vMCU: Coordinated Memory Management and Kernel Optimization for DNN Inference on MCUs

Size Zheng[1,2], Renze Chen[1], Meng Li[1], Zihao Ye[2],
Luis Ceze[2,3], Yun Liang[1]

[1]*Peking University,*[2]*University of Washington,* [3]*OctoAI*

MLSys 2024

Microcontrollers (MCUs)

■ Microcontrollers can be seen everywhere

Mobile Phone

Television

Modern Vehicles

Industry Robots

Smart Home Devices

■ Microcontrollers are really small

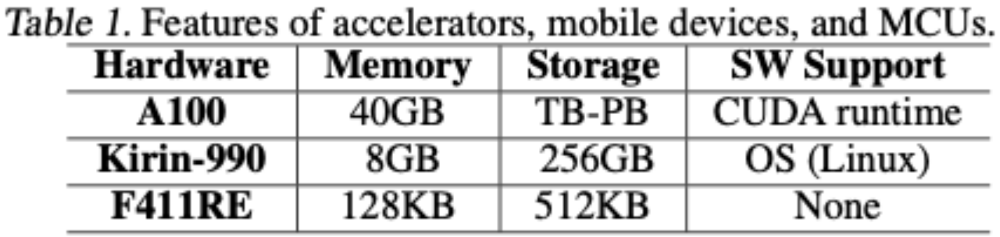
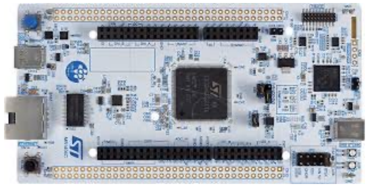


Table 1. Features of accelerators, mobile devices, and MCUs.

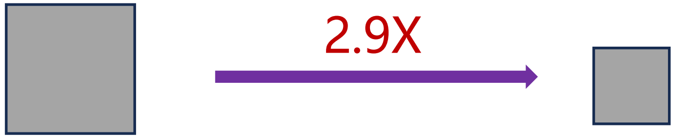
Hardware	Memory	Storage	SW Support
A100	40GB	TB-PB	CUDA runtime
Kirin-990	8GB	256GB	OS (Linux)
F411RE	128KB	512KB	None

The memory of MCU is usually 2-3 orders of magnitude smaller than mobile devices, which makes it challenging to map DNNs onto MCUs

STM32F411RE
RAM: 128KB



STM32F767ZI
RAM: 512KB

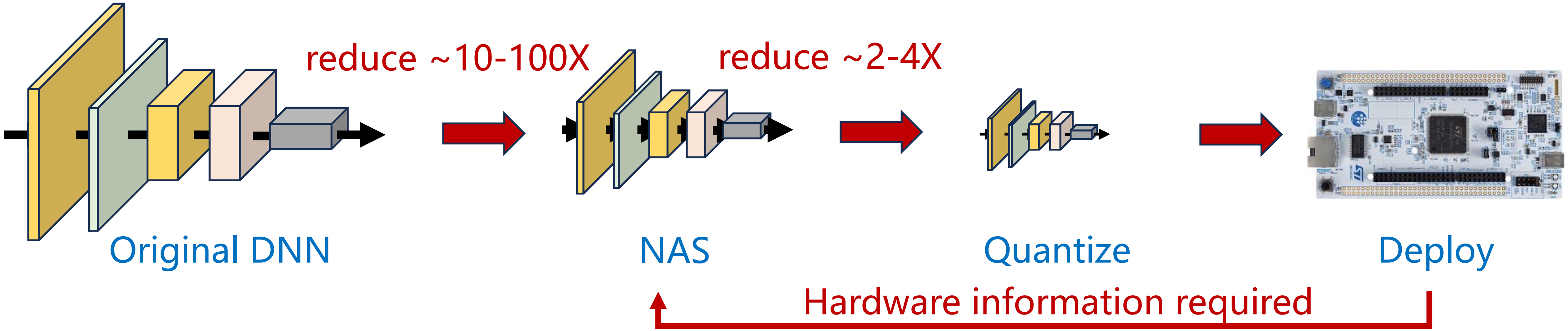


Memory requirements of a single convolution layer

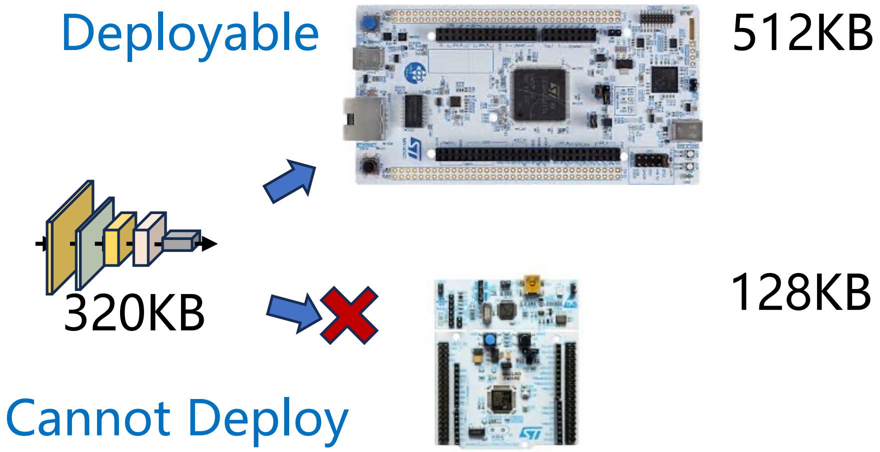
MCU Memory

Deploy DNNs on MCUs

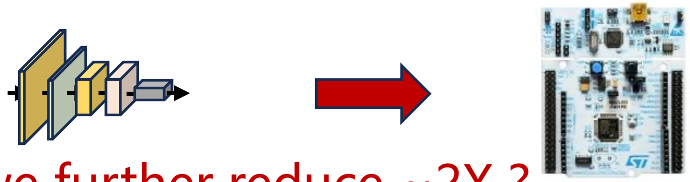
- Various approaches work together



- NAS can be expensive



New NAS process is required for new devices (with lower memory capacity), which often takes hours or days on GPU.

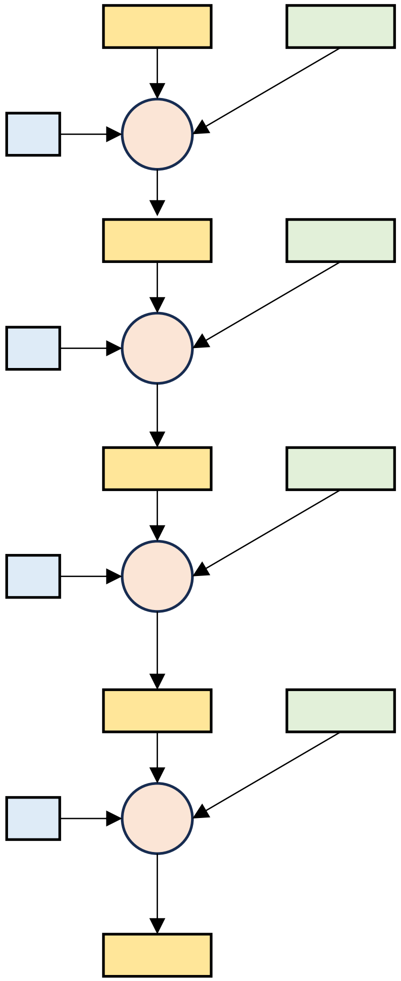


can we further reduce ~2X ?

How to address the “last mile” issue?

Tensor-level Memory Management

■ Understanding the memory consumption of DNN



Static Memory Footprint

∑ 🟩

sum of weight tensors

Dynamic Memory Footprint

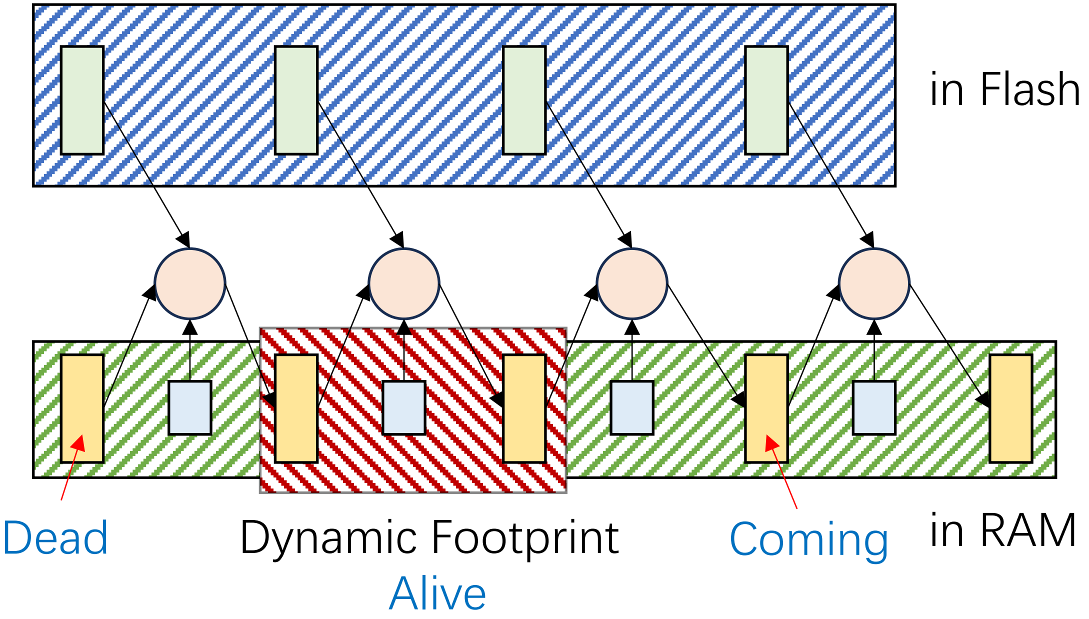
max{🟦,🟨} ≤ ? ≤ ∑ 🟦 + 🟨

depends on scheduling and optimization methods

- 🟨 Activation Tensor
- 🟩 Weight Tensor
- 🟦 Workspace Tensor
- ● DNN Operator

Facts:

- Static memory footprint remains the same
- Execution order of operators affect dynamic memory footprint
- Static memory can be put in read-only Flash
- Minimize dynamic memory foot is important

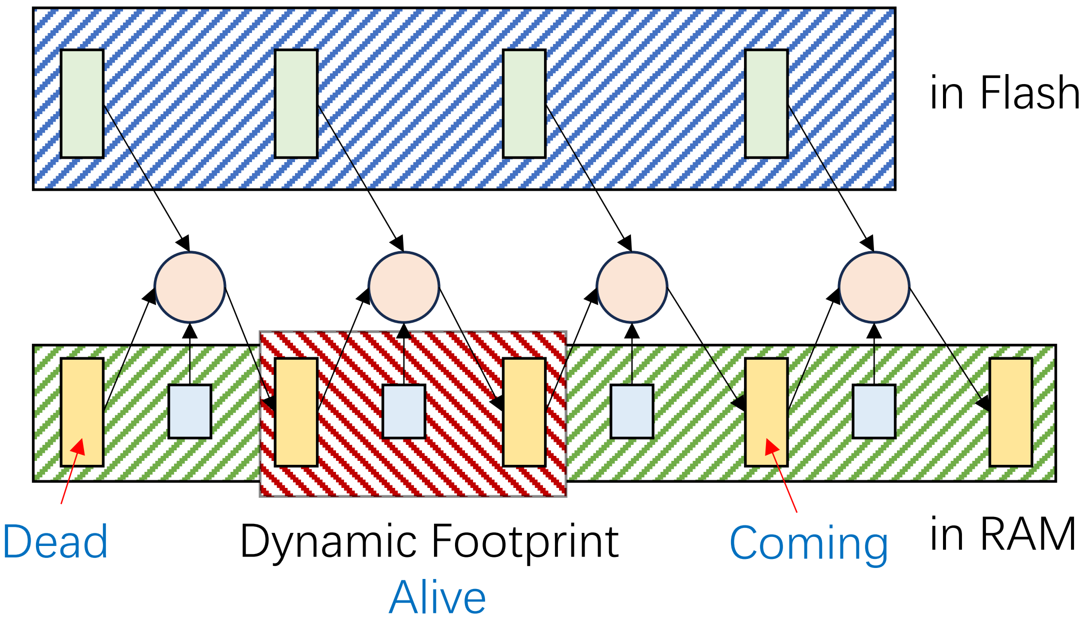


Can we do better?

■ Closer to the lower bound

Dynamic Memory Footprint

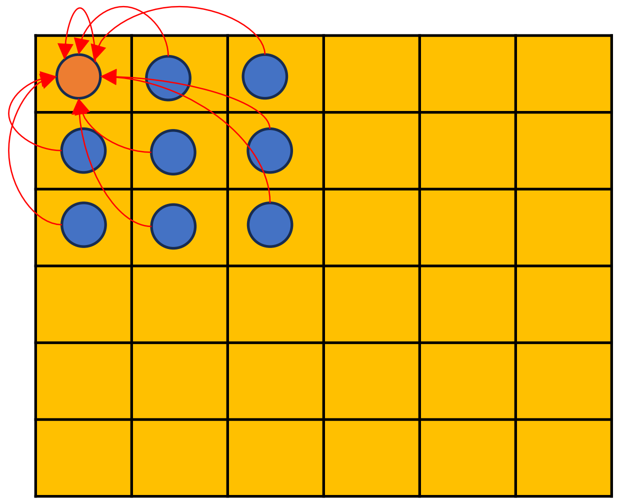
max{□,▭} ≤ max{▯ + ▯ + □} ≤ ∑□ + ▭



For certain operators, we can!

```
Out[h,w,k]+=
     In[h+r,w+s,k]*Weight[r,s,k]
```

In-place update



Input and Output use the same tensor

A special optimization in MCUNet-V2

Lin, J. et al. (2021). "Memory- efficient patch-based inference for tiny deep learning ." In: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021

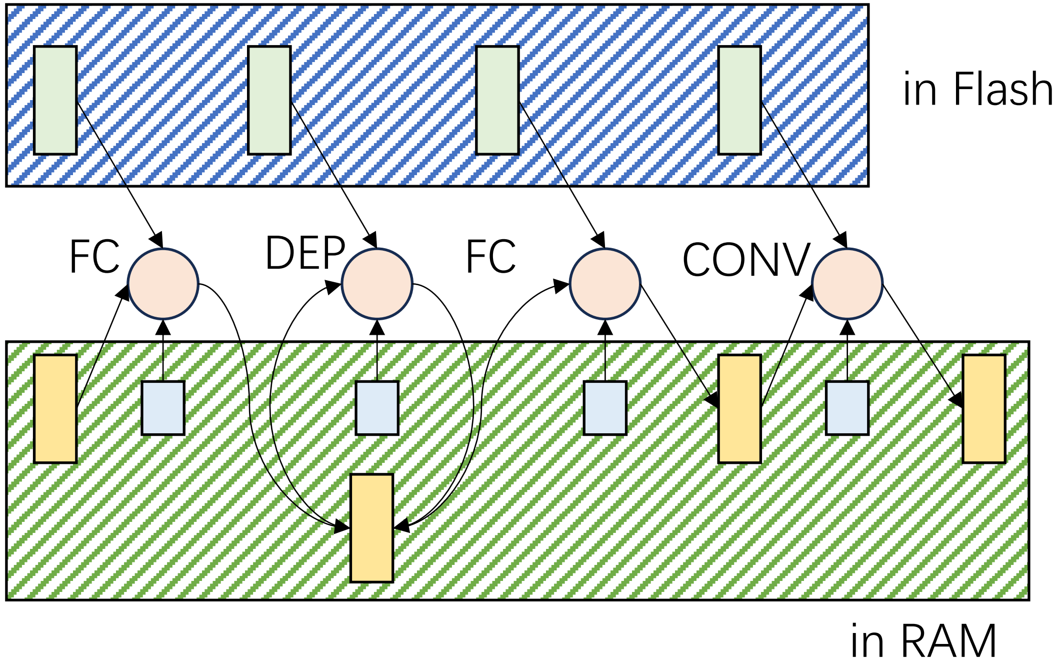
■ However…

max{▯ + ▯ + □} cannot be reduced due to the existence of FC and Conv layers…

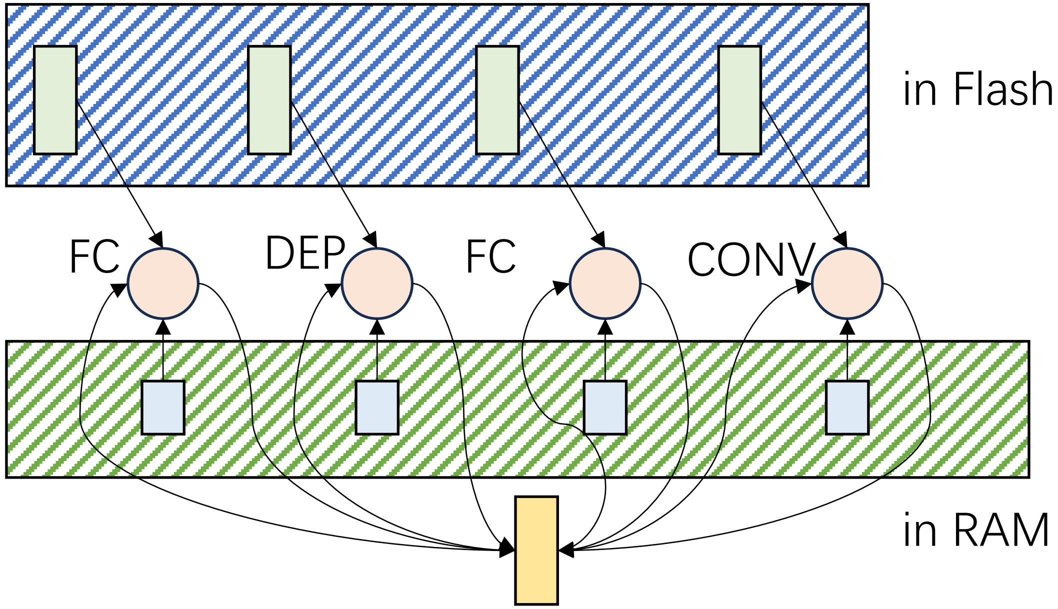
Can we do better?

■ Our aim: make in-place optimization general!

Previous: only certain operators can exploit in-place optimization, the total dynamic footprint remains the same



Ours: make in-place optimization feasible to every operator, reducing the dynamic footprint

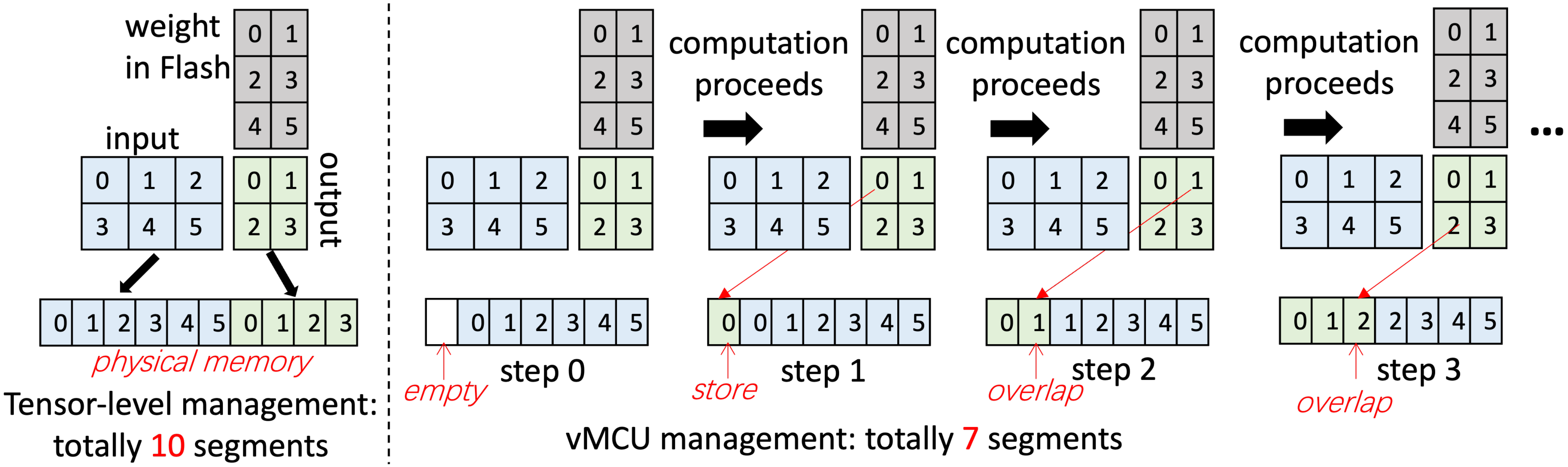


max{ 🟨 + 🟨 + 🟦 } → max{ 🟦 } + max{ 🟨 }

further reduce ~2X

vMCU: Segment-level Memory Management

■ General in-place optimization requires fine-grained memory management



Tensor-level management: totally 10 segments

vMCU management: totally 7 segments

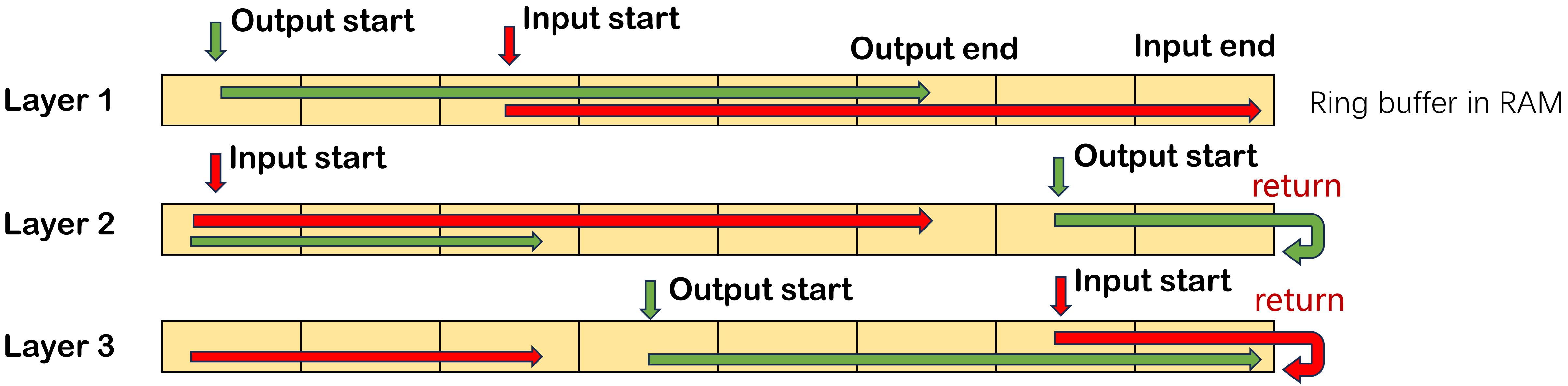
Compare Tensor-level and Segment-level Management using Fully Connected Layer

■ Input and output share the same memory buffer

- Input and output use different read/write pointers
- Make sure that write pointer never steps on alive input data

Memory Abstraction: Ring Buffer

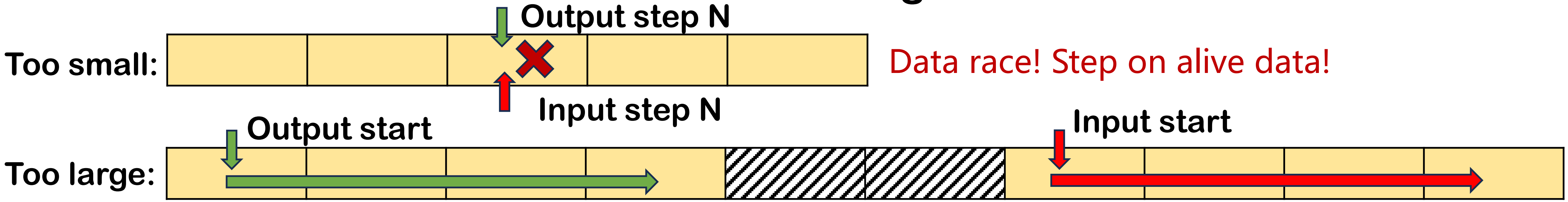
- Circular reuse of limited memory resource



Output start
Input start
Output end
Input end
Layer 1
Ring buffer in RAM
Input start
Output start
return
Layer 2
Output start
Input start
return
Layer 3

The output of previous layer is the input of the next layer

- Question: what is the minimal size of ring-buffer?



Output step N
Too small:
Data race! Step on alive data!
Input step N
Output start
Input start
Too large:

Waste of memory! (back to tensor-level management)

Problem Formalization

- **Key insight: memory access can be formulated as a linear system**

Ring Buffer

$$Pool[\frac{MemCap}{Seg}], \qquad Pool[addr] = Pool[addr \ \% \ \frac{MemCap}{Seg}].$$

where $Pool[i]$ is a vector of Seg bytes, $0 \leq i < \frac{MemCap}{Seg}$

Iteration Domain

$$\{S[\vec{i}] : \mathbf{H}\vec{i} + \vec{B} < 0\}$$

Access Function

$$\{S[\vec{i}] \rightarrow T[\vec{u}] : \vec{u} = \mathbf{A_u}\vec{i} + \vec{V}_u\}$$

Memory Mapping

$$\{S[\vec{i}] \rightarrow T[\vec{u}] \rightarrow Pool[addr] : addr = \vec{L}_{addr}\vec{u} + b_{off}\}$$

Correctness Condition

$$\vec{L}_{In}(\mathbf{A}_{In}\vec{i} + \vec{V}_{In}) + b_{In} \geq \vec{L}_{Out}(\mathbf{A}_{Out}\vec{j} + \vec{V}_{Out}) + b_{Out} \qquad \forall \vec{j} \leq \vec{i}$$

Optimization Target

$$\text{min. } b_{In} - b_{Out}$$

Use GEMM as Example

a) GEMM:

for m,n,k in ranges(M,N,K):

S: Out[m, n] += In[m, k] * W[k, n]

b) Iteration domain:

$$\{S[m,n,k] \mid \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}m\\n\\k\end{bmatrix} - \begin{bmatrix}M\\N\\K\end{bmatrix} < 0\}$$

c) Access functions:

$$S[m,n,k] \rightarrow In[m,k]$$
$$S[m,n,k] \rightarrow Out[m,n]$$

d) Memory mapping:

$$In[m,k] \rightarrow Pool[m*K+k+b_{In}]$$
$$Out[m,n] \rightarrow Pool[m*N+n+b_{Out}]$$

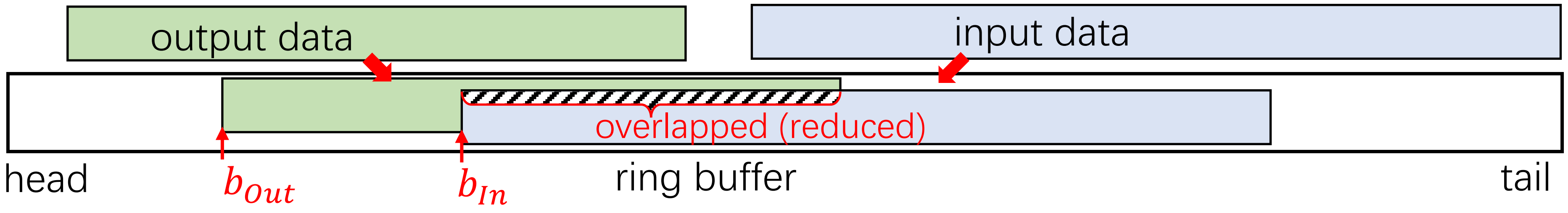
e) Access matrices:

In: $\begin{bmatrix}1&0&0\\0&0&1\end{bmatrix}$ Out: $\begin{bmatrix}1&0&0\\0&1&0\end{bmatrix}$

f) Mapping vectors:

In: $[K \quad 1]$ offset: b_{In}

Out: $[N \quad 1]$ offset: b_{Out}



Optimization Problem:

$$\min.\ b_{In} - b_{Out}$$
$$s.t.(K-N)m - n + k \geq b_{Out} - b_{In}$$
$$0 \leq m < M \qquad 0 \leq n < N \qquad 0 \leq k < K$$

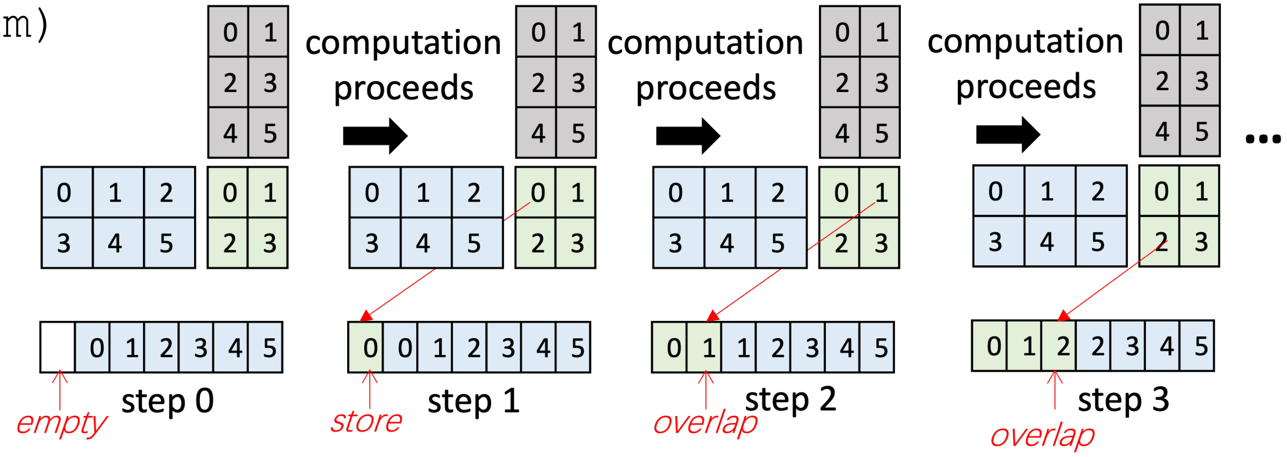
Optimal mapping solution:

$$b_{In} - b_{Out} = \min\{N,K\} - 1$$

Implementation for GEMM

Kernel Design

```
for m=0 to M step 1: //Outer level tiling
 for n=0 to N step Seg:
  Accum = RegAlloc(Seg,0) //Zero Register array of size Seg
  for k=0 to K step Seg:
   ValueA = RAMLoad(In[m,k:k+Seg])
   ValueB = FlashLoad(Weight[k:k+Seg,n:n+Seg])
   for ki=0 to Seg step KI: //Inner level tiling
    for ni=0 to Seg step NI:
     Res = Dot(ValueA[ki:ki:KI],ValueB[ki:ki+KI,ni:ni+NI])
     Accum[ni:ni+NI] += Res
  RAMStore(Out[m,n:n+Seg], Accum)
 for k=0 to K step Seg:
  RAMFree(In[m,k:k+Seg])
```

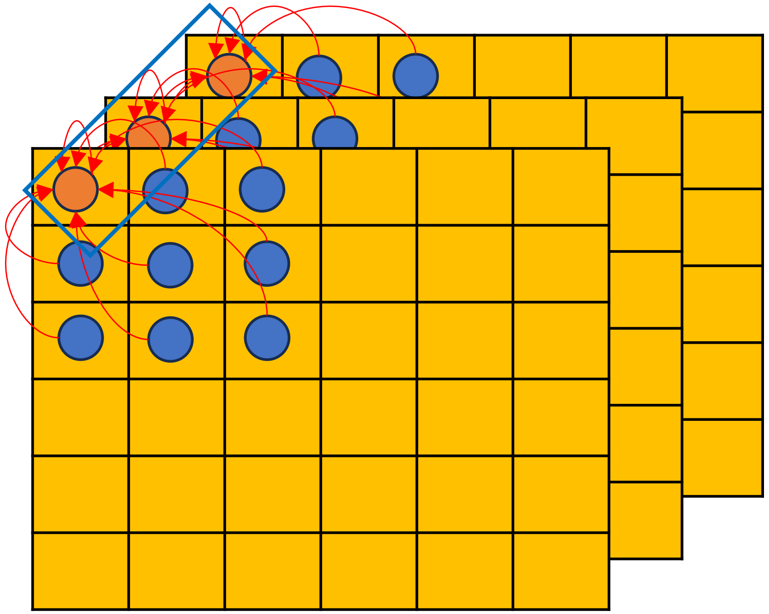


Implementation for Convolution

■ Kernel Design

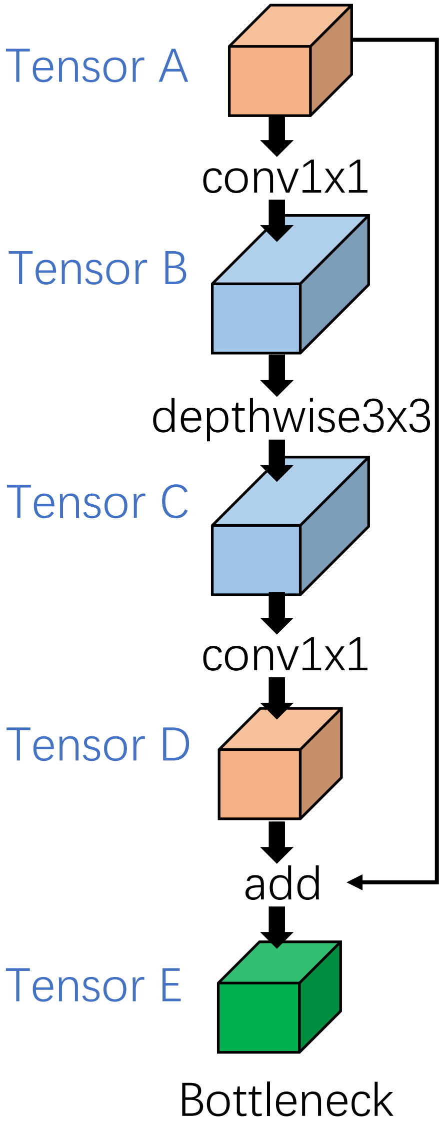
```
for n=0 to N step 1: //Outer level tiling
 for p=0 to P step 1:
  for q=0 to Q step 1:
   for k=0 to K step Seg:
    Accum = RegAlloc(Seg,0) //Register array of size Seg
    for r=0 to R step 1:
     for s=0 to S step 1:
      for c=0 to C step Seg:
       ValA = RAMLoad(In[n,p+r,q+s,c:c+Seg])
       ValB = FlashLoad(Weight[r,s,c:c+Seg,k:k+Seg])
       for ci=0 to Seg step CI: //Inner level tiling
        for ki=0 to Seg step KI:
         Res = Dot(ValA[ci:ci+CI],ValB[ci:ci+CI,k:ki+KI])
         Accum[ki:ki+KI] += Res
    RAMStore(Out[n,p,q,ki:ki+KI], Accum)
   for c=0 to C step Seg:
    RAMFree(In[n,p,q,c:c+Seg])
```

Convolution is a hybrid of depthwise and GEMM



Handle Multiple Layers

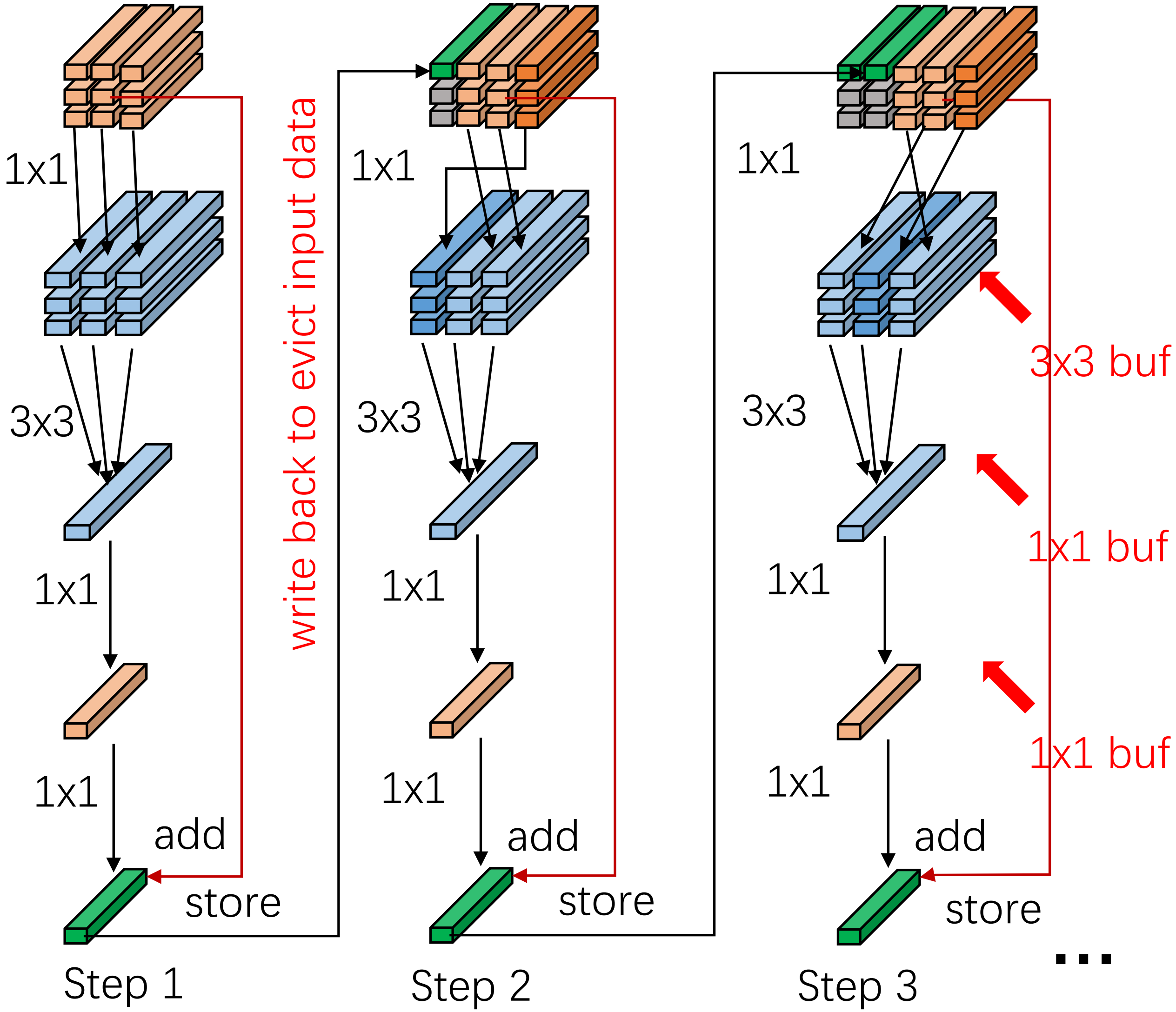
- Network topology also affect memory footprint



Skip connection causes Tensor A alive for a long time

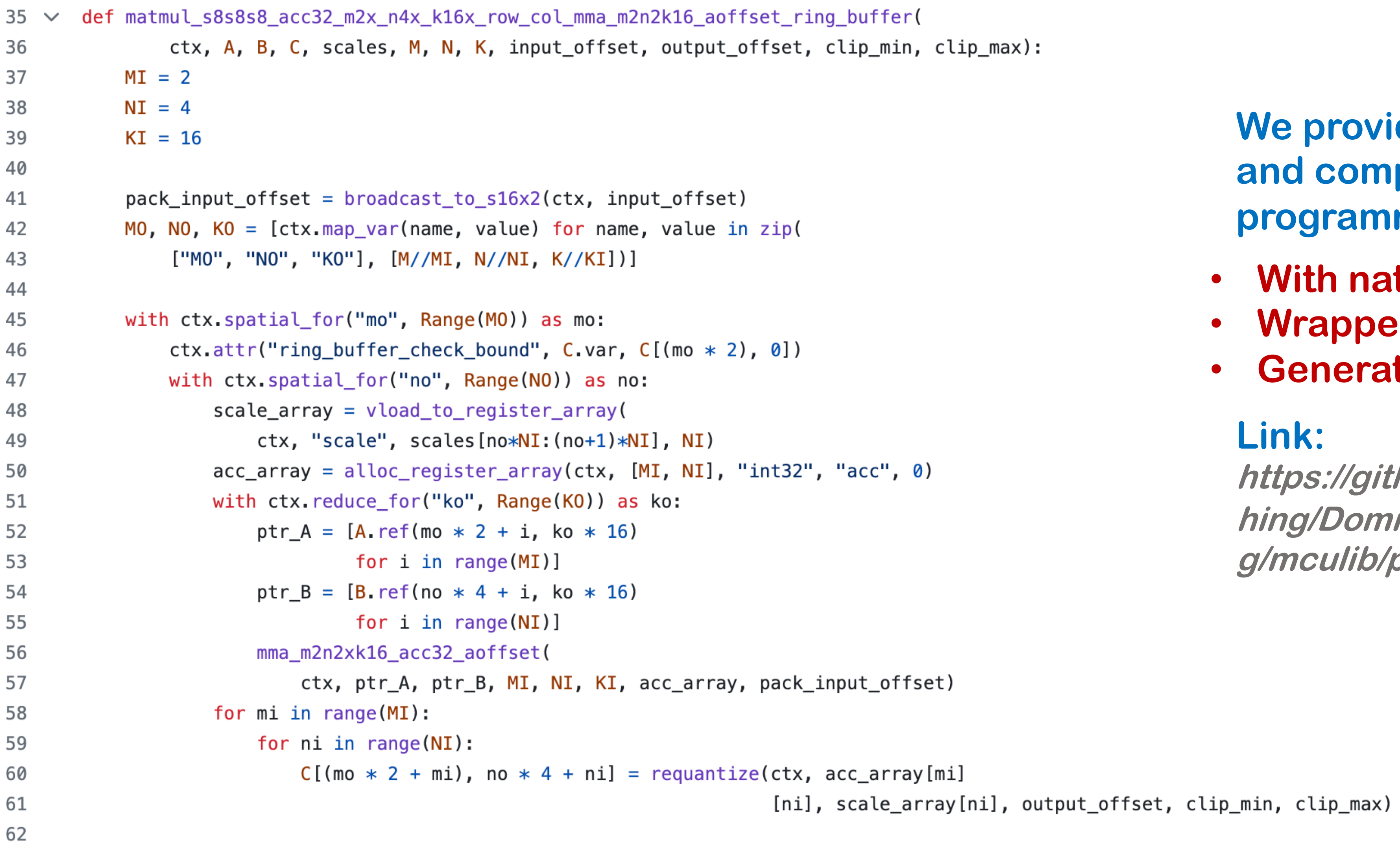
If we can fuse these layers into one operator, we can effectively reduce memory footprint from sizeof(Tensor A) + sizeof(Tensor B) + sizeof(Tensor C) to sizeof(Tensor A), which can be >2X

Fusion



Compiler Support

- Write kernels for MCU can be annoying…



```
def matmul_s8s8s8_acc32_m2x_n4x_k16x_row_col_mma_m2n2k16_aoffset_ring_buffer(
        ctx, A, B, C, scales, M, N, K, input_offset, output_offset, clip_min, clip_max):
    MI = 2
    NI = 4
    KI = 16

    pack_input_offset = broadcast_to_s16x2(ctx, input_offset)
    MO, NO, KO = [ctx.map_var(name, value) for name, value in zip(
        ["MO", "NO", "KO"], [M//MI, N//NI, K//KI])]

    with ctx.spatial_for("mo", Range(MO)) as mo:
        ctx.attr("ring_buffer_check_bound", C.var, C[(mo * 2), 0])
        with ctx.spatial_for("no", Range(NO)) as no:
            scale_array = vload_to_register_array(
                ctx, "scale", scales[no*NI:(no+1)*NI], NI)
            acc_array = alloc_register_array(ctx, [MI, NI], "int32", "acc", 0)
            with ctx.reduce_for("ko", Range(KO)) as ko:
                ptr_A = [A.ref(mo * 2 + i, ko * 16)
                         for i in range(MI)]
                ptr_B = [B.ref(no * 4 + i, ko * 16)
                         for i in range(NI)]
                mma_m2n2xk16_acc32_aoffset(
                    ctx, ptr_A, ptr_B, MI, NI, KI, acc_array, pack_input_offset)
            for mi in range(MI):
                for ni in range(NI):
                    C[(mo * 2 + mi), no * 4 + ni] = requantize(ctx, acc_array[mi]
                                                               [ni], scale_array[ni], output_offset, clip_min, clip_max)

```

We provide a Python DSL and compiler support for programming MCU kernels

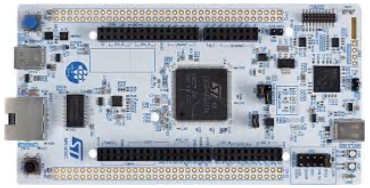
- With native Python interpreter
- Wrapper for MCU SIMD intrinsic
- Generate C code for MCU

Link: *https://github.com/KnowingNothing/Domino/tree/master/testing/mculib/python/mculib*

Evaluation Setup

- We use two MCU dev board for evaluation

STM32F411RE
RAM: 128KB



STM32F767ZI
RAM: 512KB

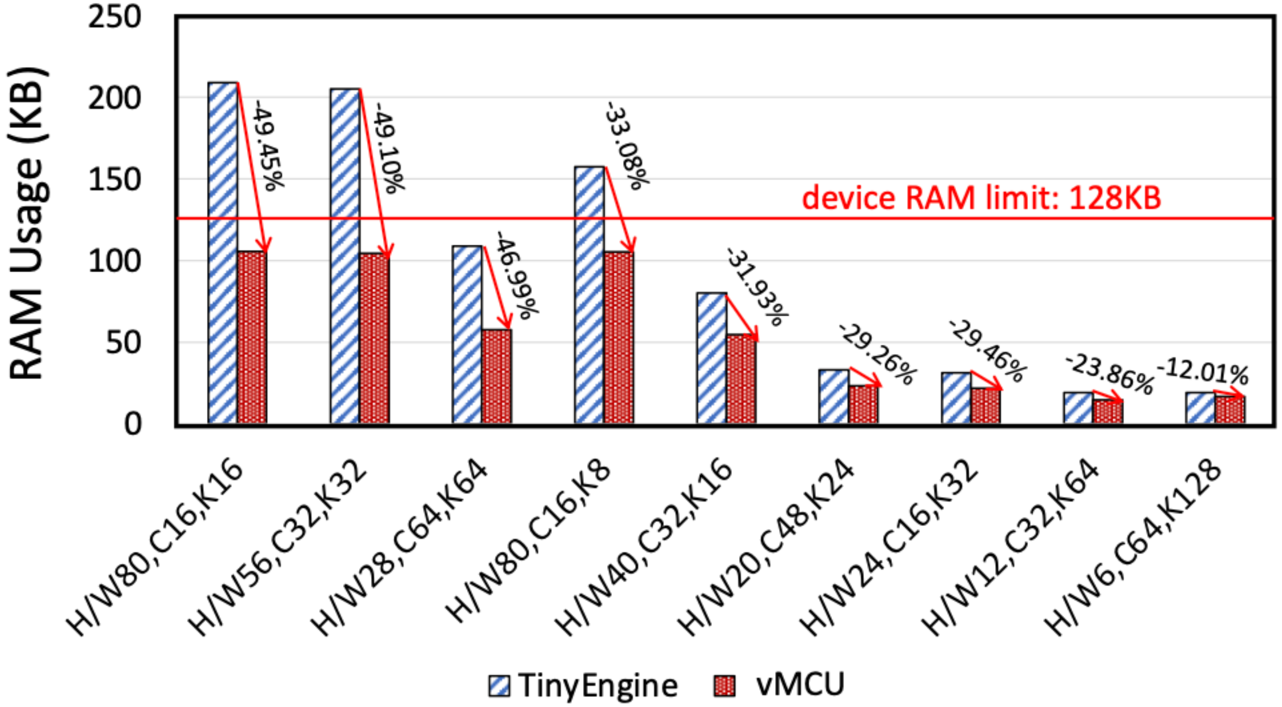
- Baseline: TinyEngine (from MCUNet)

- Benchmark:

Name	H/W	C_in	C_mid	C_out	R/S	strides
MCUNet-5fps-VWW						
S1	20	16	48	16	3	1,1,1
S2	20	16	48	16	3	1,1,1
S3	10	24	144	16	3	1,1,1
S4	10	24	120	24	3	1,1,1
S5	5	40	240	40	3	1,1,1
S6	5	48	192	48	3	1,1,1
S7	3	96	480	96	3	1,1,1
S8	3	96	384	96	3	1,1,1
MCUNet-320KB-ImageNet						
B1	176	3	16	8	3	2,1,1
B2	88	8	24	16	7	1,2,1
B3	44	16	80	16	3	1,1,1
B4	44	16	80	16	7	1,1,1
B5	44	16	64	24	5	1,1,1
B6	44	16	80	24	5	1,2,1
B7	22	24	120	24	5	1,1,1
B8	22	24	120	24	5	1,1,1
B9	22	24	120	40	3	1,2,1
B10	11	40	240	40	7	1,1,1
B11	11	40	160	40	5	1,1,1
B12	11	40	200	48	7	1,2,1
B13	11	48	240	48	7	1,1,1
B14	11	48	240	48	3	1,1,1
B15	11	48	288	96	3	1,2,1
B16	6	96	480	96	7	1,1,1
B17	6	96	384	96	3	1,1,1

Single Operator Evaluation

■ Pointwise convolution (GEMM) memory reduction

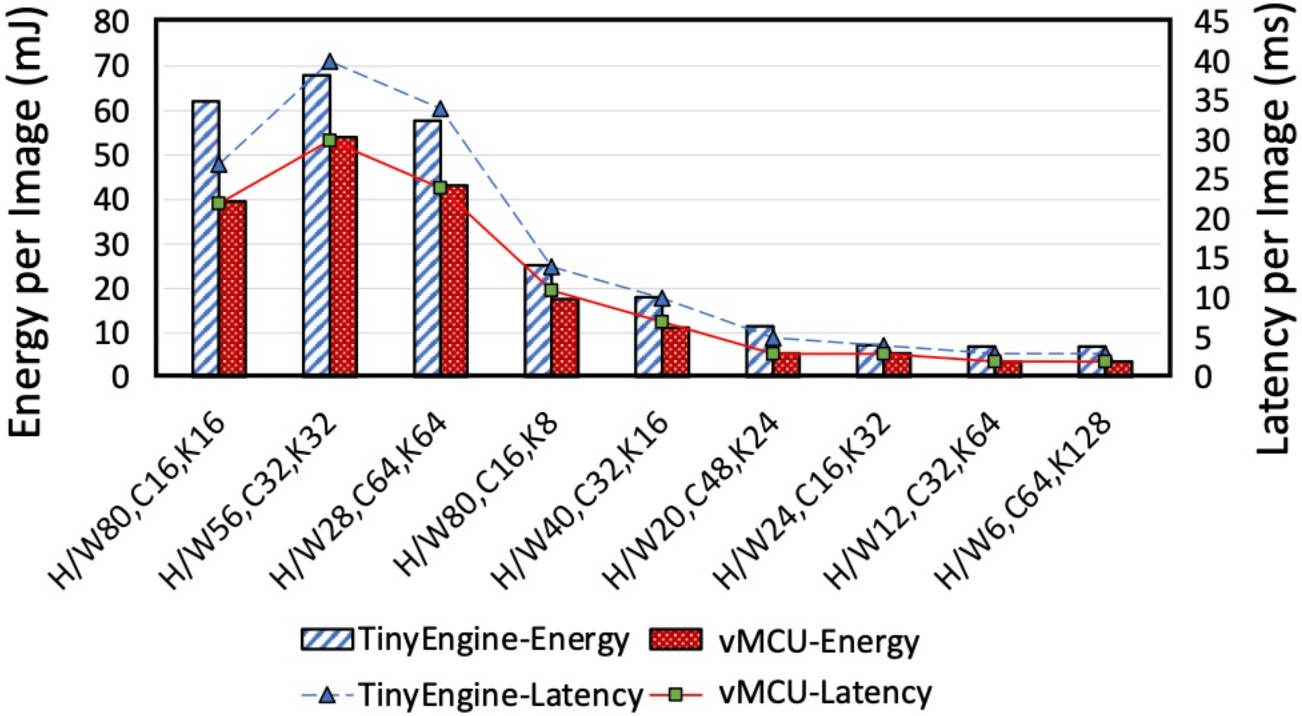


- Reduce 12%-49.5% memory consumption (~2X)

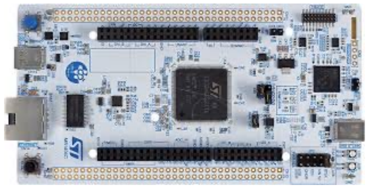
STM32F411RE
RAM: 128KB

Single Operator Evaluation

■ Pointwise convolution (GEMM) energy and latency reduction



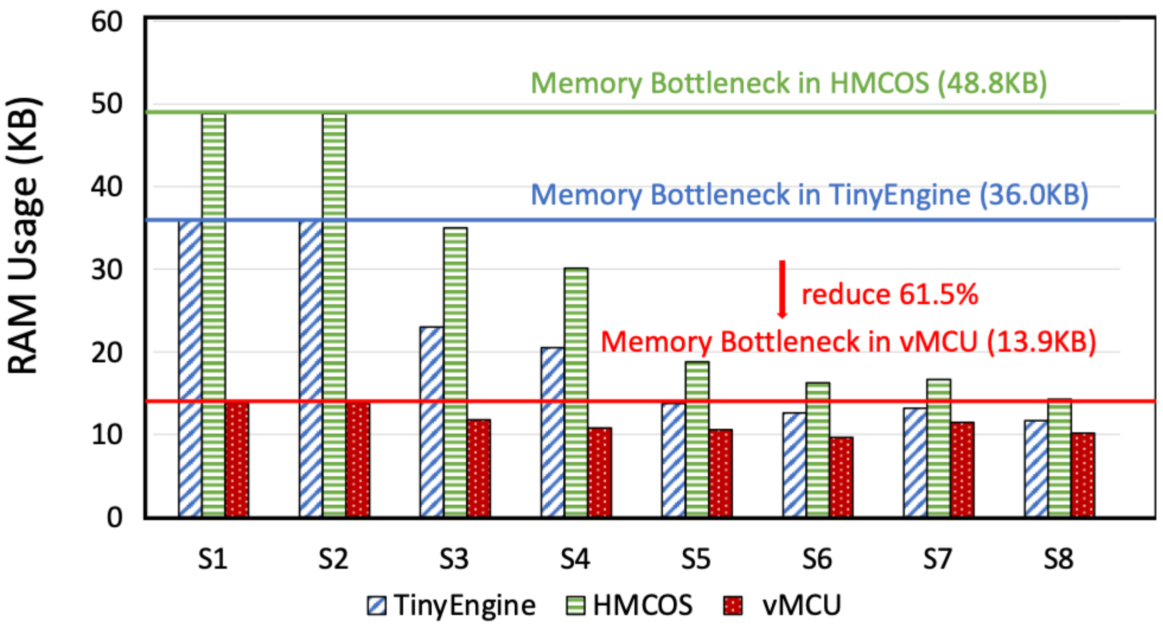
- Reduce 20%-53% energy (~2X)
- Reduce 18%-40% latency

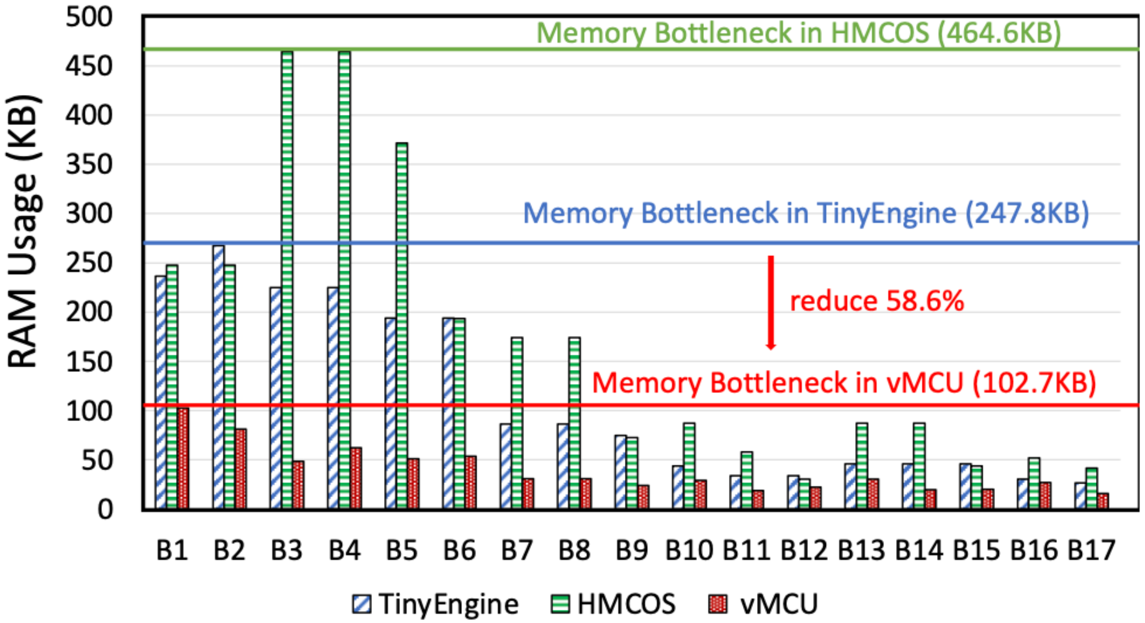


STM32F767ZI
RAM: 512KB

Multi-Layer Evaluation

■ Inverted bottleneck in MCUNet

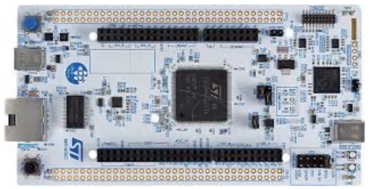




- MCUNet-5fps-VWW
- Whole DNN reduce 61.5% memory

STM32F411RE
RAM: 128KB

- MCUNet-320KB-ImageNet
- Whole DNN reduce 58.6% memory



STM32F767ZI
RAM: 512KB

Summary

■ Segment-level memory management

- Generalize in-place optimization to GEMM and convolution
- Use ring-buffer and linear-system to minimize memory requirements
- Design kernels and provide Python DSL for programming

■ Future work

- LLM on MCU? Flash-attention, MLP…
- Multiple MCU cooperation…
- Leverage Flash to swap data…

Thanks for your attention!