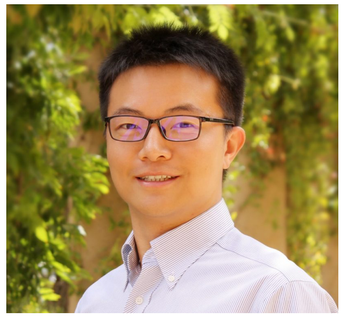
MIT HAN LAB

EfficientML.ai Lecture 11
TinyEngine and Parallel Processing



Song Han

Associate Professor, MIT
Distinguished Scientist, NVIDIA

@SongHan_MIT

Lecture Plan

Deploy Neural Networks on Edge Devices

Today we will learn:

1. **Edge AI and why they are essential.**
   - Important characteristics and components of microcontrollers.
   - Why deploying neural networks on microcontrollers is challenging?

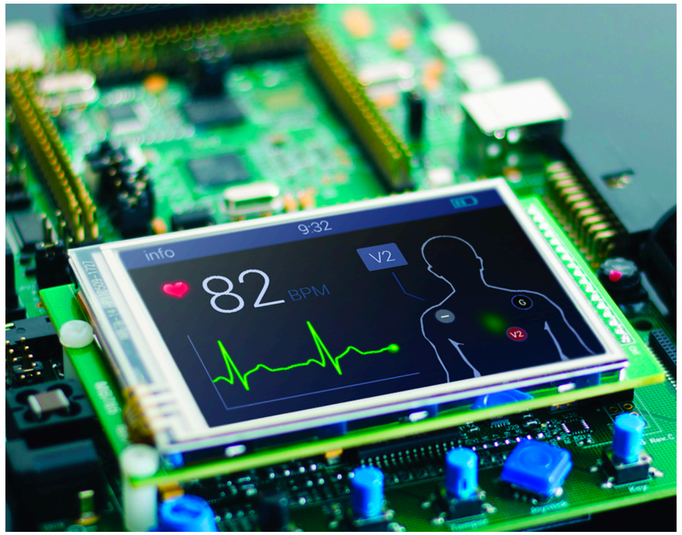
2. **Parallel computing techniques.**
   - Loop optimization, Multithreading, SIMD programming, CUDA programming

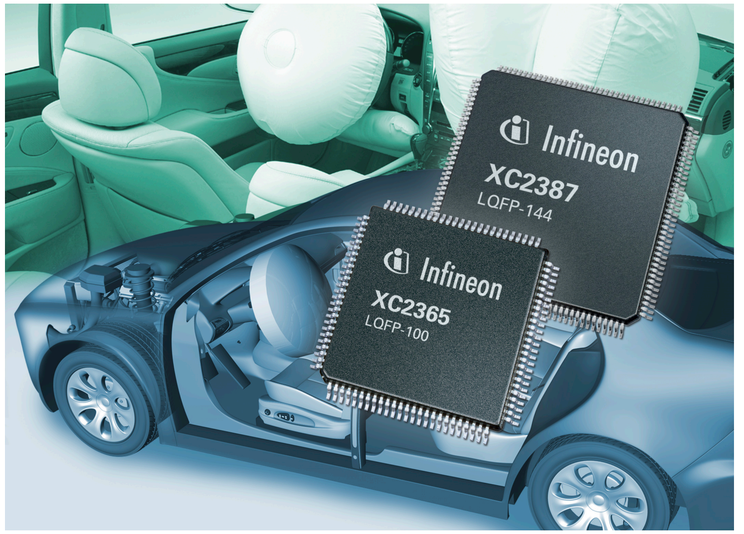
3. **Inference optimizations**
   - Im2col convolution, in-place depth-wise convolution, memory layout, and Winograd convolution.

Introduction to Edge AI

Ubiquitous Edge Devices

- Edge devices are ubiquitous and power wide applications in our daily lives:
  - Smartphones
  - Vehicles
  - Robots
  - Office machines
  - Medical devices
  - Mobile radio transceivers
  - Vending machines
  - Home appliances

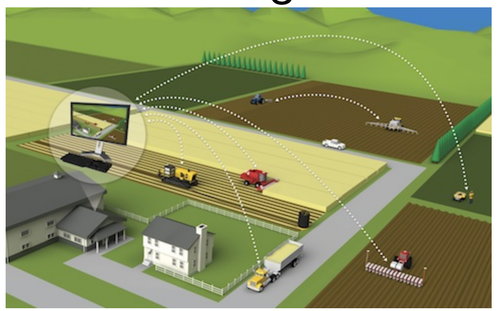




Smart Retail

Personalized Healthcare

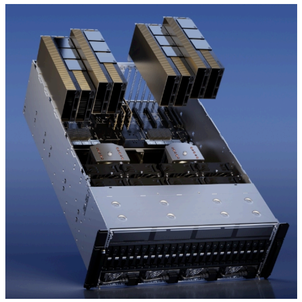
Precision Agriculture



Smart Home

Microcontroller [Link], Infineon's microcontroller [Link], Amazon Alexa [Link], Caliber Interconnect [Link]

Challenge: Limited Resource on Edge Devices

	Cloud AI		Mobile AI	Tiny AI
				
	Nvidia H100[1]	Apple M2 Ultra[2]	Qualcomm S8Gen2[3]	STM32F746NG[4]
Memory	80GB	64-192 GB	8-24 GB	320kB
Storage	~TB/PB	~GB/TB	~GB	1MB
Computing power	1,979 TOPS	31.6 TOPS	36 TOPS	462 MOPS

Nvidia H100 [1], Apple M2 Ultra [2], S8Gen2 [3], STM32F746NG [4]

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]

Section 2: Parallel Computing Techniques

Parallel Computing Techniques

"Parallel computing tutorial [Link]

To enhance computing speed and reduce memory usage

- Loop optimization: Optimize locality and reduce branching overhead
  - **Loop reordering: Optimizes locality by reordering the sequence of loops.**
  - Loop tiling: Reduces memory access by partitioning a loop's iteration space.
  - Loop unrolling: reduces branching overhead at the expense of its binary size.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Multithreading:
  - Concurrent execution of multiple threads within a single process.
- CUDA programming:
  - Use GPUs to accelerate computation.

“Loop reordering example code” [Link]

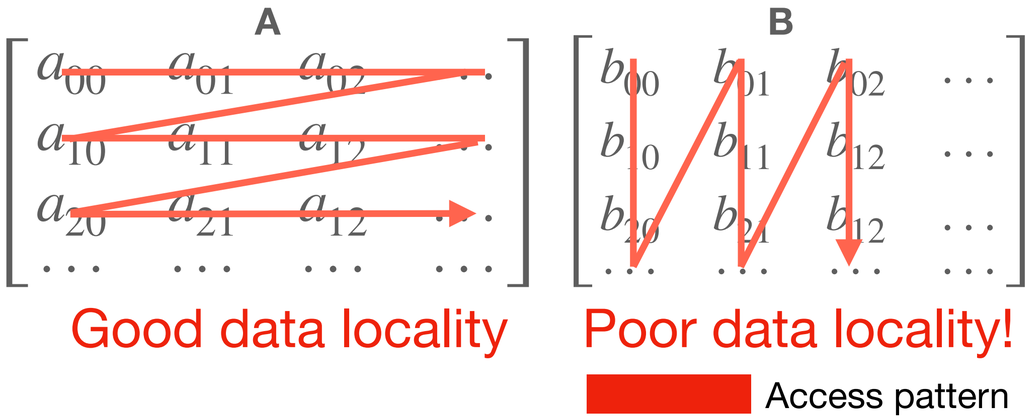
Loop Reordering

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 example code” [Link]

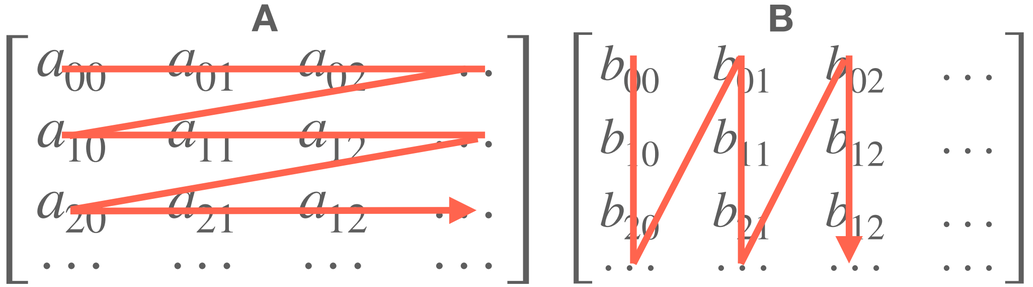
Loop Reordering

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]
```

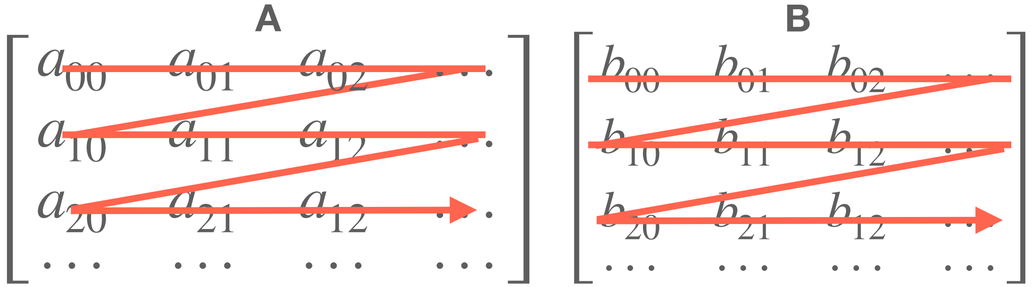
Reorder (i, j, k) as (i, k, j)

```
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]

“Loop reordering example code” [Link]

Loop Reordering

Speed up Matrix Multiplication

```
void MatmulOperator::naive_mat_mul(const struct matmul_params *params)
{
  int i, j, k;
  // … Get A[][], B[][], C[][] from params

  for (i = 0; i < C->row; i++)
    for (j = 0; j < C->column; j++){
      float acc = 0;
      for (k = 0; k < A->column; k++)
        acc += A[i][k] * B[k][j];
      C[i][j] = acc;
    }
}
```

```
void MatmulOperator::mat_mul_reordering(const struct matmul_params *params)
{
  int i, j, k;
  // … Get A[][], B[][], C[][] from params

  for (i = 0; i < C->row; i++)
    for (k = 0; k < A->column; k++)
    {
      float acc = 0;
      float Aik = A[i][k];
      for (j = 0; j < C->column; j++)
        acc += Aik * B[k][j];
      C[i][j] = acc;
    }
}
```

Without Loop reordering:

```
naive_mat_mul: 24296 ms
```

With Loop reordering:

```
mat_mul_reordering: 1979 ms
```

12x speed up

*Results are measured on Intel Xeon 4114

“Parallel computing tutorial [Link]

Parallel Computing Techniques

To enhance computing speed and reduce memory usage

- Loop optimization: Optimize locality and reduce branching overhead
  - Loop reordering: Optimizes locality by reordering the sequence of loops.
  - **Loop tiling: Reduces memory access by partitioning a loop's iteration space.**
  - Loop unrolling: reduces branching overhead at the expense of its binary size.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Multithreading:
  - Concurrent execution of multiple threads within a single process.
- CUDA programming:
  - Use GPUs to accelerate computation.

Parallel computing tutorial [Link]

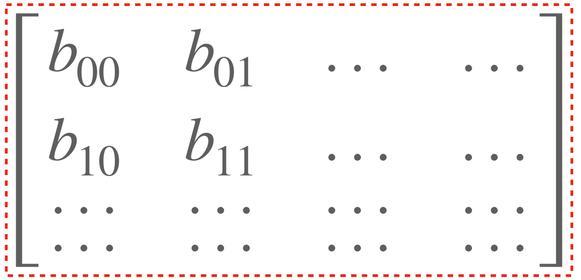
Loop Tiling

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]

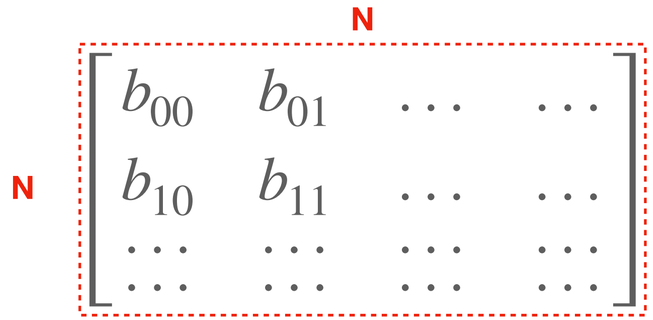
Loop Tiling

Reduce cache miss

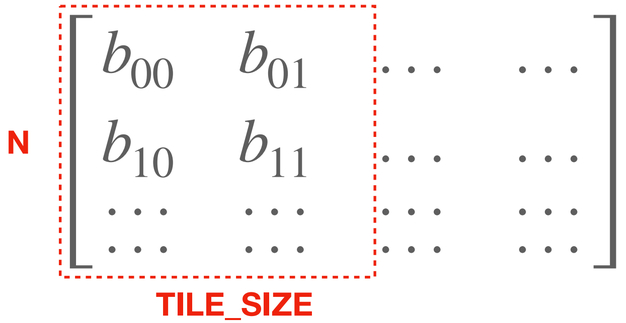
```
for i in range(0, N):
    for k in range(0, N):
        for j in range(0, N):
            C[i][j] += A[i][k] * B[k][j]
```

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^2



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_SIZE²

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_SIZE² → 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]

“Loop tiling example code” [Link]

Loop Tiling

Speed up Matrix Multiplication

```
#define BLK_SIZE 32
void MatmulOperator::mat_mul_tiling(const struct matmul_params *params)
{
  int i, j, k;
  // … Get A[][], B[][], C[][] from params

  for (ti = 0; ti < C->row; ti += BLK_SIZE){ // … Tile i by BLK_SIZE
    for (tk = 0; tk < A->column; tk += BLK_SIZE){
      for (tj = 0; tj < C->column; tj += BLK_SIZE){
        for (i = ti; i < ti + BLK_SIZE; I++){
          for (k = tk; k < tk + BLK_SIZE; k++){
            Aik = data_A[i * A->column + k];
            for (j = tj; j < tj + BLK_SIZE; j++){
              data_C[i * C->column + j] += Aik * data_B[k * B->column +
j];
            }
          }
        }
      }
    }
  }
}
```

Naive implementation: `naive_mat_mul: 24296 ms`

Loop tiling: `mat_mul_tiling: 1269 ms`

19x speed up

*Results are measured on Intel Xeon 4114

Parallel Computing Techniques

"Parallel computing tutorial [Link]

To enhance computing speed and reduce memory usage

- Loop optimization: Optimize locality and reduce branching overhead
  - Loop reordering: Optimizes locality by reordering the sequence of loops.
  - Loop tiling: Reduces memory access by partitioning a loop's iteration space.
  - **Loop unrolling: reduces branching overhead at the expense of its binary size.**
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Multithreading:
  - Concurrent execution of multiple threads within a single process.
- CUDA programming:
  - Use GPUs to accelerate computation.

“Loop unrolling example code” [Link]

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 example code” [Link]

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]

"Loop unrolling example code" [Link]

Loop Unrolling

Speed up Matrix Multiplication

```
void MatmulOperator::mat_mul_unrolling(const struct matmul_params *params)
{
  int i, j, k;
  // … Get A[][], B[][], C[][] from params

  for (i = 0; i < C->row; I++){
    for (j = 0; j < C->column; j += 8){ // unroll j by 8
      float[8] acc = 0;
      for (k = 0; k < A->column; k += 4){ // unroll k by 4
        acc[0] += A[i][k] * B[k][j];
        acc[0] += A[i][k+1] * B[k+1][j];
        acc[0] += A[i][k+2] * B[k+2][j];
        acc[0] += A[i][k+3] * B[k+3][j];

        acc[1] += A[i][k] * B[k][j+1];
        acc[1] += A[i][k+1] * B[k+1][j+1];
        acc[1] += A[i][k+2] * B[k+2][j+1];
        acc[1] += A[i][k+3] * B[k+3][j+1];
        // …
      }
      C[i][j] = acc[0];
      C[i][j+1] = acc[1];
      // …
    }
  }
}
```

Code [Link]

Naive implementation: `naive_mat_mul: 24296 ms`

Unrolling: `mat_mul_unrolling: 8512 ms`

2.85x speed up

*Results are measured on Intel Xeon 4114

Parallel Computing Techniques

"Parallel computing tutorial [Link]

To enhance computing speed and reduce memory usage

- Loop optimization: Optimize locality and reduce branching overhead
  - Loop reordering: Optimizes locality by reordering the sequence of loops.
  - Loop tiling: Reduces memory access by partitioning a loop's iteration space.
  - Loop unrolling: reduces branching overhead at the expense of its binary size.
- **SIMD (single instruction, multiple data) programming:**
  - Performs the same operation on multiple data points simultaneously.
- Multithreading:
  - Concurrent execution of multiple threads within a single process.
- CUDA programming:
  - Use GPUs to accelerate computation.

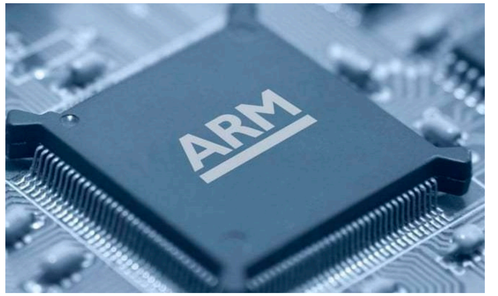
Introduction to Instruction Set

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



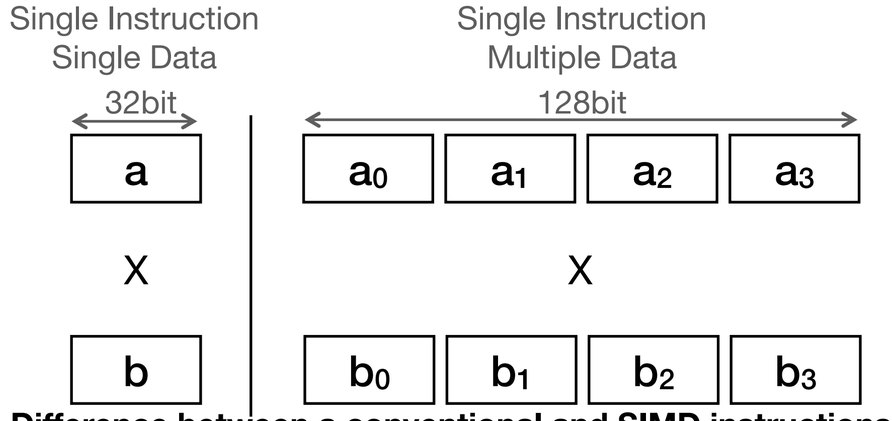
Instruction set architecture (ISA)

- **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)
  - A parallel processing paradigm that applies a single instruction to multiple data elements simultaneously.
  - Commonly used in modern processors to exploit data-level parallelism.
- Key Features:
  - Vector Registers:
    - Specialized registers that can hold and process multiple data elements.
  - Vector Operations:
    - Arithmetic and logical operations that work on entire vectors.
- With SIMD programming, we can
  - Increase computational throughput and speed
  - Improve energy efficiency



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

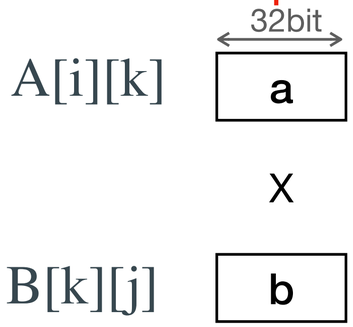
Example: SSE and NEON intrinsics

- SSE: _mm_load_ps/_mm_mul_ps/_mm_add_ps
  - mm: multimedia
  - load/mul/add: load/multiply/add
  - ps: packed single-precision

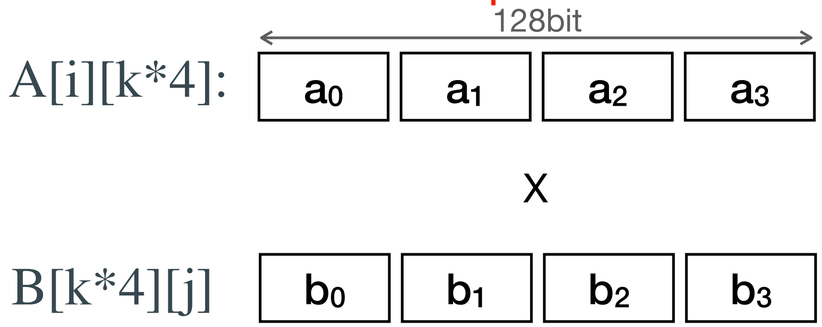
- NEON: vld1q_f32/vmulq_f32/vaddq_f32:
  - v: vector
  - ld/mul/add: load/multiply/add
  - 1: number of vector
  - q: quadword

```
// SISD programming
for k in range(0, N):
    C += A[k] * B[k]

// with SSE
for k in range(0, N/4):
    C += _mm_mul_ps(_mm_load_ps(A[k*4]), _mm_load_ps(B[k*4]))

// with NEON
for k in range(0, N/4):
    C += vmulq_f32(vld1q_f32(A[k*4]), vld1q_f32(B[k*4]))
```

Arithmetic operations: N^3



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]

SIMD Programming

Speed up Matrix Multiplication on x86 and ARM

SSE intrinsics

```
preprocessing(); // Initialize A, B, C and transpose B as transpose_tmp

for (i = 0; i < C->row; i++)
  for (j = 0; j < C->column; j++) {
    float accumulators[4] = {0, 0, 0, 0};
    __m128 *acc = (__m128*)accumulators; // initialize four 32-bit accumulators
    for (k = 0; k < A->column; k += 4) {
      // val[0:4] = A[i][k:k+4] * A[j][k:k+4]
      __m128 val = _mm_mul_ps(_mm_load_ps(&A[i][k]), _mm_load_ps(&transpose_tmp[j][k]));
      // accumulators[0:4] = accumulators[0:4] + val[0:4];
      *acc = _mm_add_ps(*acc, val);
    }
    c[i][j] = accumulators[0] + accumulators[1] + accumulators[2] + accumulators[3];
  }
```

```
_mm_load_ps/_mm_mul_ps/_mm_add_ps:
  • mm: multimedia
  • load/mul/add: load/multiply/add
  • ps: packed single-precision
```

ARM NEON intrinsics

```
preprocessing(); // Initialize A, B, C and transpose B as transpose_tmp

for (i = 0; i < C->row; i++)
  for (j = 0; j < C->column; j++) {
    float accumulators[4] = {0, 0, 0, 0};
    float32x4_t *acc = (float32x4_t*)accumulators; // initialize four 32-bit accumulators
    for (k = 0; k < A->column; k += 4) {
      // val[0:4] = A[i][k:k+4] * A[j][k:k+4]
      float32x4_t val = vmulq_f32(vld1q_f32(&A[i][k]), vld1q_f32(&transpose_tmp[j][k]));
      // accumulators[0:4] = accumulators[0:4] + val[0:4];
      *acc = vaddq_f32(*acc, val);
    }
    c[i][j] = accumulators[0] + accumulators[1] + accumulators[2] + accumulators[3];
  }
```

```
vld1q_f32/vmulq_f32/vaddq_f32:
  • v: vector
  • ld/mul/add: load/multiply/add
  • 1: number of vector
  • q: quadword
```

“Parallel computing tutorial [Link]

Parallel Computing Techniques

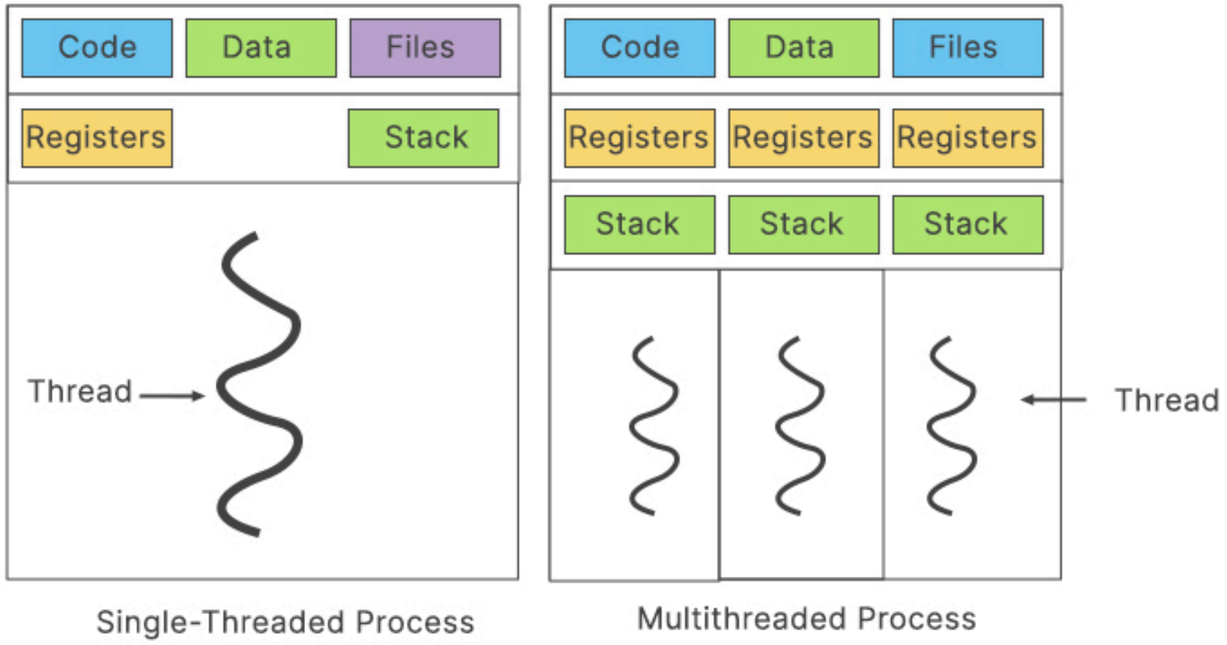
To enhance computing speed and reduce memory usage

- Loop optimization: Optimize locality and reduce branching overhead
  - Loop reordering: Optimizes locality by reordering the sequence of loops.
  - Loop tiling: Reduces memory access by partitioning a loop's iteration space.
  - Loop unrolling: reduces branching overhead at the expense of its binary size.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- **Multithreading:**
  - Concurrent execution of multiple threads within a single process.
- CUDA programming:
  - Use GPUs to accelerate computation.

Multithreading

Shared-Memory Programming

- Multithreading is the concurrent execution of multiple threads within a single process.
- A thread is the smallest unit of execution in a program.
- Threads share the same memory space and resources but have their own stack and program counter.
- Different threads can run on separate CPU cores, improving performance and allowing parallelism.



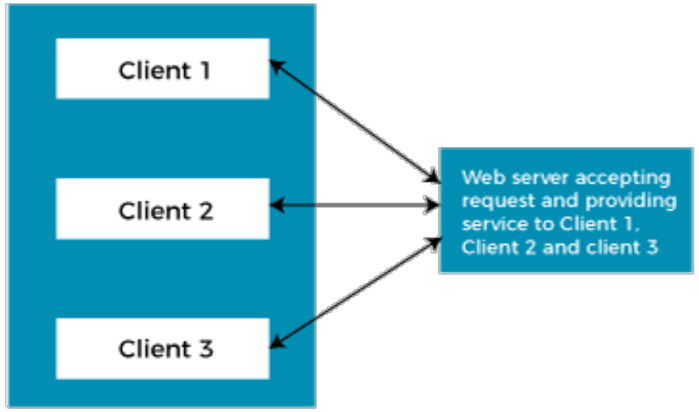
"What Is Multithreading In OS? Understanding The Details" [Link]

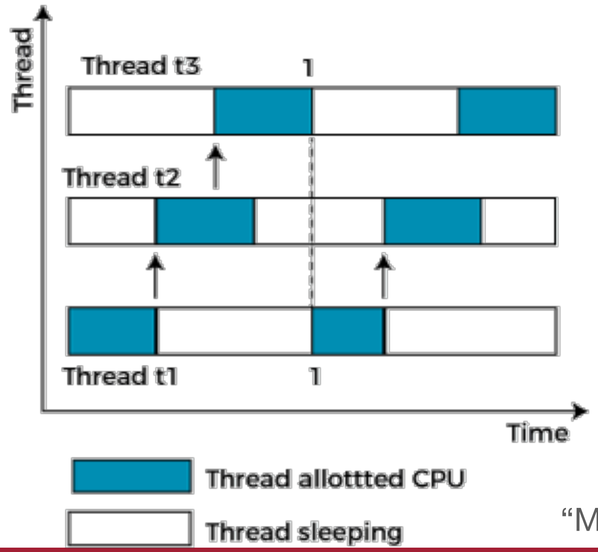
“Multithreading example code” [Link]

Multithreading

Shared-Memory Programming

- Improved Performance:
  - Multiple threads can execute tasks simultaneously, increasing overall program speed.
- Responsiveness:
  - A program can remain responsive to user input without blocking.
- Resource Utilization:
  - Threads can share resources, reducing the overhead of creating multiple processes.
- Simplified Program Structure:
  - Multithreading can help break down complex problems into simpler, smaller tasks.





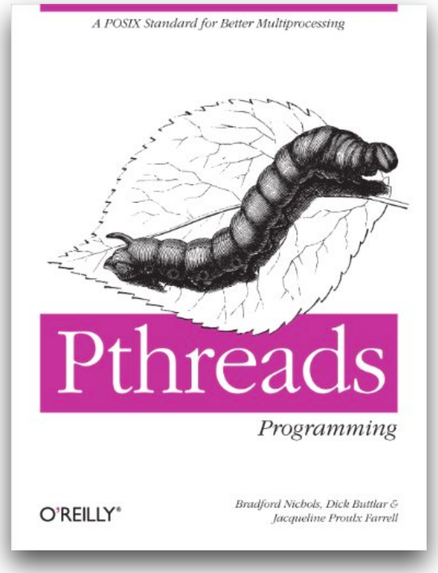
“Multithreading Models in Operating System” [Link]

“Multithreading example code” [Link]

Multithreading

Shared-Memory Programming

- Pthreads:
  - A C library for creating and managing POSIX threads.
- OpenMP:
  - An API for C, C++, and Fortran to support parallel programming using shared-memory model.



“PThreads Programming” [Link]; “OpenMP” [Link]

Multithreading

Naive implementation on CPU: `naive_mat_mul: 24296 ms`

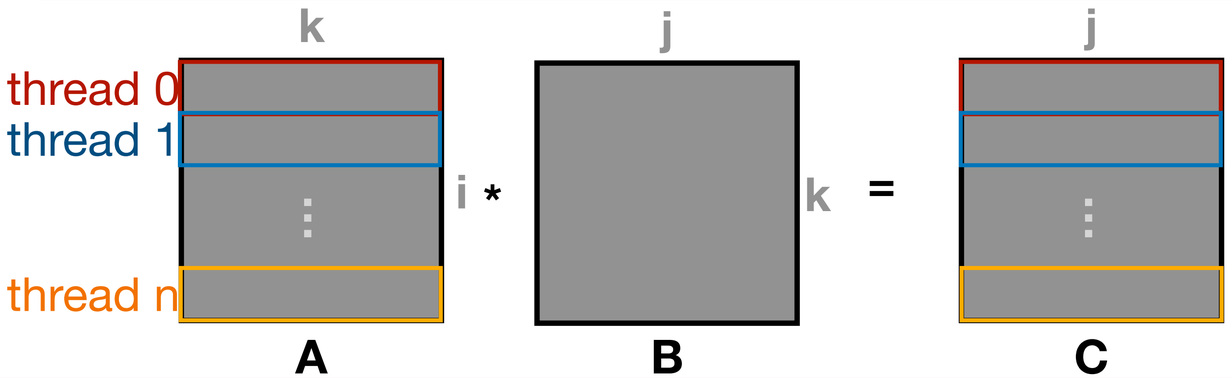
Naive multithreading (4 threads): `mat_mul_multithreading: 5864 ms`

4.1x speed up

*Results are measured on Intel Xeon 4114

Speed up Matrix Multiplication with Multithreading

```
int main() {
    // Initiate the threads
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];

    // Create threads and assign work
    for (int i = 0; i < NUM_THREADS; ++i) {
        thread_data[i].thread_id = i;
        pthread_create(&threads[i], nullptr, mat_mul_multithr
eading,&thread_data[i]);
    }

    // Join threads to wait for their completion
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], nullptr);
    }

    return 0;
}
```




```
struct ThreadData {
    int thread_id;
};

// Specify the function that each thread needs to do
void* mat_mul_multithreading(void* arg) {
    ThreadData* data = static_cast<ThreadData*>(arg);
    int thread_id = data->thread_id;
    int rows_per_thread = SIZE_MATRIX / NUM_THREADS;

    // Indicate the starting and ending rows of each
       threads
    int start_row = thread_id * rows_per_thread;
    int end_row = (thread_id + 1) * rows_per_thread;

    // Each thread only conducts a part of the mat_mul
    for (int i = start_row; i < end_row; ++i) {
        for (int j = 0; j < SIZE_MATRIX; ++j) {
            for (int k = 0; k < SIZE_MATRIX; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return nullptr;
}
```

"Multithreading example code" [Link]

Multithreading

Introduction of OpenMP

- Open Multi-Processing (OpenMP):
  - A shared-memory parallel programming model
  - API for C, C++, and Fortran
  - Portable across different platforms and operating systems
  - Easy integration with existing code
  - Scalability for varying numbers of processors
- OpenMP Compiler Directives:
  - #pragma omp parallel: create a parallel region
  - #pragma omp for: parallelize a for loop
  - #pragma omp sections: define parallel sections
- Manage threads and synchronization:
  - #pragma omp single, critical, and barrier
  - omp_set_num_threads(), omp_get_num_threads(), etc.

Multithreading

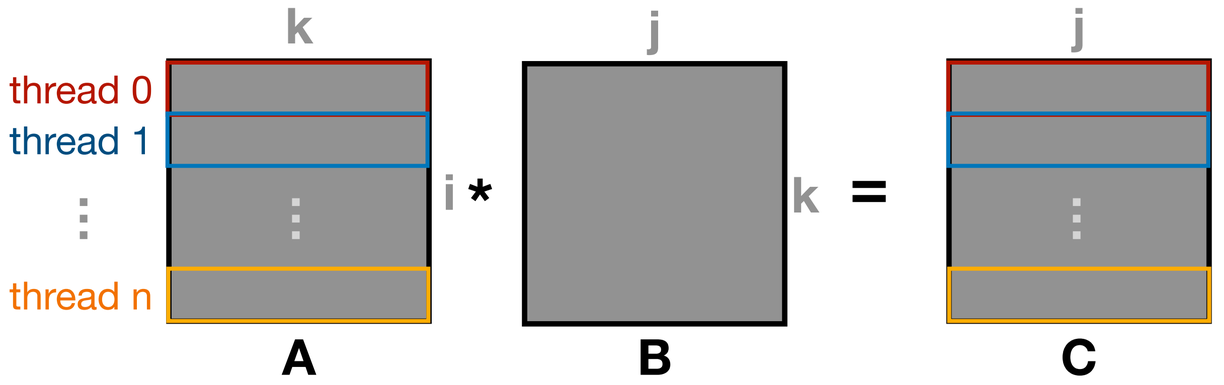
Speed up Matrix Multiplication with OpenMP

```
int main() {
    const int N = 100;  // Size of matrix

    // Initialize two matrices with random integers
    std::vector<std::vector<int>> A(N, std::vector<int>(N));
    std::vector<std::vector<int>> B(N, std::vector<int>(N));
    std::vector<std::vector<int>> C(N, std::vector<int>(N, 0));

    // Set the number of threads to use in the parallel region
    omp_set_num_threads(4);

    // Parallelize the loop with OpenMP
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return 0;
}
```

The code of using OpenMP is cleaner than that of using Pthreads
(Easy integration with existing code)

Indicate the number of threads

In OpenMP, we use #pragma to indicate where to parallelize



Parallel Computing Techniques

"Parallel computing tutorial [Link]

To enhance computing speed and reduce memory usage

- Loop optimization: Optimize locality and reduce branching overhead
  - Loop reordering: Optimizes locality by reordering the sequence of loops.
  - Loop tiling: Reduces memory access by partitioning a loop's iteration space.
  - Loop unrolling: reduces branching overhead at the expense of its binary size.
- SIMD (single instruction, multiple data) programming:
  - Performs the same operation on multiple data points simultaneously.
- Multithreading:
  - Concurrent execution of multiple threads within a single process.
- **CUDA programming:**
  - Use GPUs to accelerate computation.

CUDA Programming

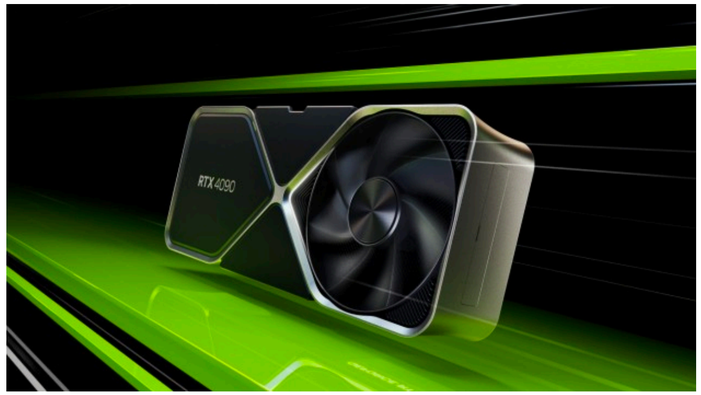
"CUDA programming example code" [Link]

Introduction

- The Graphics Processing Unit (GPU) provides much higher instruction throughput and memory bandwidth.

- CUDA is introduced by Nvidia in 2006 as a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs.

- CUDA is a C-like language to express programs that run on GPUs using the compute-mode hardware interface



NVIDIA CUDA

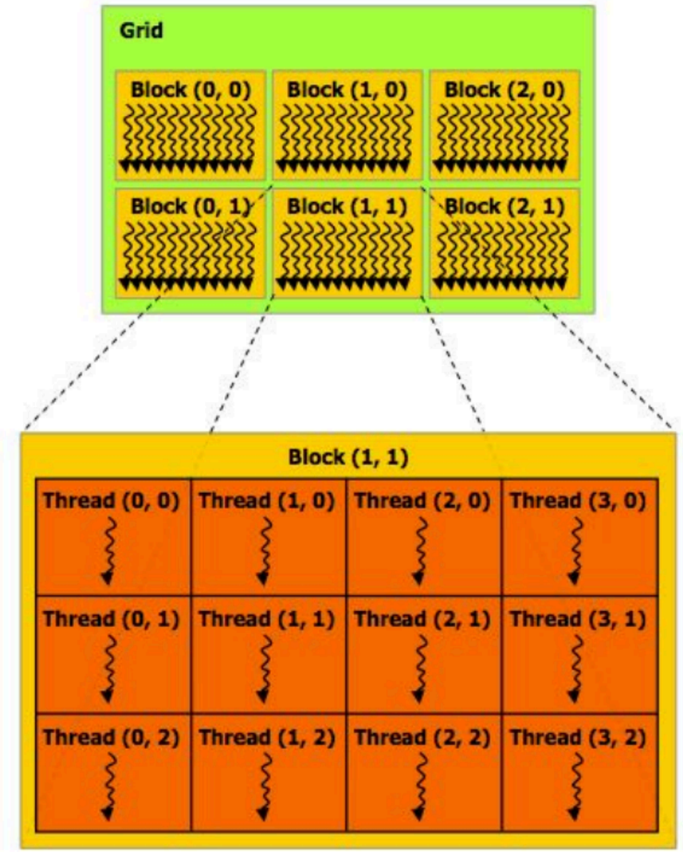
Graphics cards: https://www.nvidia.com/en-us/geforce/graphics-cards/

CUDA Programming

"CUDA programming example code" [Link]

Hierarchy of CUDA Threads

- Thread IDs can be up to 3-dimensional (2D example below)
- Multi-dimensional thread ids are convenient for problems that are naturally N-D



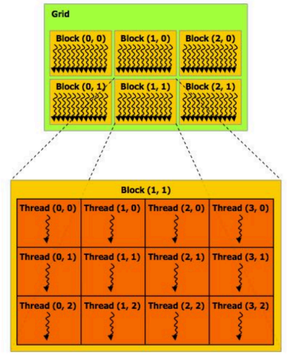
Regular application thread running on CPU (the "host")

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will launch 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Graphics cards: https://www.nvidia.com/en-us/geforce/graphics-cards/

CUDA Programming

"CUDA programming example code" [Link]



Programming Model

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will launch 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Host code runs on CPU: Serial execution

Bulk launch of many CUDA threads "launch a grid of CUDA thread blocks"

CUDA kernel definition

```
// kernel definition (runs on GPU)
__global__ void matrixAdd(float A[Ny][Nx], float B[Ny][Nx],
float C[Ny][Nx])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  C[j][i] = A[j][i] + B[j][i];
}
```

Kernel code runs on CUDA device: Parallel execution

Each thread computes its overall grid thread id from its position in its block (threadIdx) and its block's position in the grid (blockIdx)

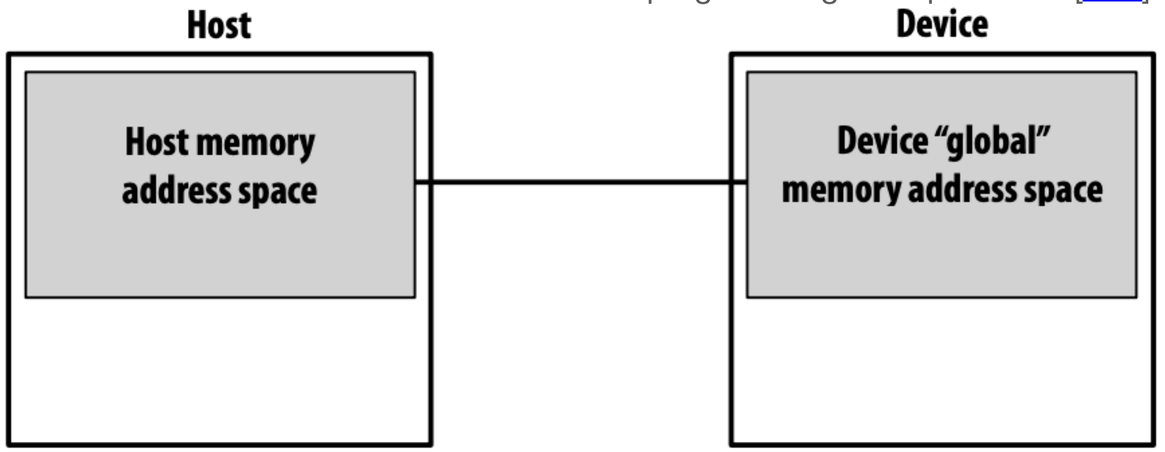
Stanford CS149, GPU architecture and CUDA Programming

CUDA Programming

Memory Model

- Distinct host and device address spaces
- Data can be moved between address spaces

"CUDA programming example code" [Link]




```
float* A = new float[N]; // allocate buffer in host memory

// populate host address space pointer A
for (int i=0;i<N;i++)
  A[i] = (float)I;

int bytes = sizeof(float) * N;

float* deviceA;
cudaMalloc(&deviceA, bytes)  // Allocate buffer in device address space

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);
```

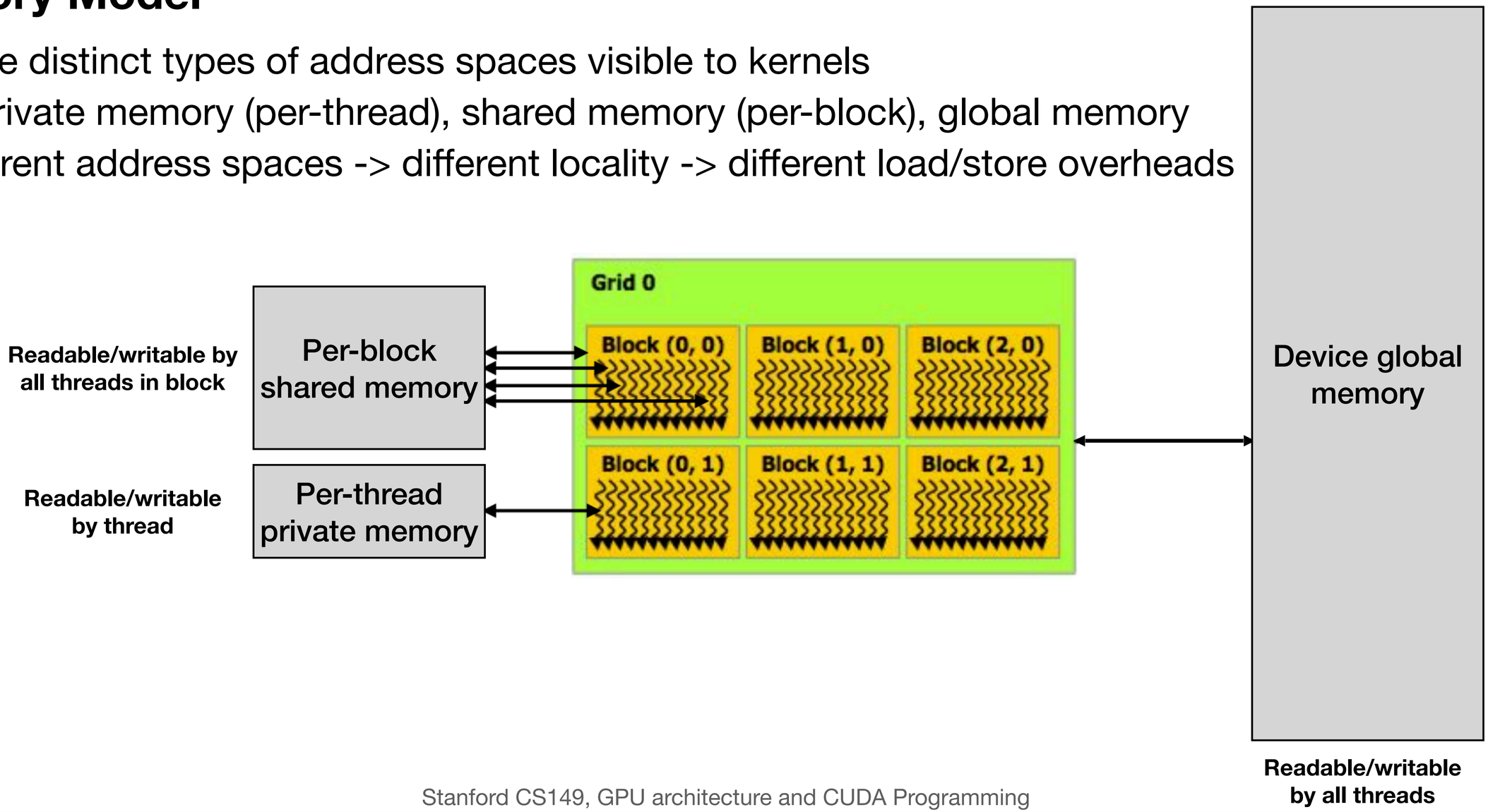
Access deviceA[i] from host will be invalid (not host address space)

CUDA Programming

"CUDA programming example code" [Link]

Memory Model

- Three distinct types of address spaces visible to kernels
  - Private memory (per-thread), shared memory (per-block), global memory
- Different address spaces -> different locality -> different load/store overheads



CUDA Programming

Speed up Matrix Multiplication

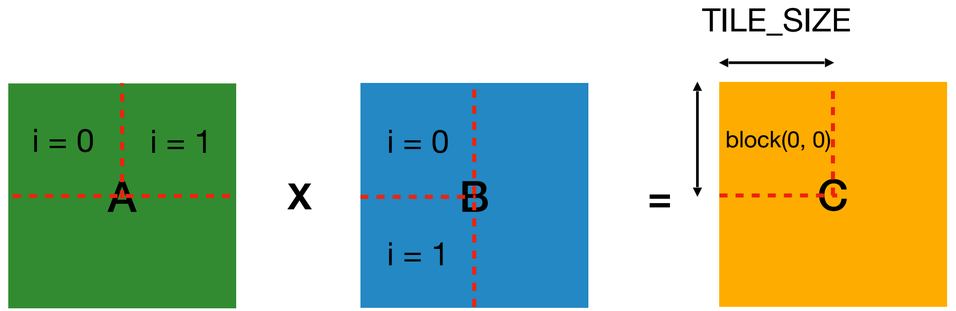
"CUDA programming example code" [Link]

Naive implementation on CPU:
```
naive_mat_mul: 24296 ms
```

CUDA implementation on 2080Ti:
```
cuda kernel: 6.796 ms
mat_mul_cuda: 258 ms
```

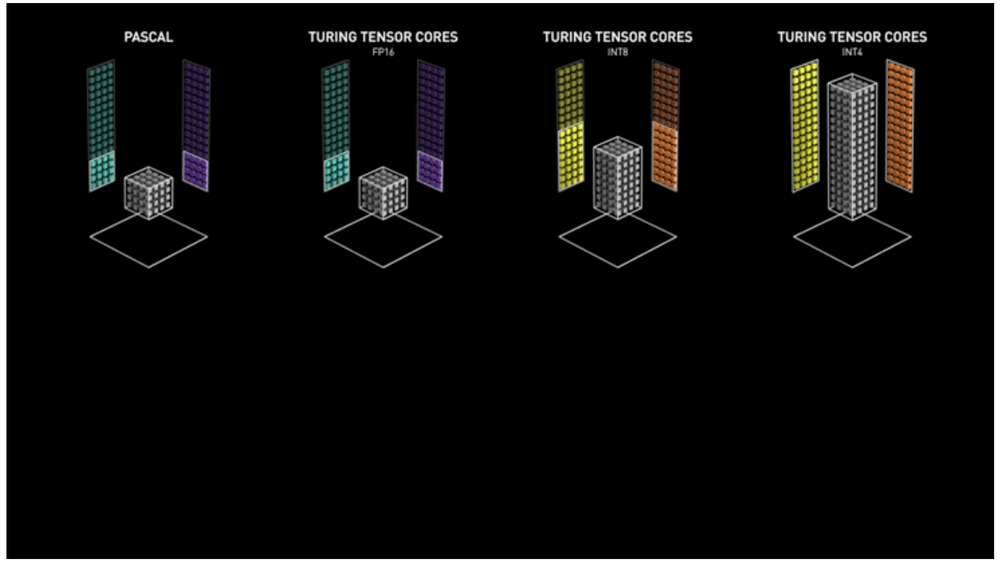
94x end-to-end speed up

```
__global__ void matrixMultiplyShared(const float *A, const float *B, float *C, int A_row, int A_column, int B_column) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  __shared__ float As[TILE_SIZE][TILE_SIZE]; // per-block allocation for tiling A
  __shared__ float Bs[TILE_SIZE][TILE_SIZE]; // per-block allocation for tiling B

  float value = 0;

  for (int i = 0; i < A_column / TILE_SIZE; i++){ // per-block allocation for tiling A
    As[threadIdx.y][threadIdx.x] = A[(blockIdx.y * TILE_SIZE + threadIdx.y) * A_column + TILE_SIZE * i + threadIdx.x];
    Bs[threadIdx.y][threadIdx.x] = B[(i * TILE_SIZE + threadIdx.y) * B_column + blockIdx.x * TILE_SIZE + threadIdx.x];

    __syncthreads(); // Wait for memory loading

    for (int k = 0; k < TILE_SIZE; k++)
      value += As[threadIdx.y][k] * Bs[k][threadIdx.x];

    __syncthreads(); // Wait for multiplication and accumulation
  }
  C[row * B_column + col] = value;

}
```

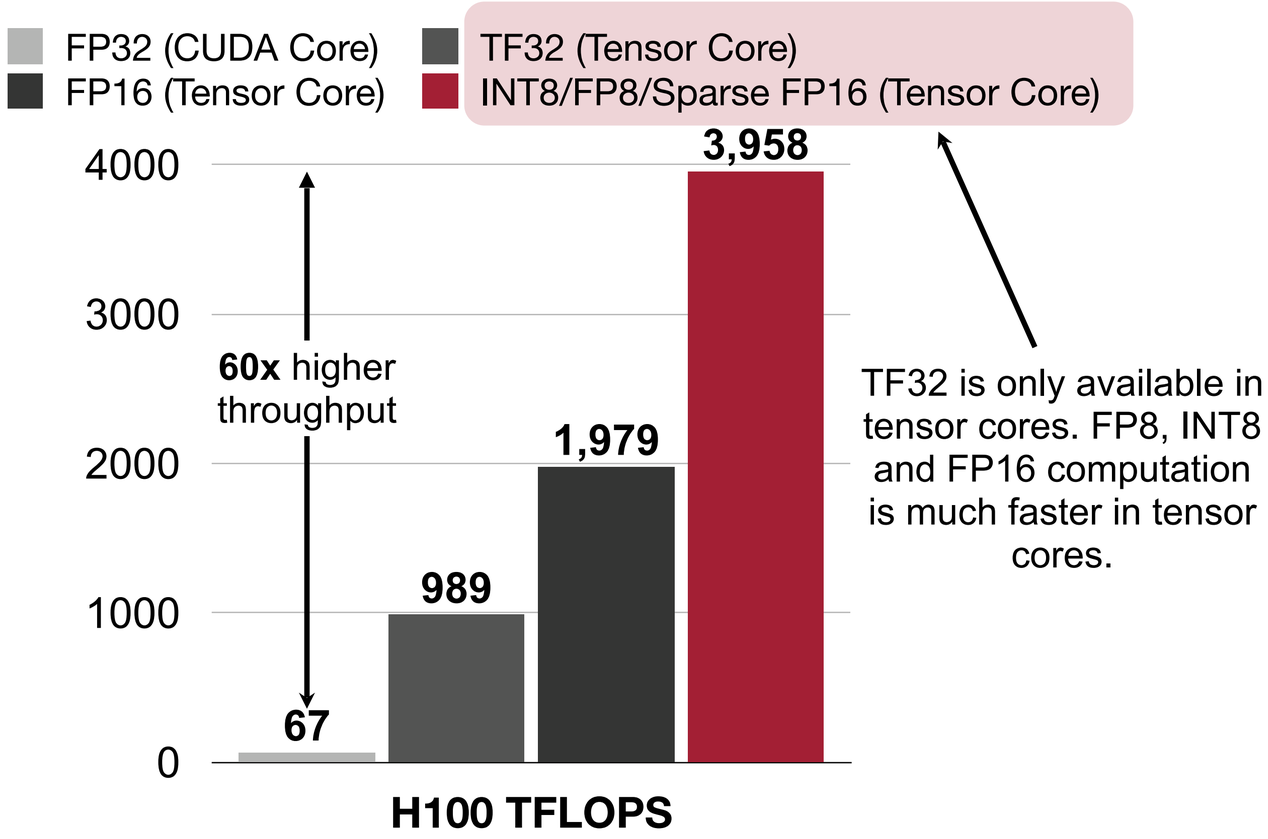


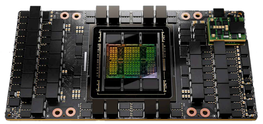
CUDA Programming on Tensor Cores

Motivation: Higher throughput and more data types



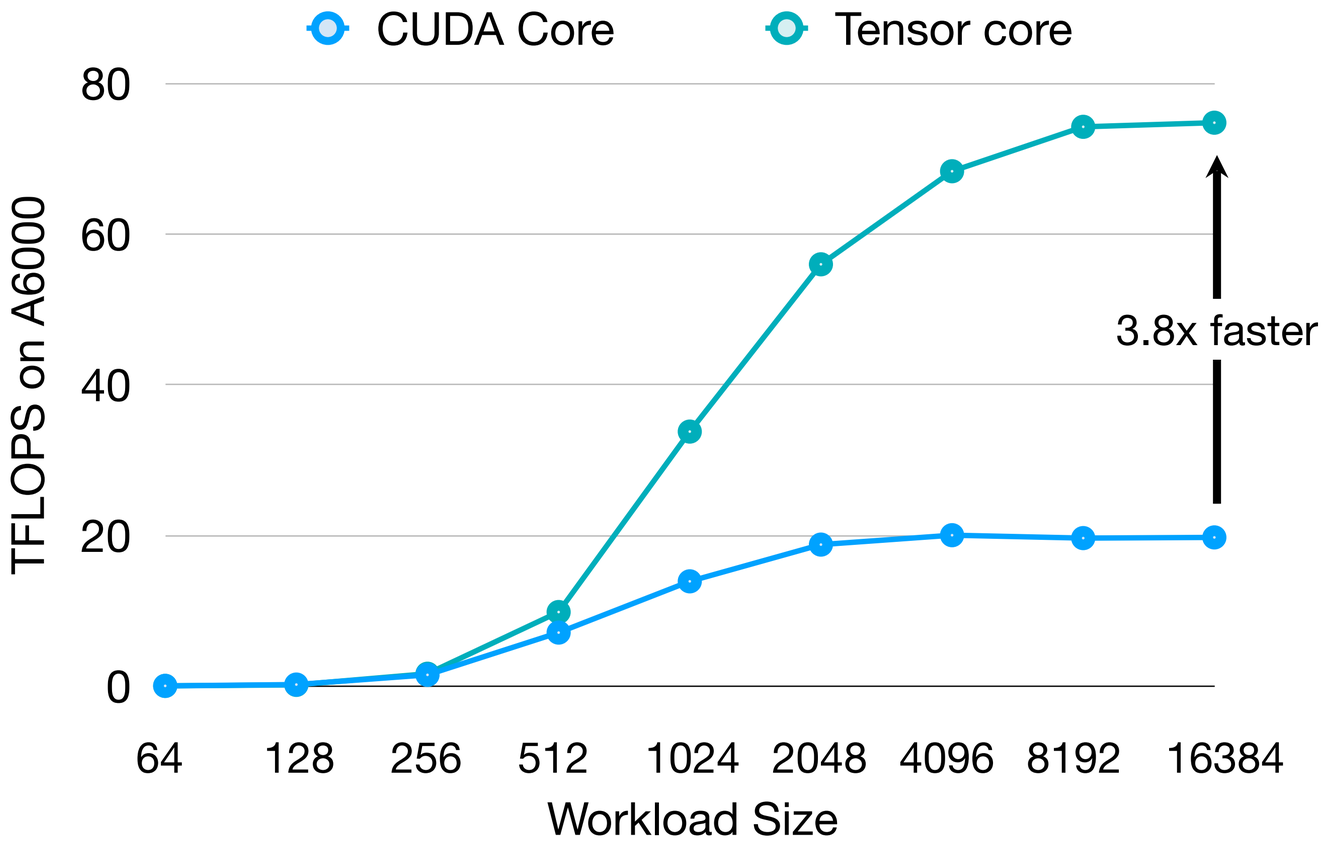
A CUDA core performs
1 FP32 / 2 FP16 multiply-accumulates (MACs) in each cycle;
In contrast, a tensor core can finish an entire matrix multiplication (**4x4x4 for Turing; 8x4x8 for Ampere**) in FP16 each cycle.





CUDA Programming on Tensor Cores

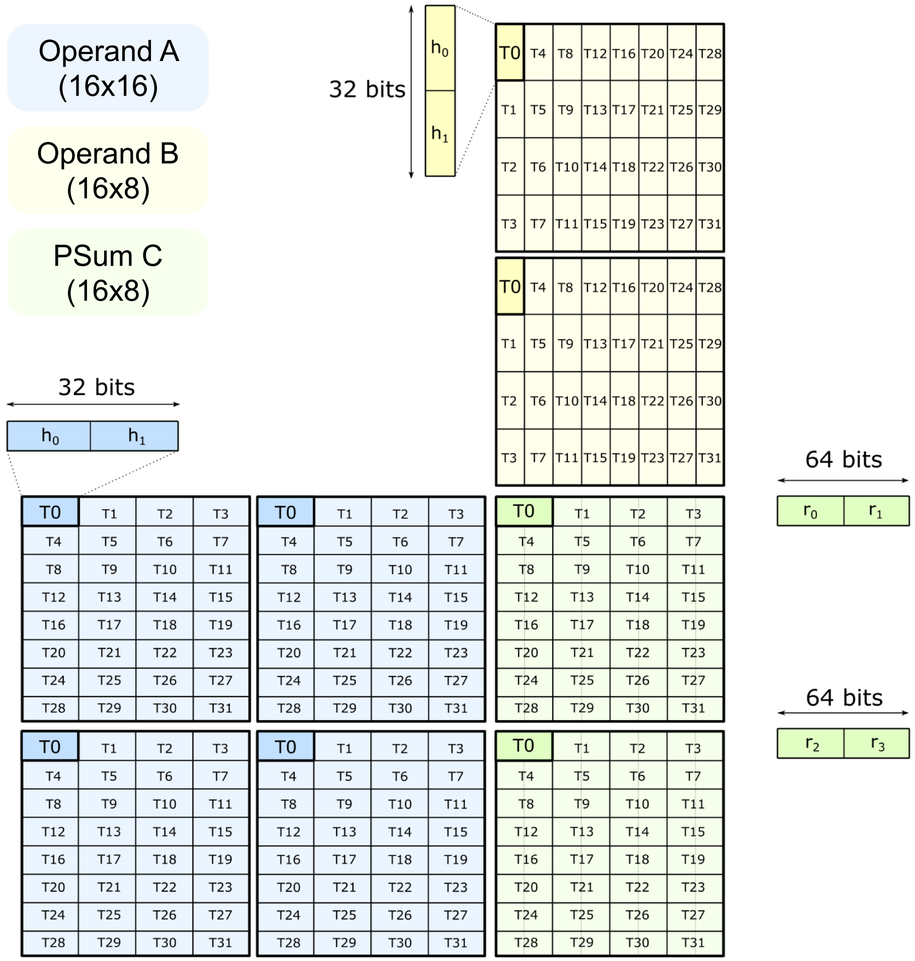
Motivation: Higher throughput and more data types



Tensor cores are much faster than CUDA cores for N x N x N matrix multiplication workloads when N is large.

Matrix multiplication intrinsics (MMA)

Motivation: Higher throughput and more data types

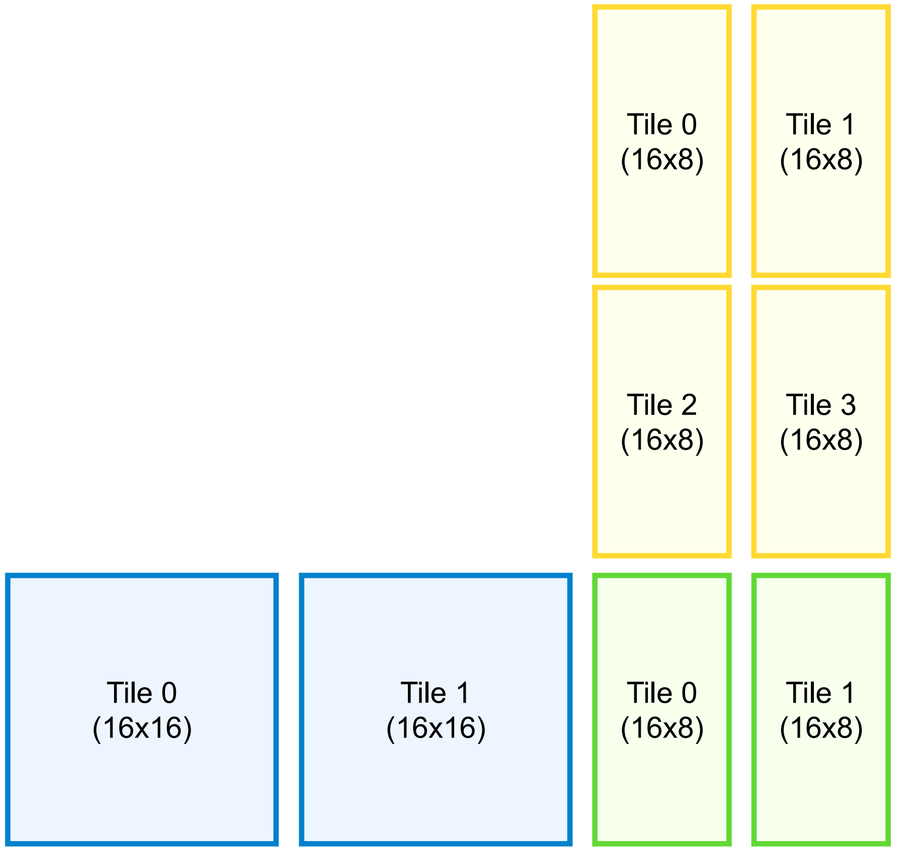



```
float           D[4];
uint32_t const A[4];
uint32_t const B[2];
float    const C[4];
// Example targets 16-by-8-by-16 Tensor Core operation
asm(
  "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 "
  "{ %0, %1, %2, %3 },"
  "{ %4, %5, %6, %7 },"
  "{ %8, %9 },"
"{ %10, %11, %12, %13 };"
:"=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])
:"r"(A[0]), "r"(A[1]), "r"(A[2]), "r"(A[3]),
  “r"(B[0]), "r"(B[1]),
"f"(C[0]), "f"(C[1]), "f"(C[2]), "f"(C[3])
);
```

Source: NVIDIA GTC 2021 slides

Matrix multiplication intrinsics (MMA)

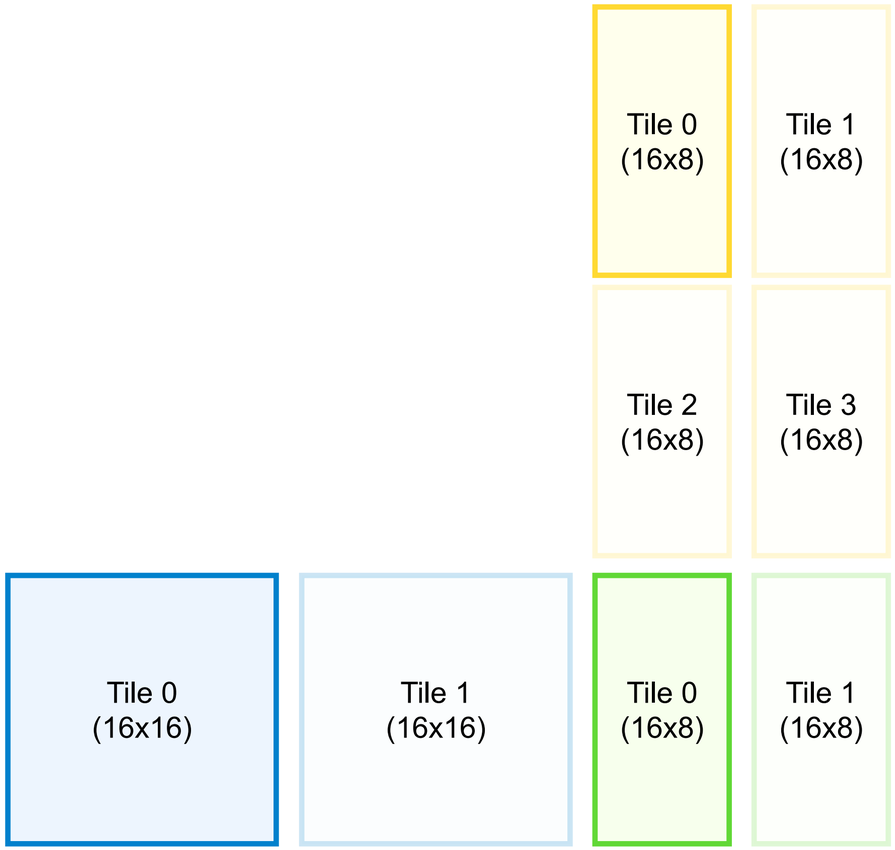
Compute 16x16x32 MMA using 16x8x16 intrinsics



Step 1. Break down input/output operands to 16x16 and 16x8 tiles. This results in two tiles for operand A, four tiles for operand B and two tiles for operand C.

CUDA Programming on Tensor Cores

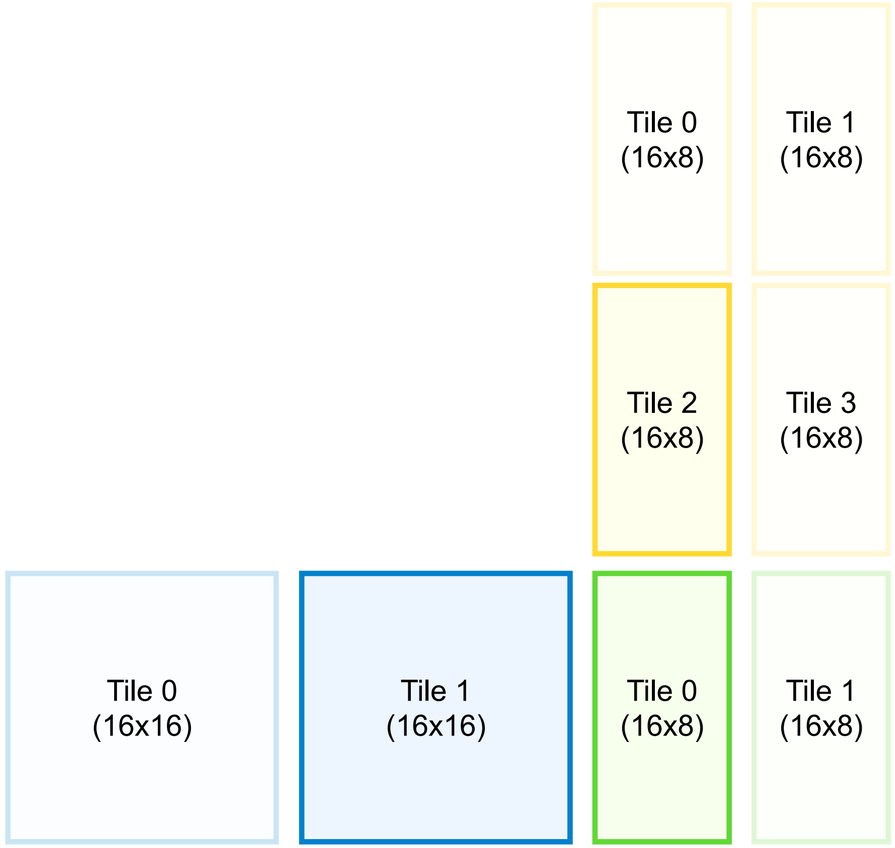
Motivation: Higher throughput and more data types



Step 2. Launch mma instruction between tile 0 of A and tile 0 of B; accumulate to output tile 0.

CUDA Programming on Tensor Cores

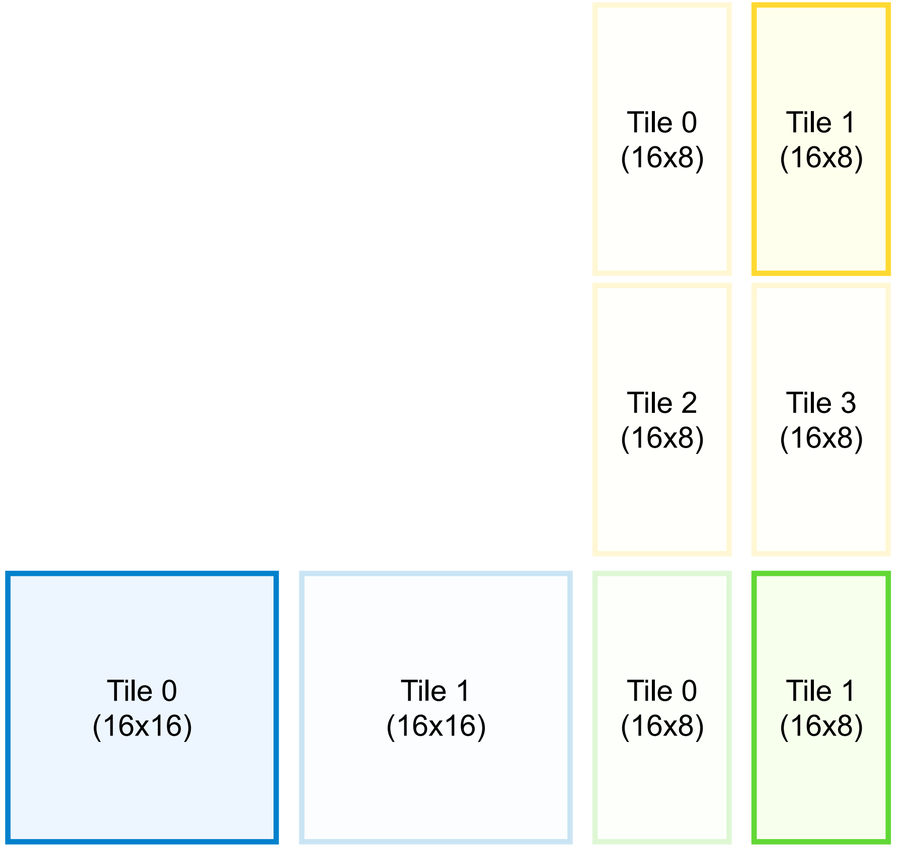
Motivation: Higher throughput and more data types



Step 3. Launch mma instruction between tile 1 of A and tile 2 of B; accumulate to output tile 0.

CUDA Programming on Tensor Cores

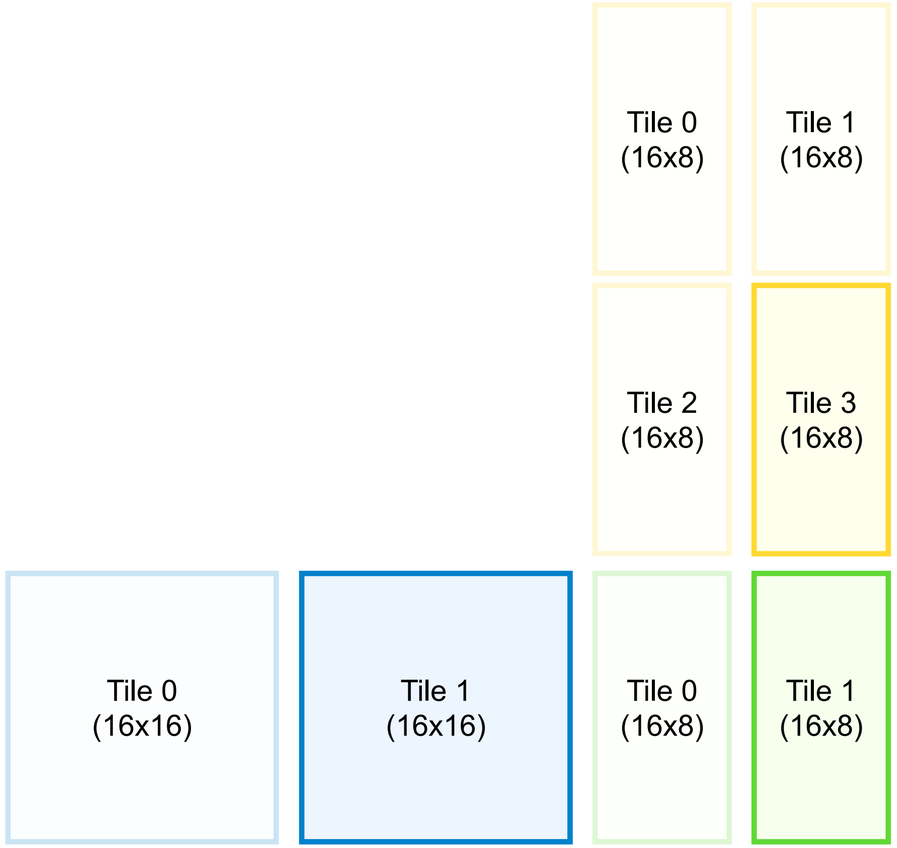
Motivation: Higher throughput and more data types



Step 4. mma instruction between tile 0 of A and tile 1 of B; accumulate to output tile 1.

CUDA Programming on Tensor Cores

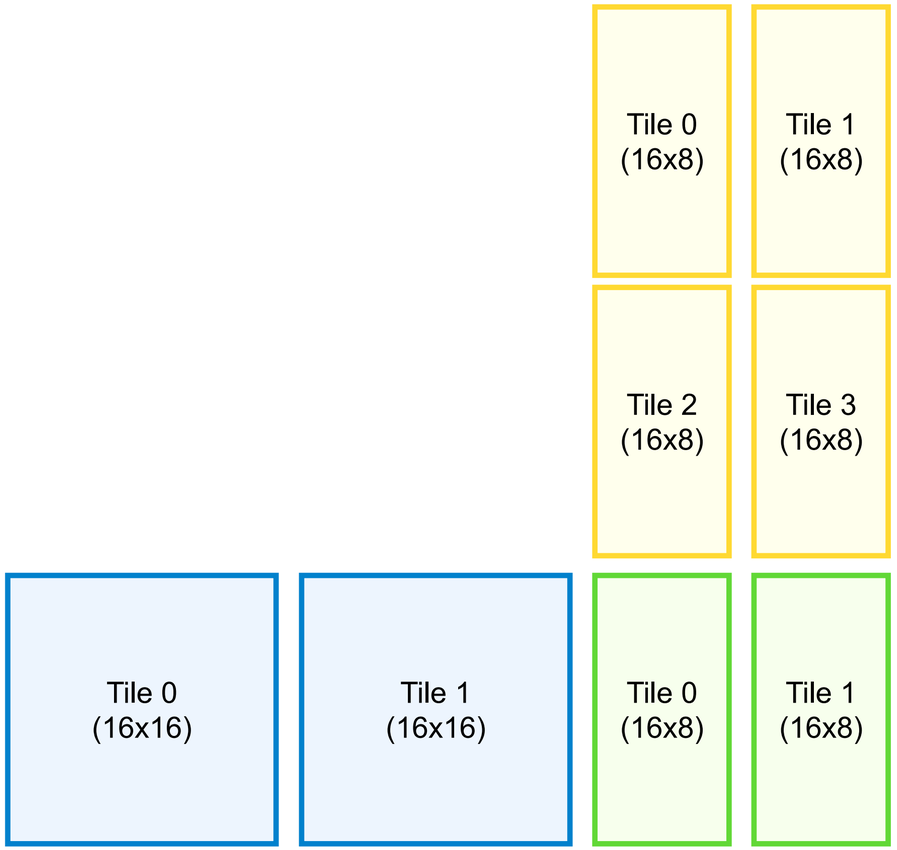
Motivation: Higher throughput and more data types



Step 5 mma instruction between tile 1 of A and tile 3 of B; accumulate to output tile 1.

CUDA Programming on Tensor Cores

Motivation: Higher throughput and more data types



Note: The order of applying these four MMA intrinsics does not matter.

Section 3: Inference Optimizations

Inference Optimizations

"Parallel computing tutorial [Link]

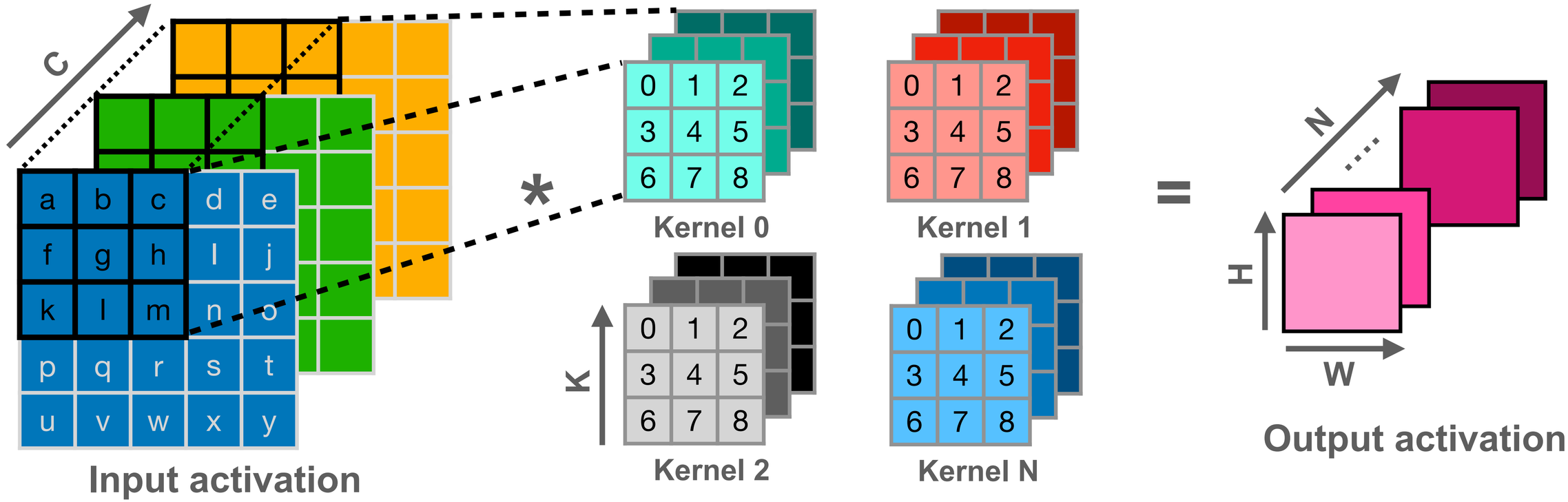
To enhance computing speed and reduce memory usage

- **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

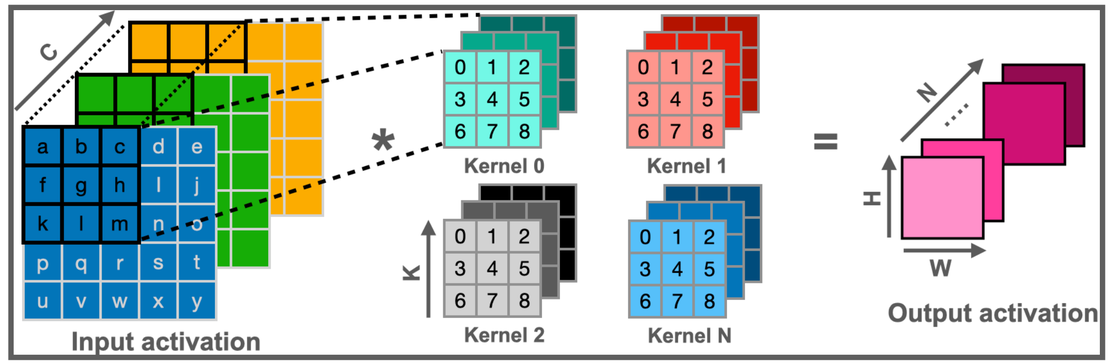
- Im2col is a technique to implement convolution with Generalized Matrix Multiplication (GEMM).

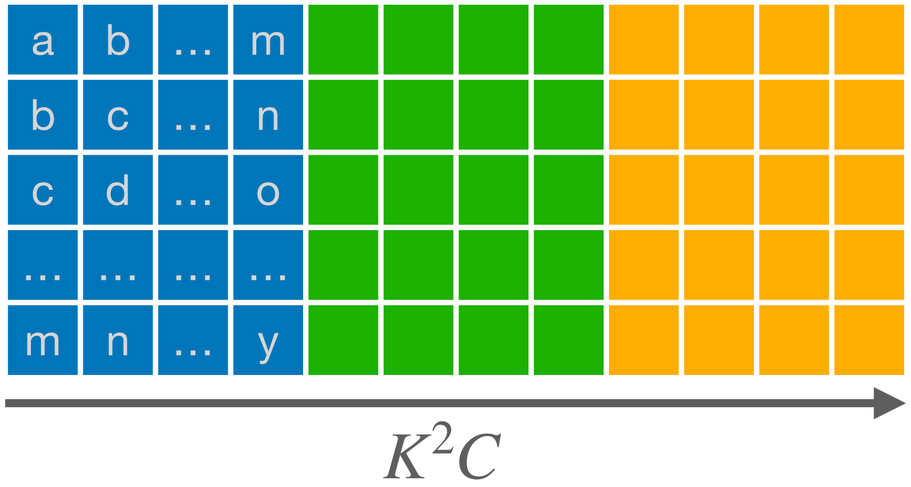


Anatomy of a High-Speed Convolution [Link]
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

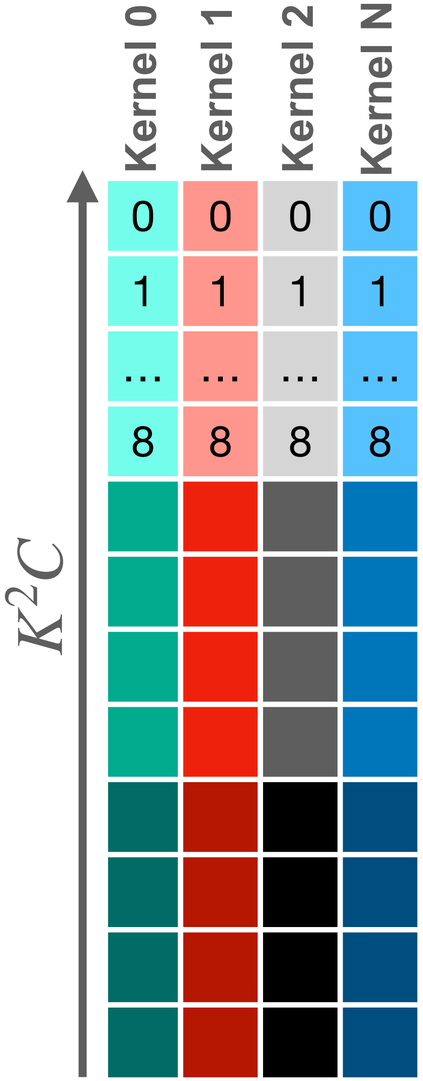
Im2col Convolution

Image to Column Convolution

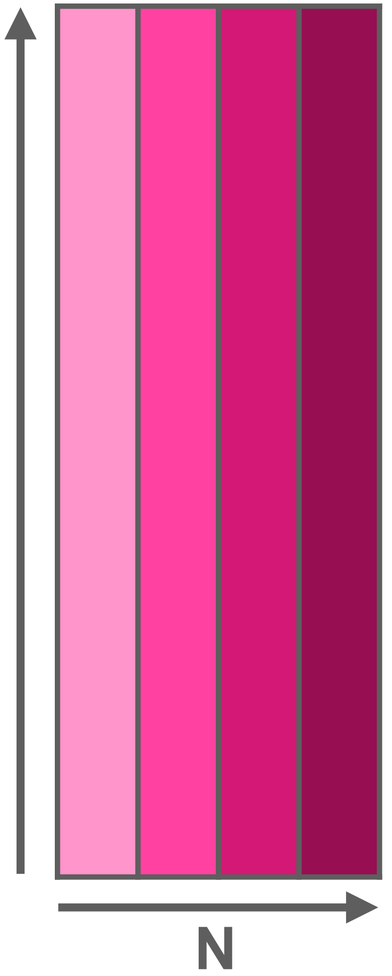




*



=



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]

Inference Optimizations

"Parallel computing tutorial [Link]

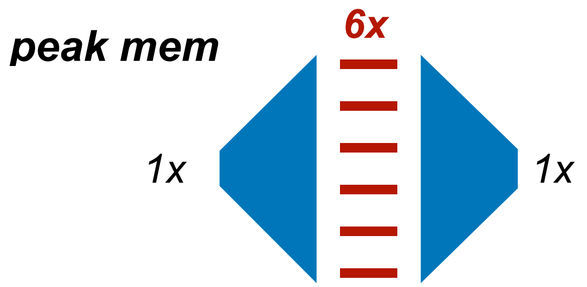
To enhance computing speed and reduce memory usage

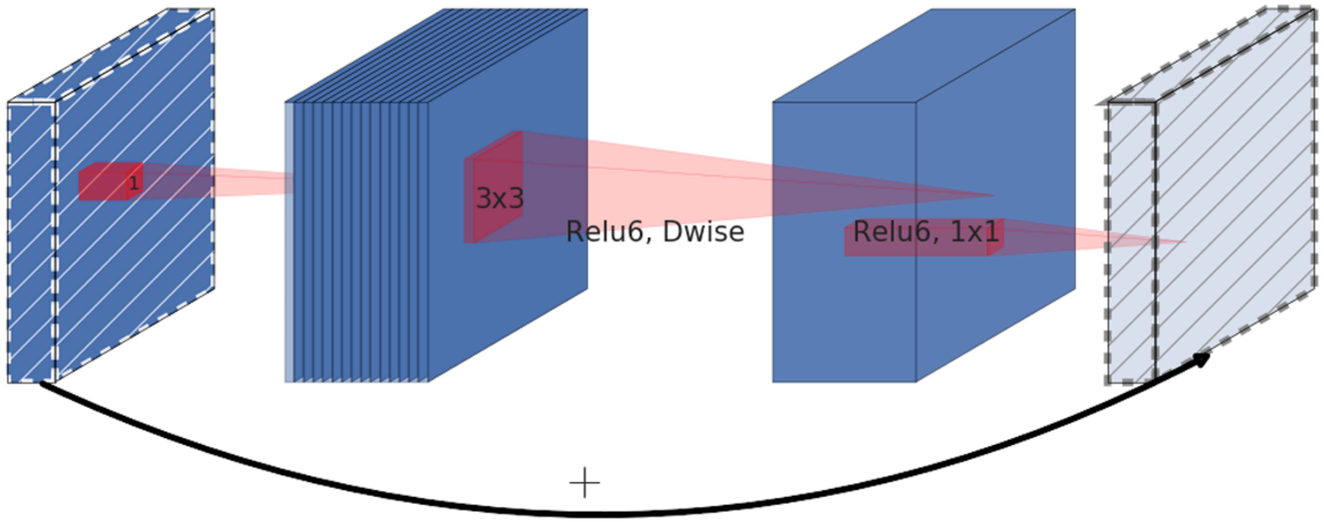
- Image to Column (Im2col) convolution:
  - Rearranges input data to directly utilize matrix multiplication kernels.
- **In-place depth-wise convolution:**
  - Reuse the input buffer to write the output data, so as to reduce peak SRAM memory.
- NHWC for point-wise convolution, and NCHW for depth-wise convolution:
  - Exploit the appropriate data layout for different types of convolution.
- Winograd convolution:
  - Reduce the number of multiplications to enhance computing speed for convolution.

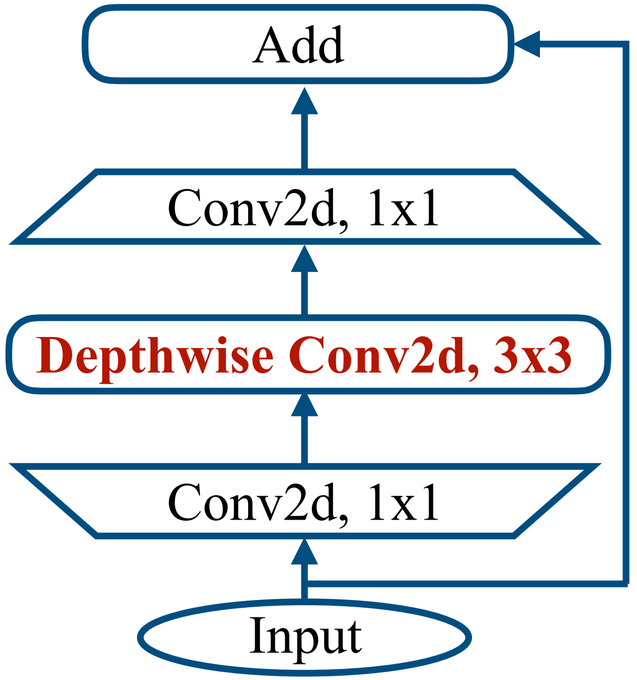
In-place Depth-wise Convolution

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).



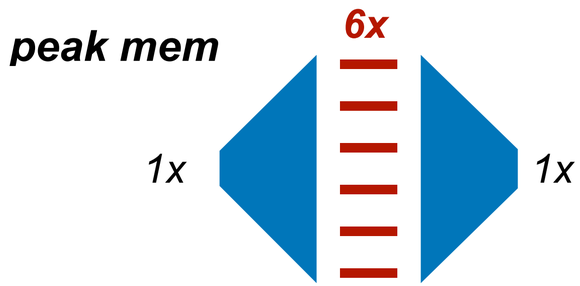




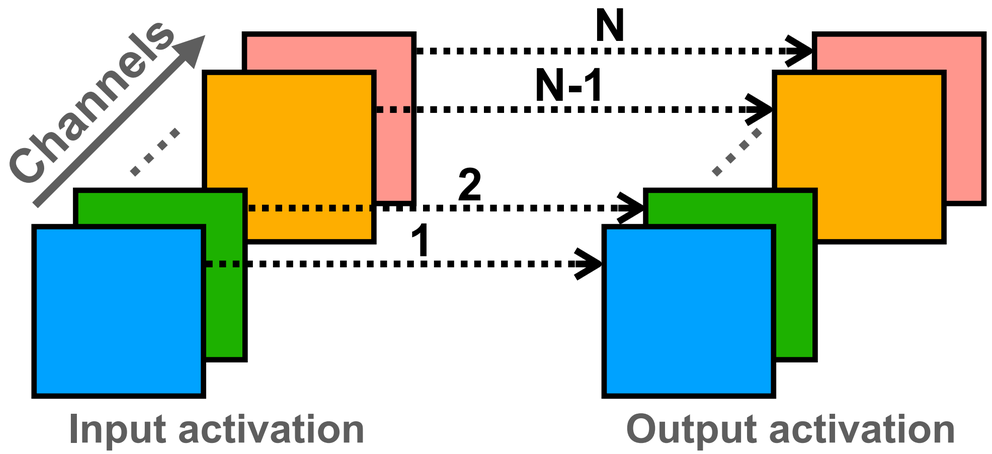
MobileNetV2: Inverted Residuals and Linear Bottlenecks [Sandler et al., CVPR 2018]
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

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

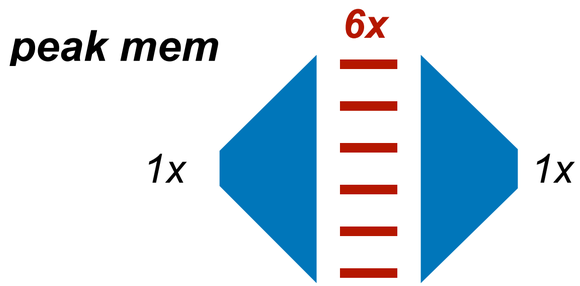
Peak Memory: 2xCxHxW

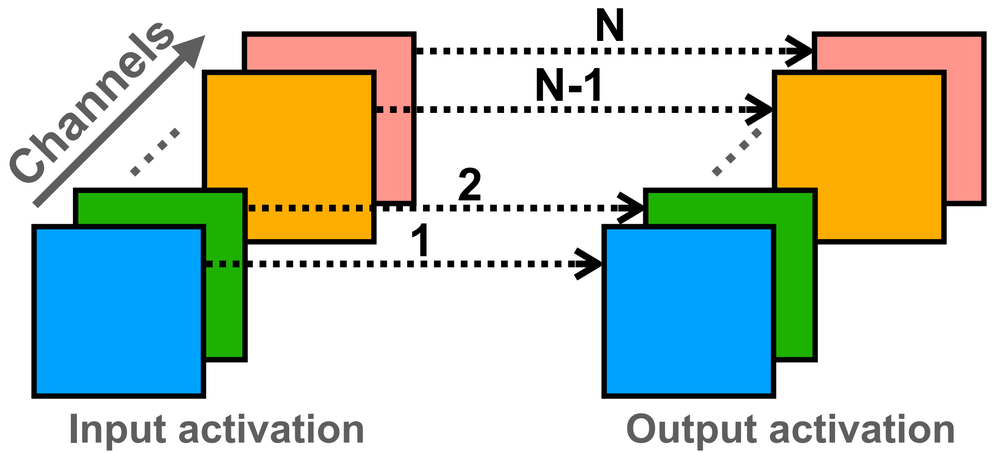
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

In-place Depth-wise Convolution

Reduce peak memory

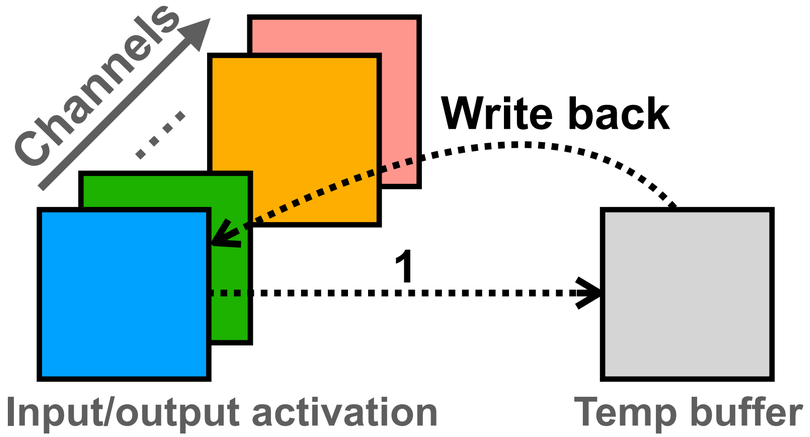
- To reduce the peak memory of depth-wise convolution, we utilize the “in-place” updating policy with a temporary buffer.





General depth-wise convolution

Peak Memory: 2xCxHxW



In-place depth-wise convolution

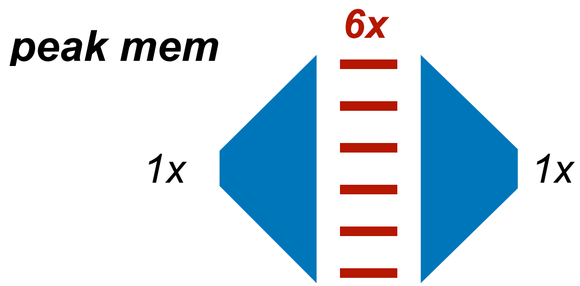
Peak Memory: (1+C)xHxW

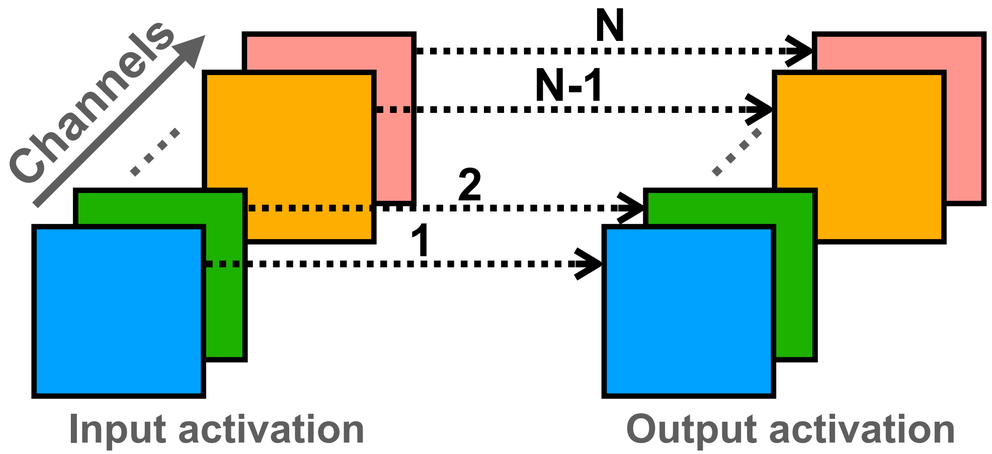
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

In-place Depth-wise Convolution

Reduce peak memory

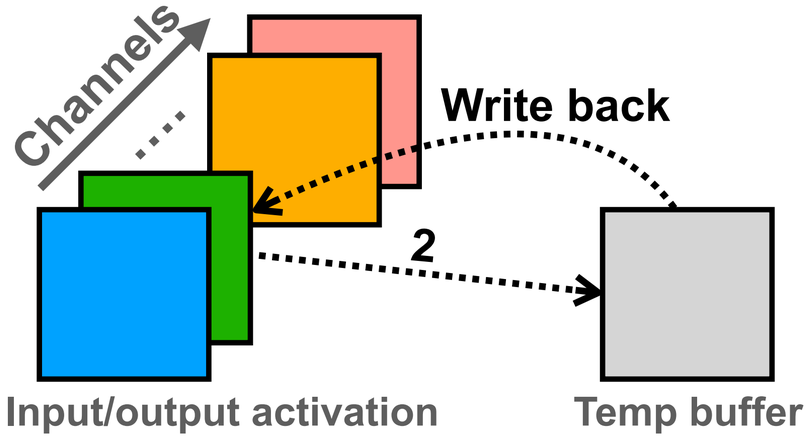
- To reduce the peak memory of depth-wise convolution, we utilize the “in-place” updating policy with a temporary buffer.





General depth-wise convolution

Peak Memory: 2xCxHxW



In-place depth-wise convolution

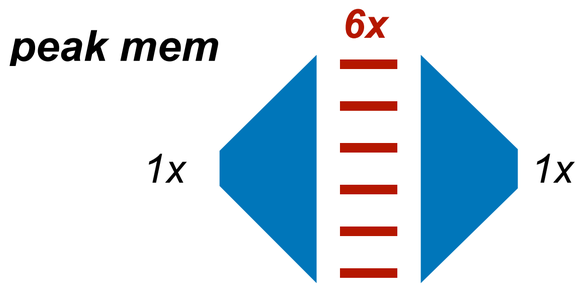
Peak Memory: (1+C)xHxW

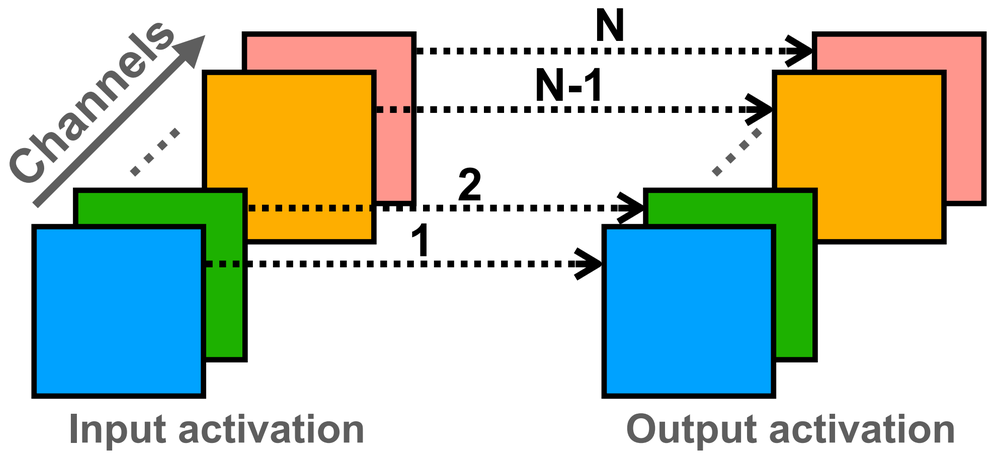
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

In-place Depth-wise Convolution

Reduce peak memory

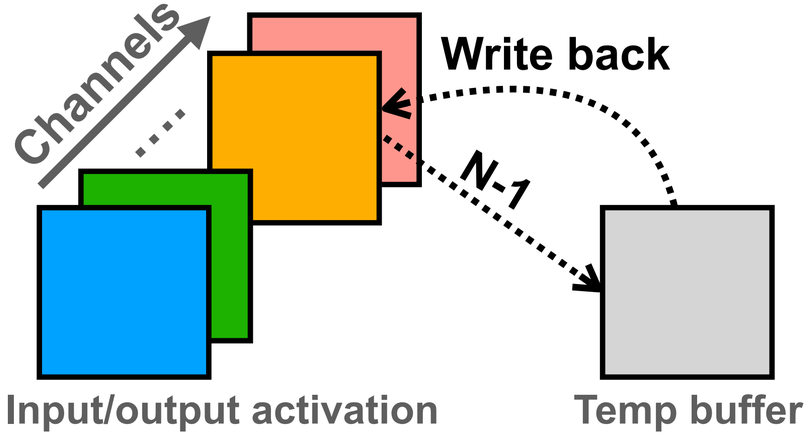
- To reduce the peak memory of depth-wise convolution, we utilize the “in-place” updating policy with a temporary buffer.





General depth-wise convolution

Peak Memory: 2xCxHxW



In-place depth-wise convolution

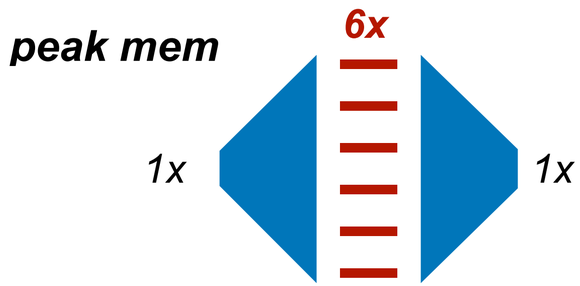
Peak Memory: (1+C)xHxW

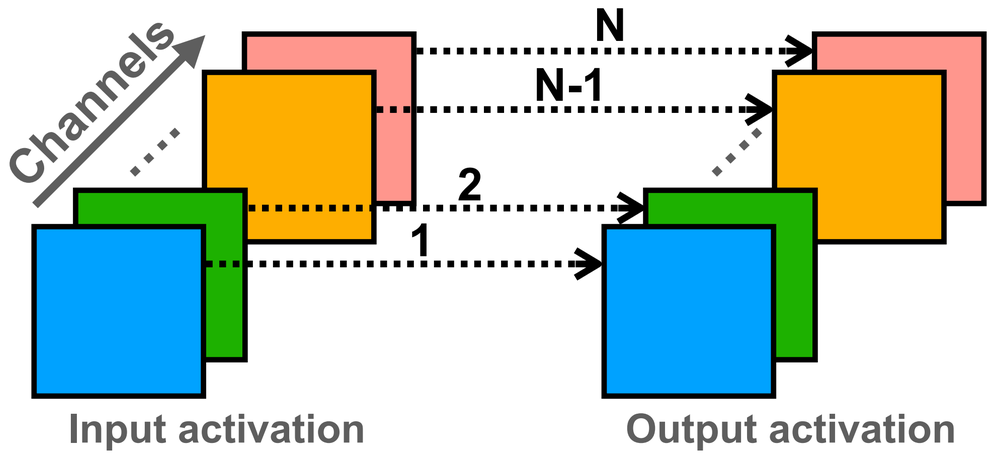
MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

In-place Depth-wise Convolution

Reduce peak memory

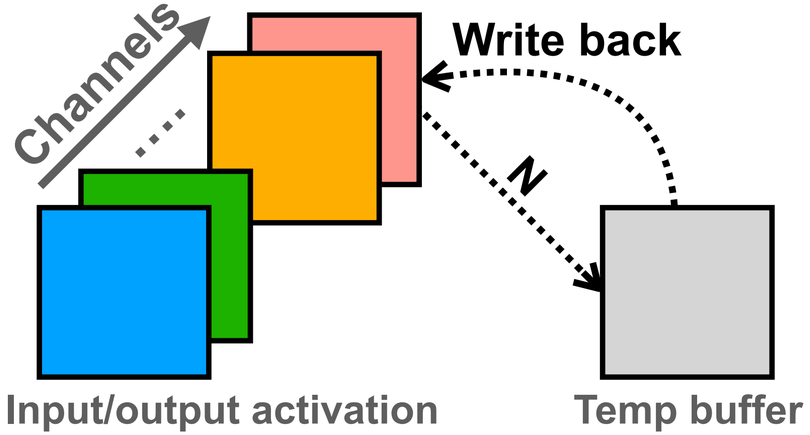
- To reduce the peak memory of depth-wise convolution, we utilize the “in-place” updating policy with a temporary buffer.





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]

Inference Optimizations

"Parallel computing tutorial [Link]

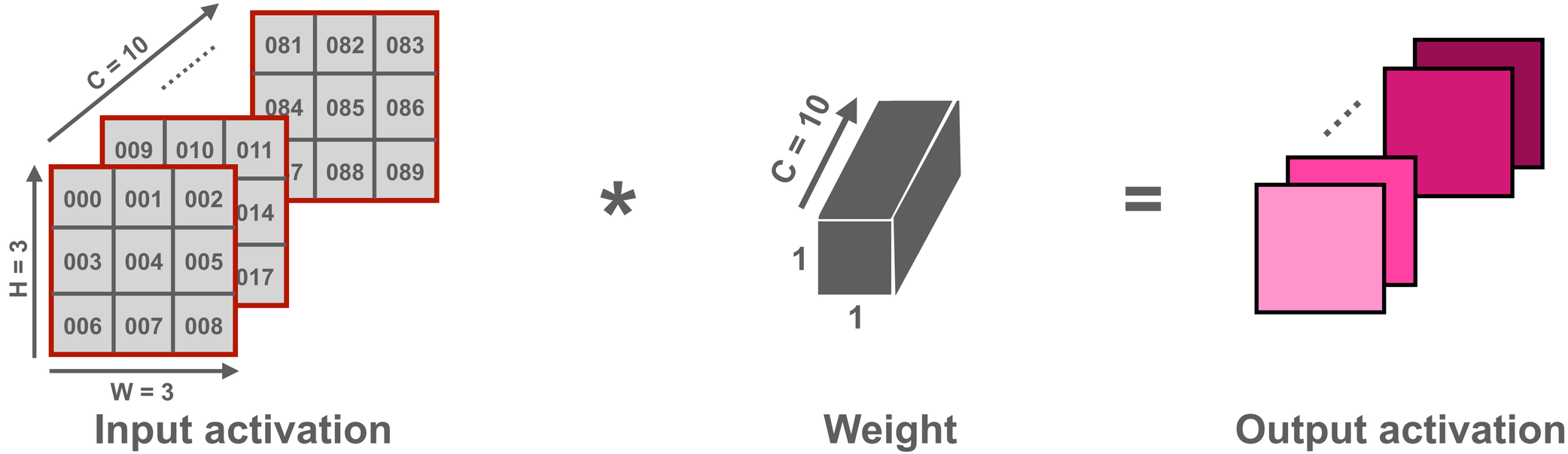
To enhance computing speed and reduce memory usage

- 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.



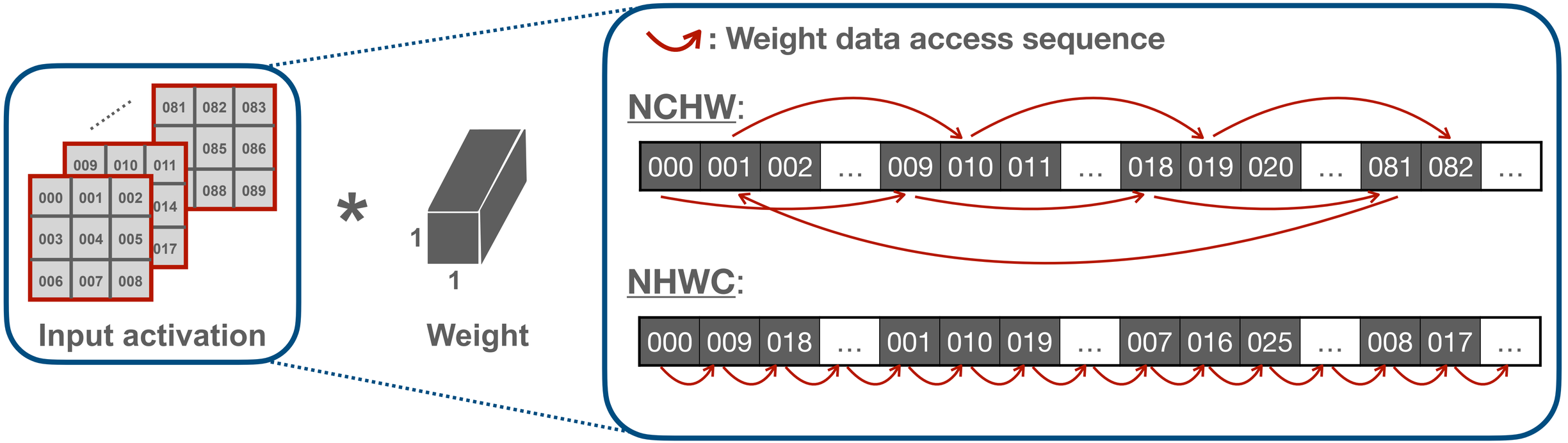
Input activation | Weight | Output activation

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.

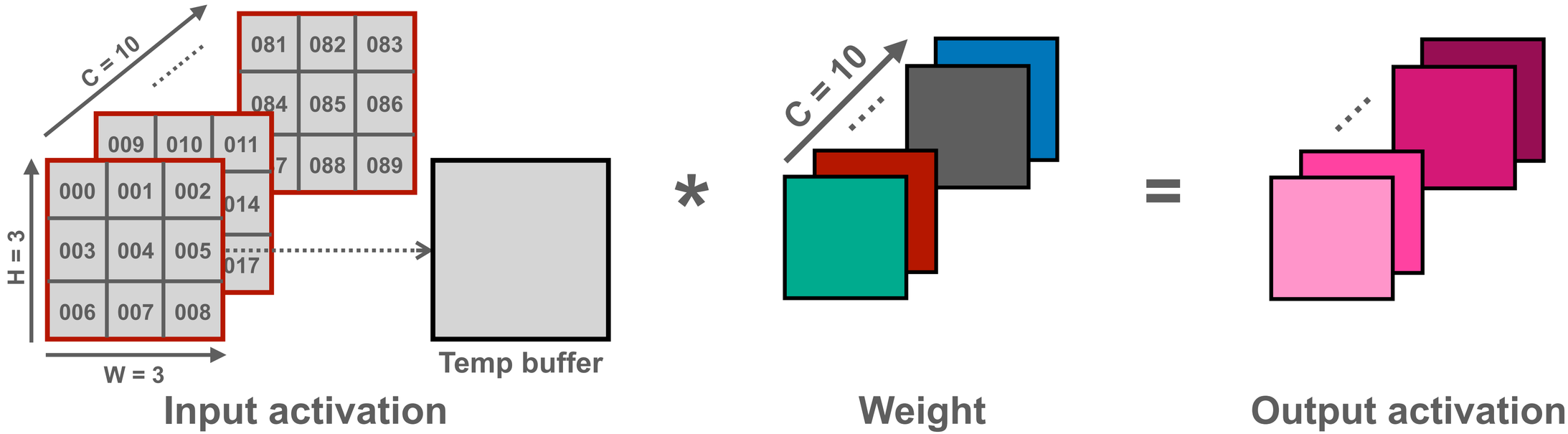


MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

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.

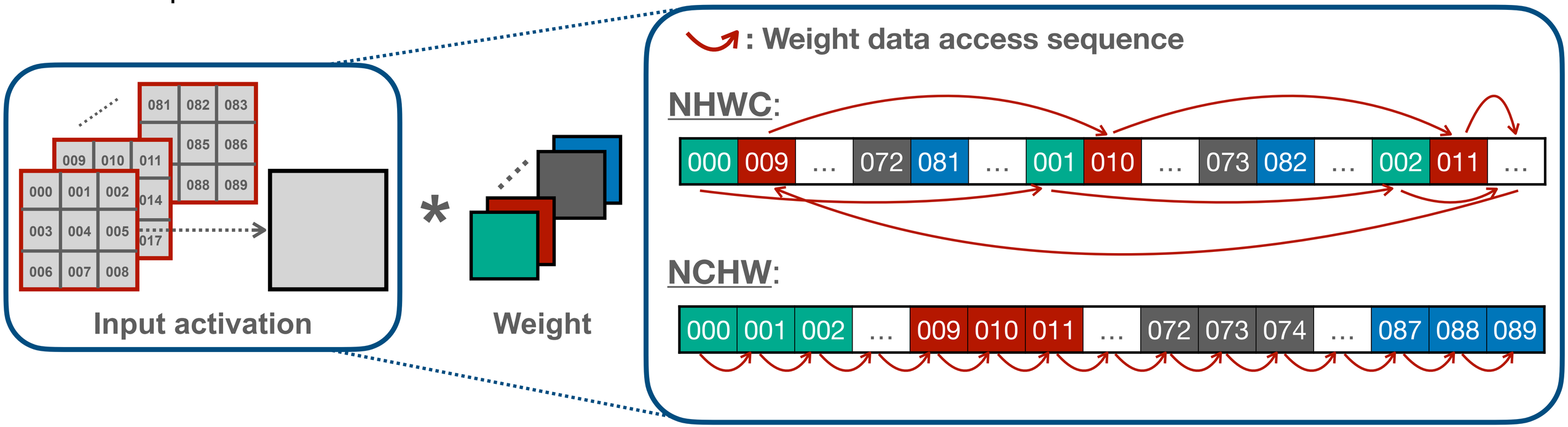


MCUNet: Tiny Deep Learning on IoT Devices [Lin et al., NeurIPS 2020]
On-Device Training Under 256KB Memory [Lin et al., NeurIPS 2022]

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]

Inference Optimizations

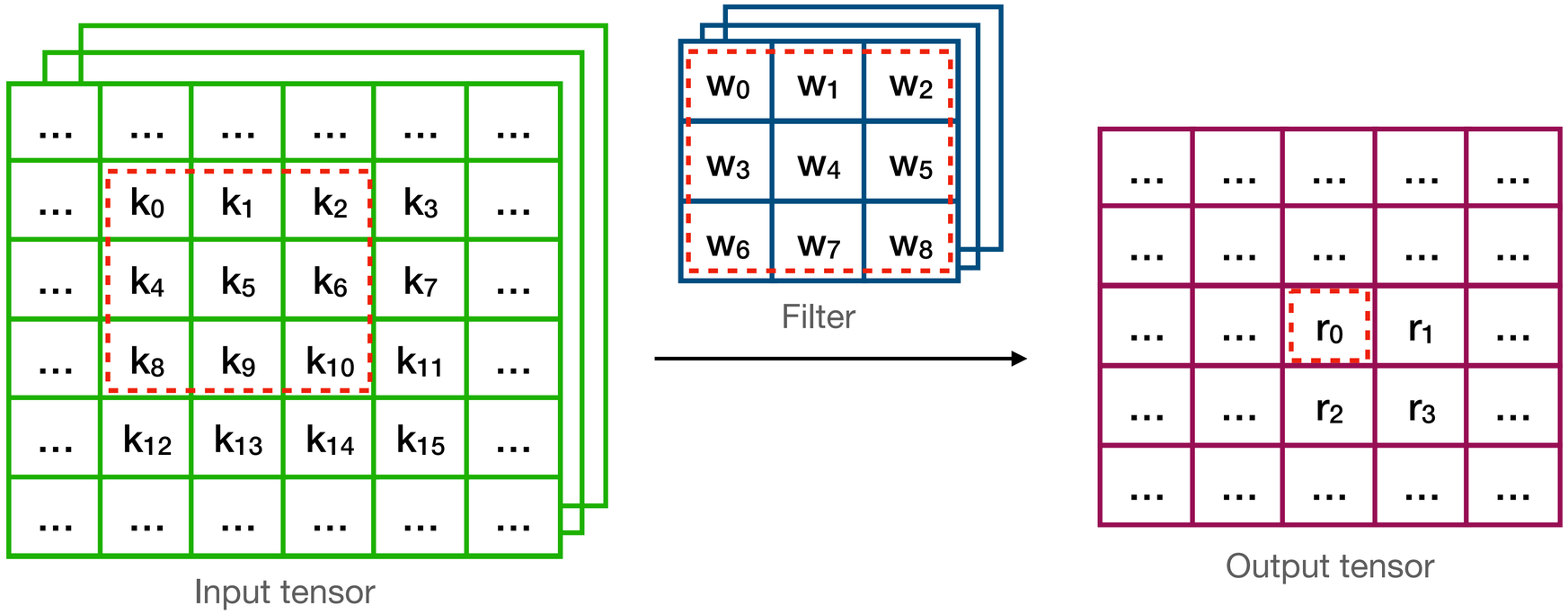
"Parallel computing tutorial [Link]

To enhance computing speed and reduce memory usage

- 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

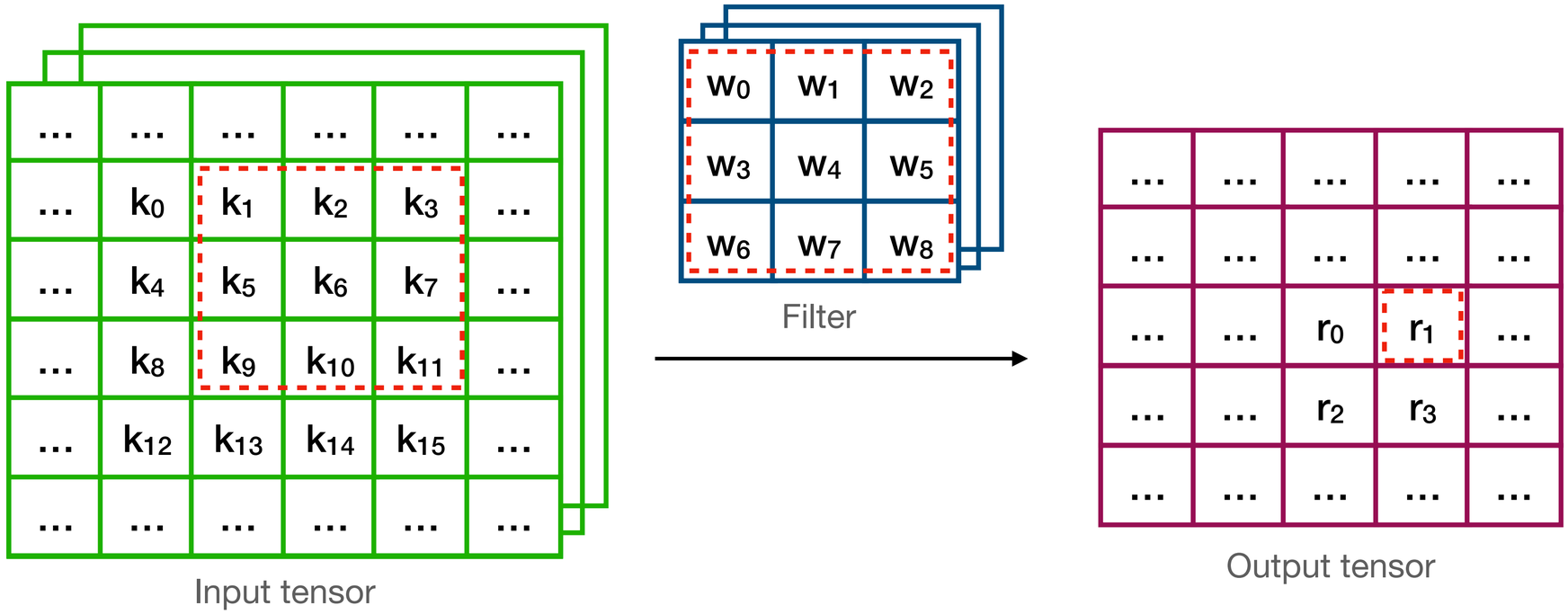
- Direct convolution need 9xCx4 MACs for 4 outputs



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

Winograd Convolution

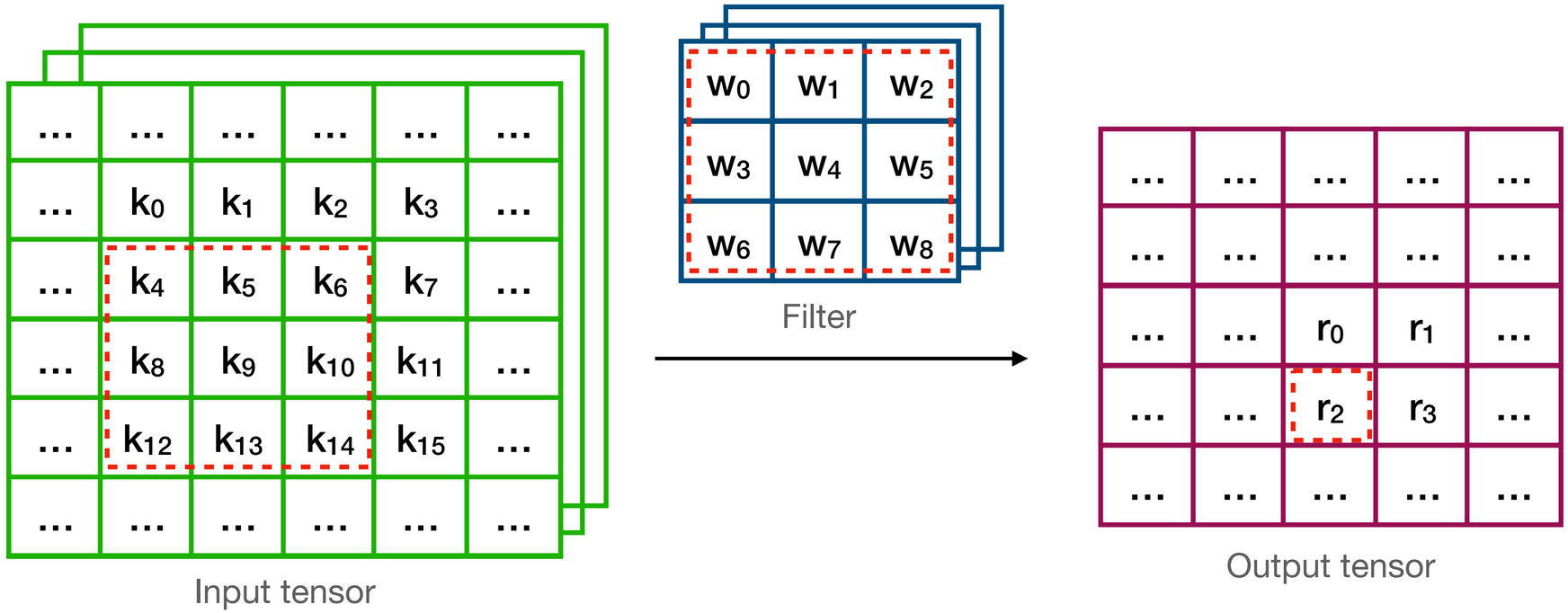
- Direct convolution need 9xCx4 MACs for 4 outputs



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

Winograd Convolution

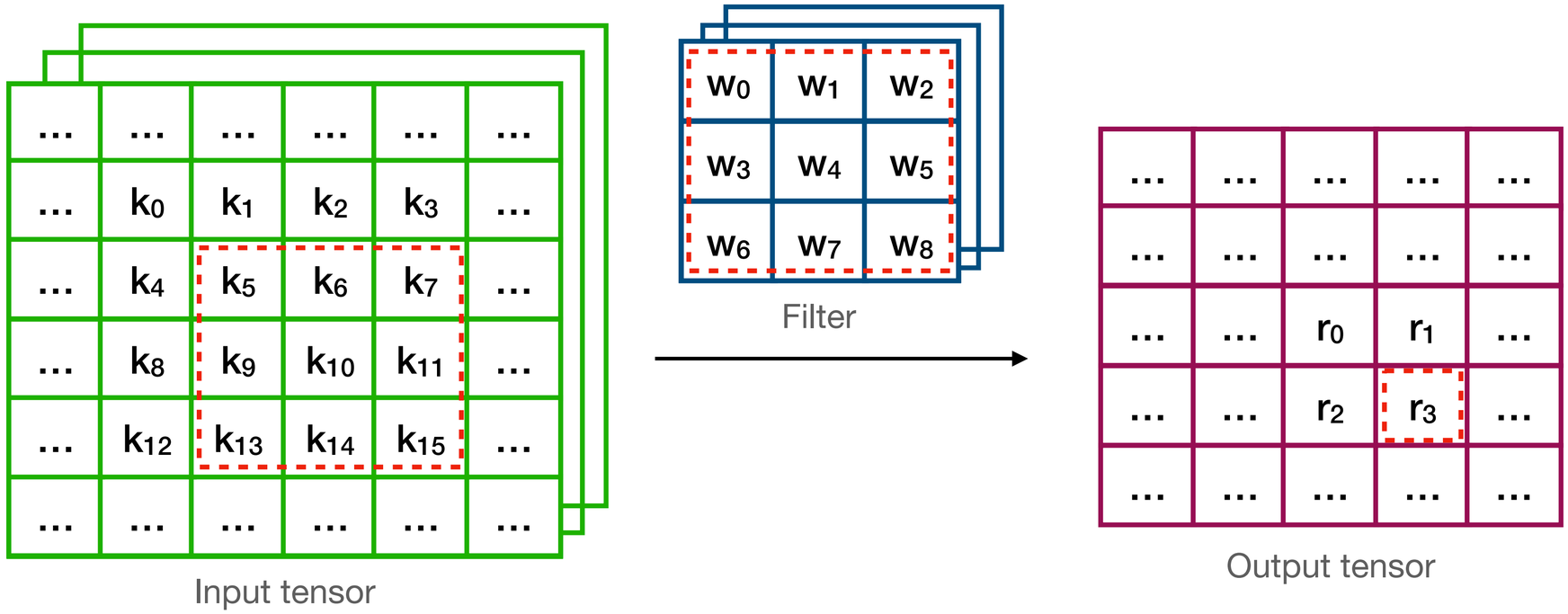
- Direct convolution need 9xCx4 MACs for 4 outputs



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

Winograd Convolution

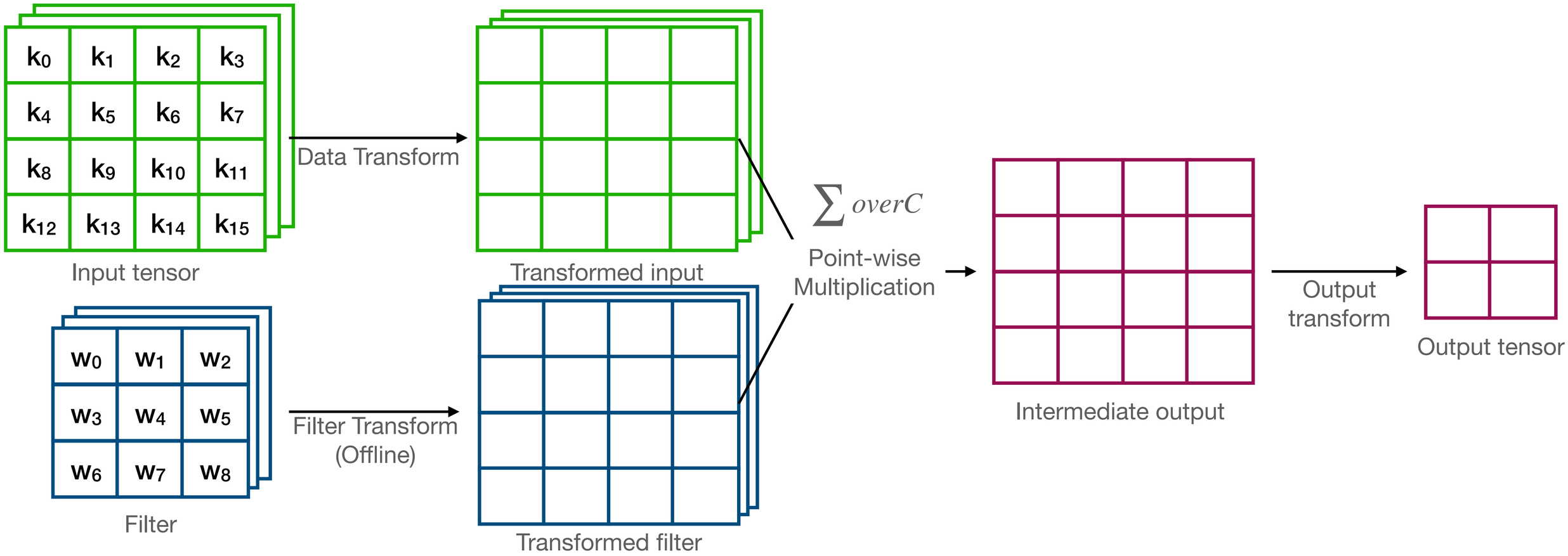
- Direct convolution need 9xCx4 MACs for 4 outputs



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

Winograd Convolution

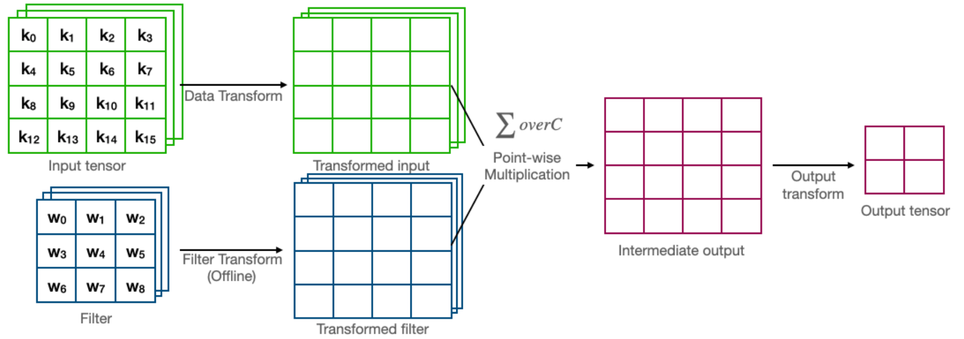
- Direct convolution need 9xCx4 MACs for 4 outputs
- Winograd convolution need only16xC MACs for 4 outputs -> 2.25x fewer MACs



"Fast Algorithms for Convolutional Neural Networks" [Link]

Winograd Convolution

Formula for 3x3 convolution



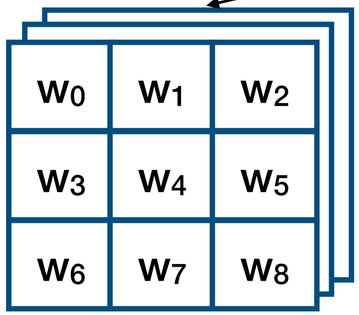
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix} \quad G^T = \begin{bmatrix} 1 & 1/2 & 1/2 & 0 \\ 0 & 1/2 & -1/2 & 0 \\ 0 & 1/2 & 1/2 & 1 \end{bmatrix} \quad B^T = \begin{bmatrix}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

Output transform (Online)

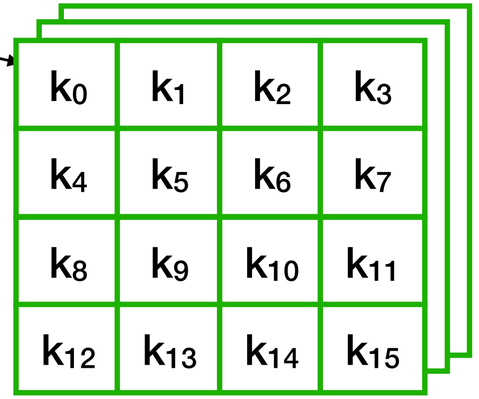
Output tensor ← $Y = A^T[[GgG^T] \odot [B^T dB]]$

Filter Transform (Offline)

Data Transform (Online)



Filter



Input tensor

”Fast Algorithms for Convolutional Neural Networks” [Link]

Welcome to Try TinyEngine and TinyChatEngine

Open Source on GitHub (TODO: update TinyChatEngine once it is public)

Product Team Enterprise Explore Marketplace Pricing

mit-han-lab / **mcunet** Public

Code Issues 6 Pull requests Actions Projects Security Insights

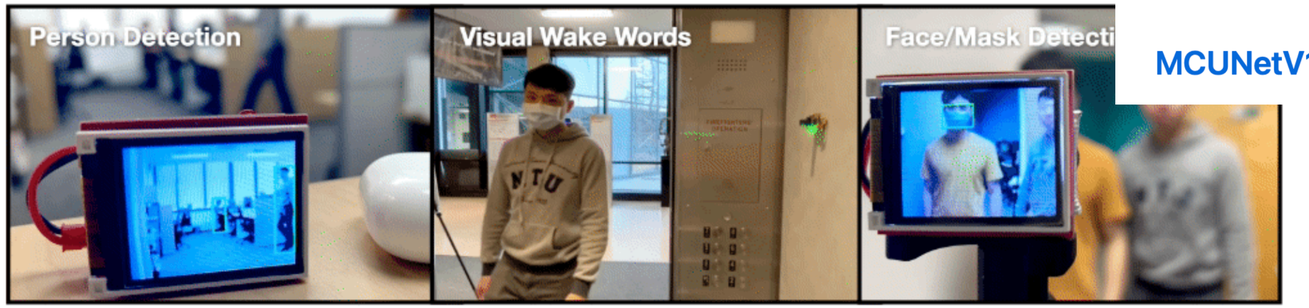
master 2 branches 0 tags Go to

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.

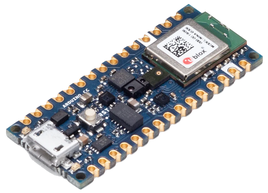
MCUNetV1 | **MCUNetV2** | **MCUNetV3**

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

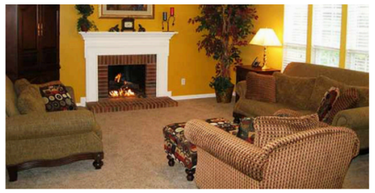
Summary of Today's Lecture

Today we learned:

1. What microcontrollers are and why they are essential.
2. Critical factors of deploying neural networks on edge devices.
   - Limited memory and computing resources.
   - Data layouts/formats of neural networks.
3. Optimization techniques.
   - Loop optimization, Multithreading, SIMD programming, CUDA programming, Im2col convolution, in-place depth-wise convolution, NHWC for point-wise convolution and NCHW for depth-wise convolution, and Winograd convolution.

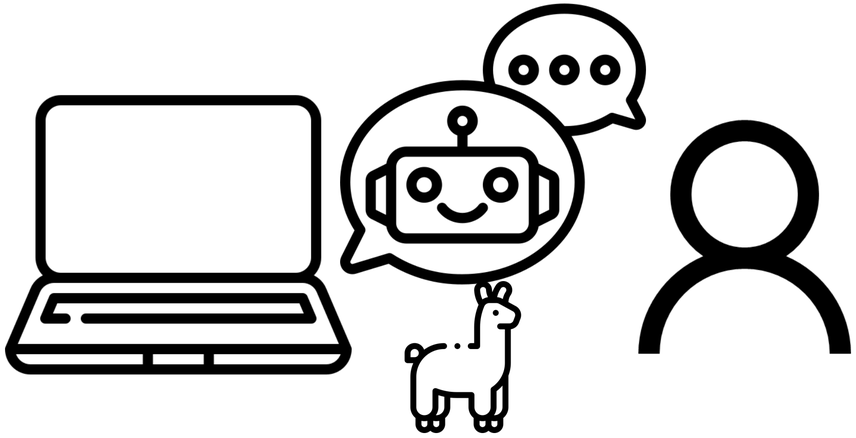


(a) 'Person'



(b) 'Not-person'

Visual wake word



Local chatbot

Lab 4/5 - Quantize & Deploy LLM

In Lab 4/5, we will:

1. Quantize LLM to 4bit using AWQ
2. Deploy a large language model with TinyLLMEngine and run a local chatbot on your computer.
3. Learn how to implement optimization techniques (loop unrolling/reordering, SIMD, Multithreading).
4. Evaluate on-device latency improvement achieved by different optimization techniques.

```
TinyChatEngine by MIT HAN Lab: https://github.com/mit-han-lab/TinyChatEngine
Using model: LLaMA2_7B_chat
Using AWQ for 4bit quantization: https://github.com/mit-han-lab/llm-awq
Loading model... Finished!
USER: tell me 3 attractions in Boston.
ASSISTANT:
Certainly! Here are three popular tourist attractions in Boston, Massachusetts:
1. The Freedom Trail - This historic trail takes you on a journey through some of the most imp
ortant sites in American history, including Old North Church, Faneuil Hall, and Bunker Hill Mo
nument.
2. Fenway Park - Home to the Boston Red Sox baseball team, this iconic stadium is known for it
s green monster wall and lively atmosphere. Visitors can take a guided tour of the park or cat
ch a game during baseball season.
3. Museum of Fine Arts - With over 450,000 works of art spanning 8,000 years of history, this
world-class museum is sure to impress even the most discerning art lover. Highlights include E
uropean and American paintings, sculpture gardens, and a collection of Egyptian antiquities.
I hope you find these recommendations helpful! Is there anything else I can assist you with?
Inference latency, Total time: 13.4 s, 62.7 ms/token, 15.9 token/s, 213 tokens
```