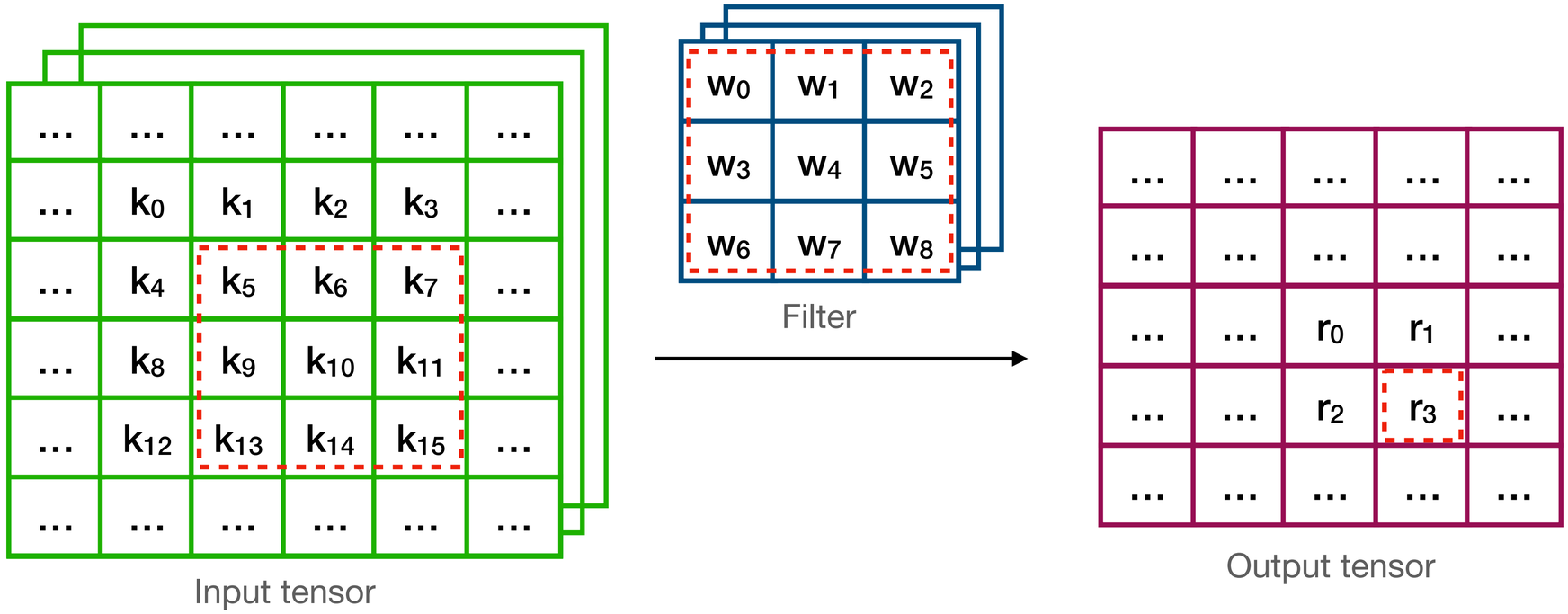
- Direct convolution need 9xCx4 MACs for 4 outputs



"Even Faster CNNs: Exploring the New Class of Winograd Algorithms," a Presentation from Arm

Winograd Convolution

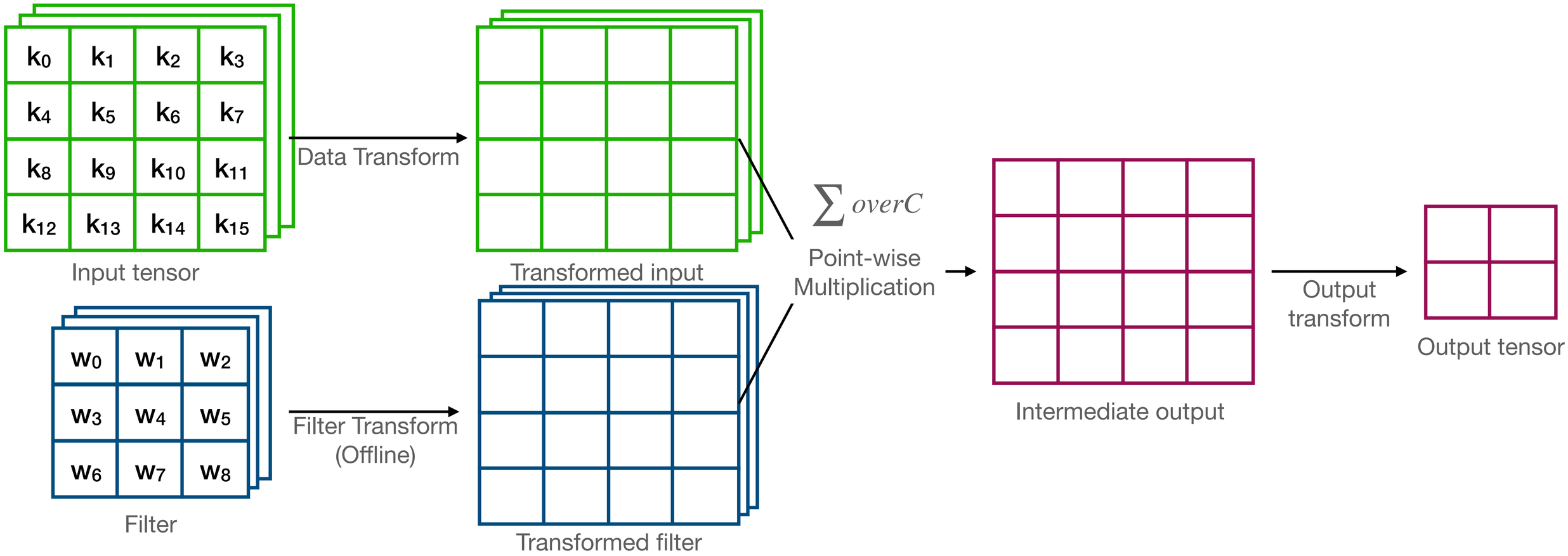
- Direct convolution need 9xCx4 MACs for 4 outputs



"Even Faster CNNs: Exploring the New Class of Winograd Algorithms," a Presentation from Arm

Winograd Convolution

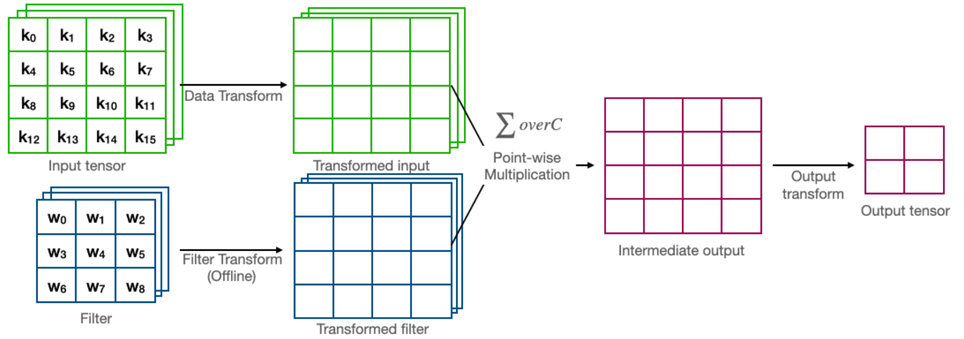
- Direct convolution need 9xCx4 MACs for 4 outputs
- Winograd convolution need only16xC MACs for 4 outputs -> 2.25x fewer MACs



"Fast Algorithms for Convolutional Neural Networks" [Link]

Winograd Convolution

Formula for 3x3 convolution



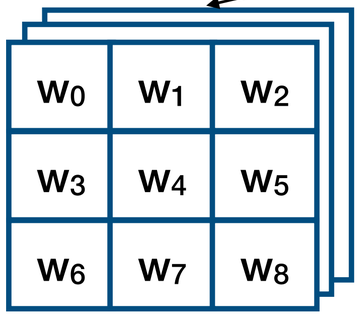
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix} \quad G^T = \begin{bmatrix} 1 & 1/2 & 1/2 & 0 \\ 0 & 1/2 & -1/2 & 0 \\ 0 & 1/2 & 1/2 & 1 \end{bmatrix} \quad B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Output transform (Online)

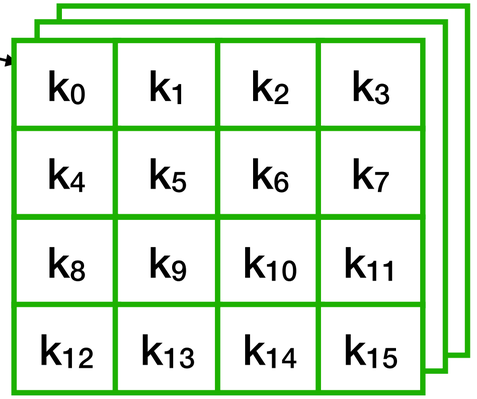
Output tensor ← $Y = A^T[[GgG^T] \odot [B^T dB]]A$

Filter Transform (Offline)

Data Transform (Online)



Filter



Input tensor

”Fast Algorithms for Convolutional Neural Networks” [Link]

Welcome to Try TinyEngine and MCUNet

Open Source on GitHub

Product Team Enterprise Explore Marketplace Pricing

mit-han-lab / mcunet Public

Code Issues 6 Pull requests Actions Projects Security Insights

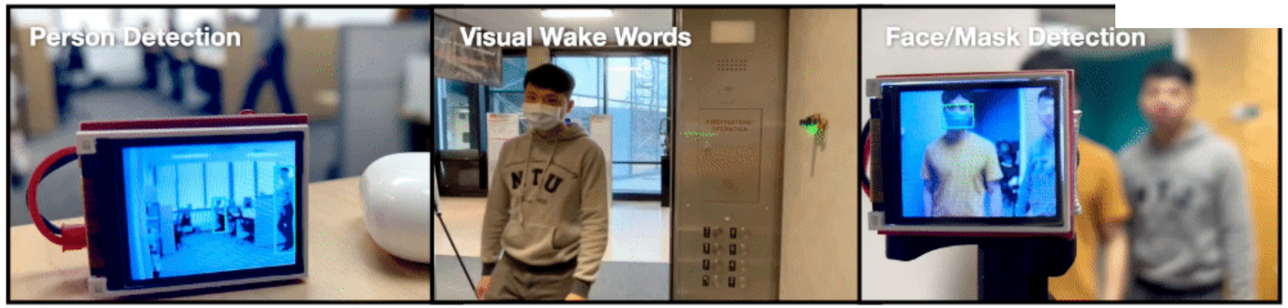
master 2 branches 0 tags Go to

README.md

MCUNet: Tiny Deep Learning on IoT Devices

This is the official implementation of the MCUNet series.

website | paper | paper (v2) | demo video



Product Team Enterprise Explore Marketplace Pricing Search

mit-han-lab / tinyengine Public

Code Issues Pull requests Actions Projects Security Insights

master 1 branch 0 tags Go to file Code

README.md

TinyEngine

This is the official implementation of TinyEngine, a memory-efficient and high-performance neural network library for Microcontrollers. TinyEngine is a part of MCUNet, which also consists of TinyNAS. MCUNet is a system-algorithm co-design framework for tiny deep learning on microcontrollers. TinyEngine and TinyNAS are co-designed to fit the tight memory budgets.

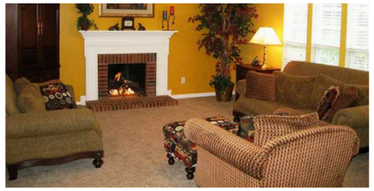
The MCUNet and TinyNAS repo is here.

MCUNetV1 | MCUNetV2 | MCUNetV3

"TinyEngine" on GitHub [Link]; "MCUNet" on GitHub [Link]

Summary of Today's Lecture

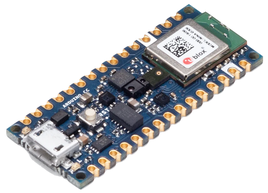
(a) 'Person'

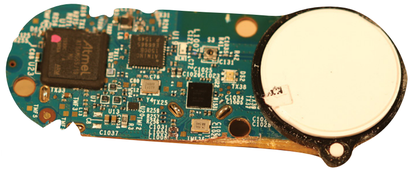


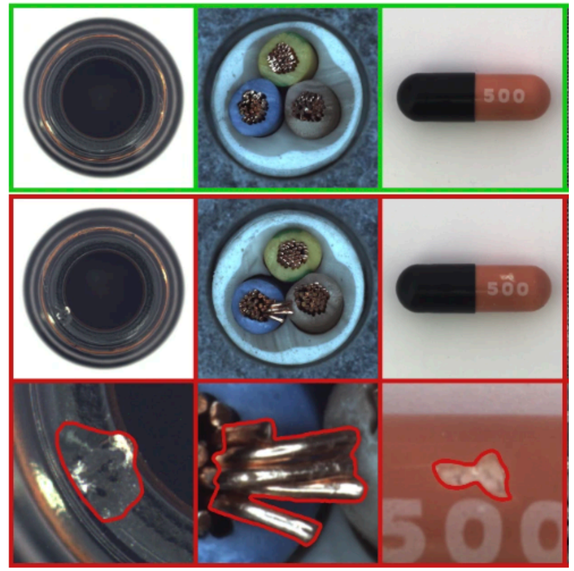
(b) 'Not-person'

Today we learned:

1. What microcontrollers are and why they are essential.
2. Important characteristics and components of microcontrollers.
3. Critical data layouts/formats of neural networks on microcontrollers.
   - NHWC and NCHW
4. The eight optimization techniques used in TinyEngine.
   - Loop unrolling, loop reordering, loop tiling, SIMD programming, Im2col convolution, in-place depth-wise convolution, NHWC for point-wise convolution and NCHW for depth-wise convolution, and Winograd convolution
5. How the optimization techniques work.



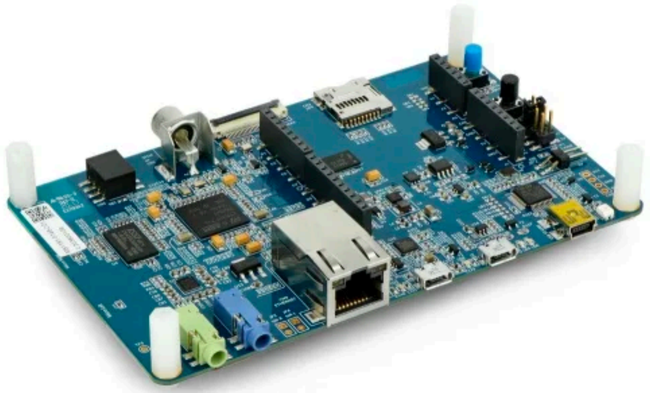


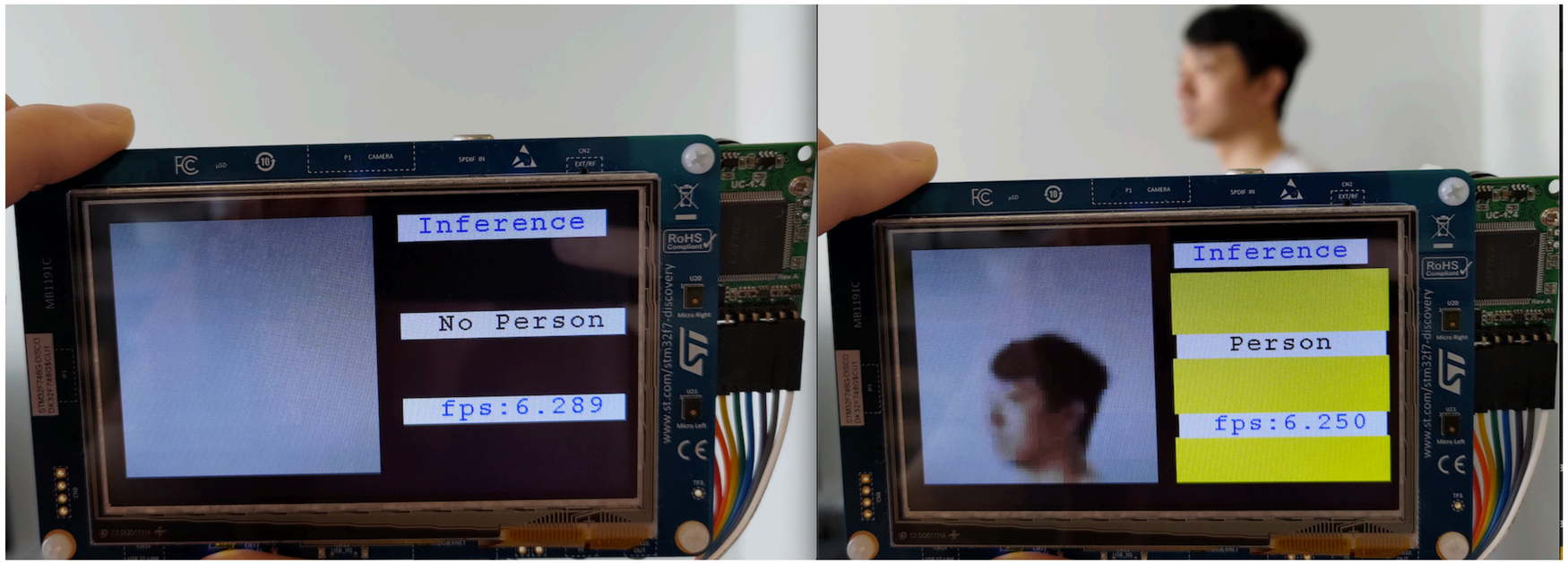


Lab 4 - Deployment on MCU with TinyEngine

In Lab 4, we will:

1. Learn how to visualize deep learning models.
2. Deploy a visual wake words (VWW) model on MCU (STM32F746G-DISCO board).
3. Evaluate on-device latency and memory footprint improvement achieved by different optimization techniques (loop unrolling/reordering, SIMD, Im2col, in-place depth-wise, NHWC/NCHW).
4. Learn how to implement the optimization techniques by tracing the codebase of TinyEngine.





STM32F746G-Disco [Link]