6.5930/1
Hardware Architectures for Deep Learning

Vectorized Kernel Computation

February 26, 2024

Joel Emer and Vivienne Sze

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

Goals of Today's Lecture

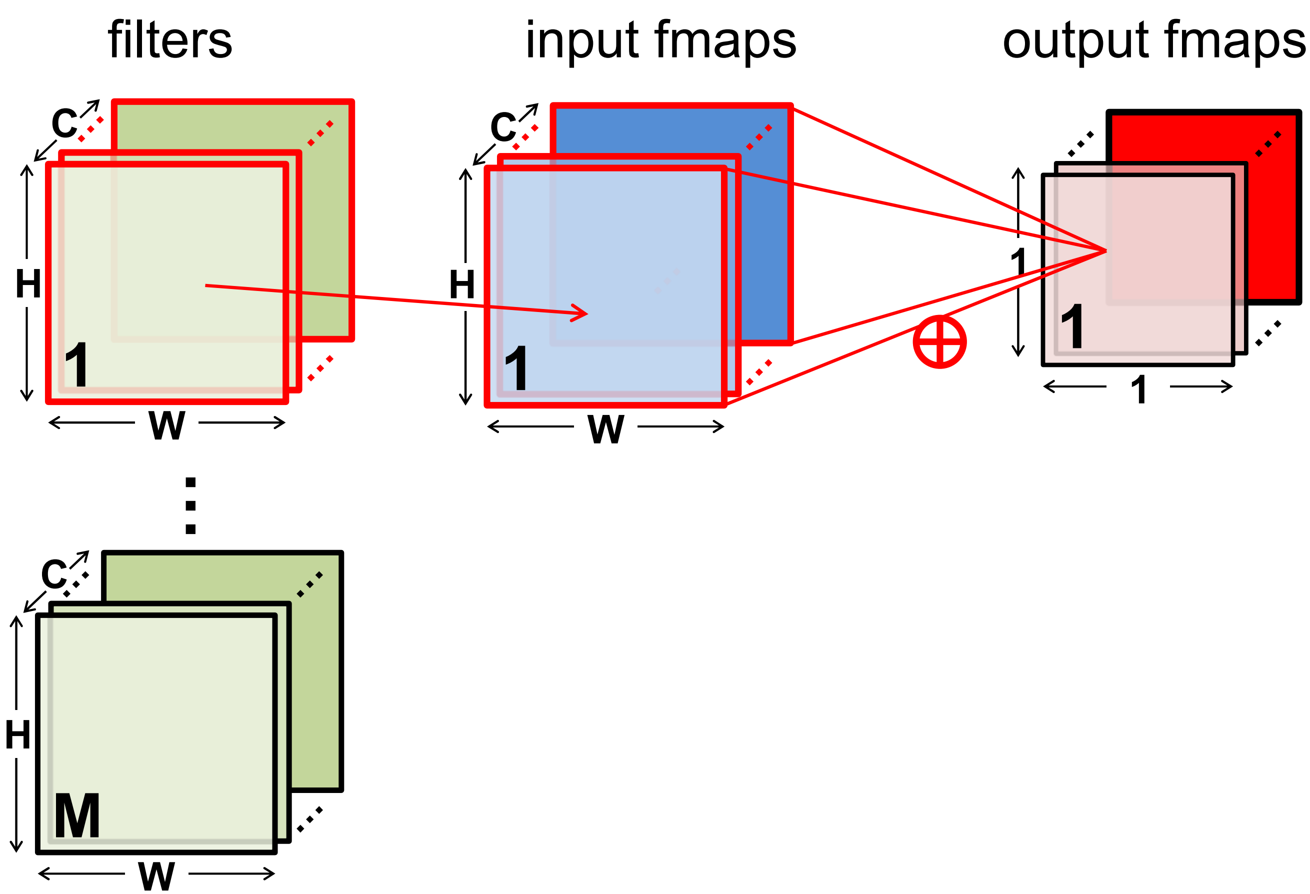
- Understand parallelism and improved efficiency through:
  - loop unrolling, and
  - vectorization

Background Reading

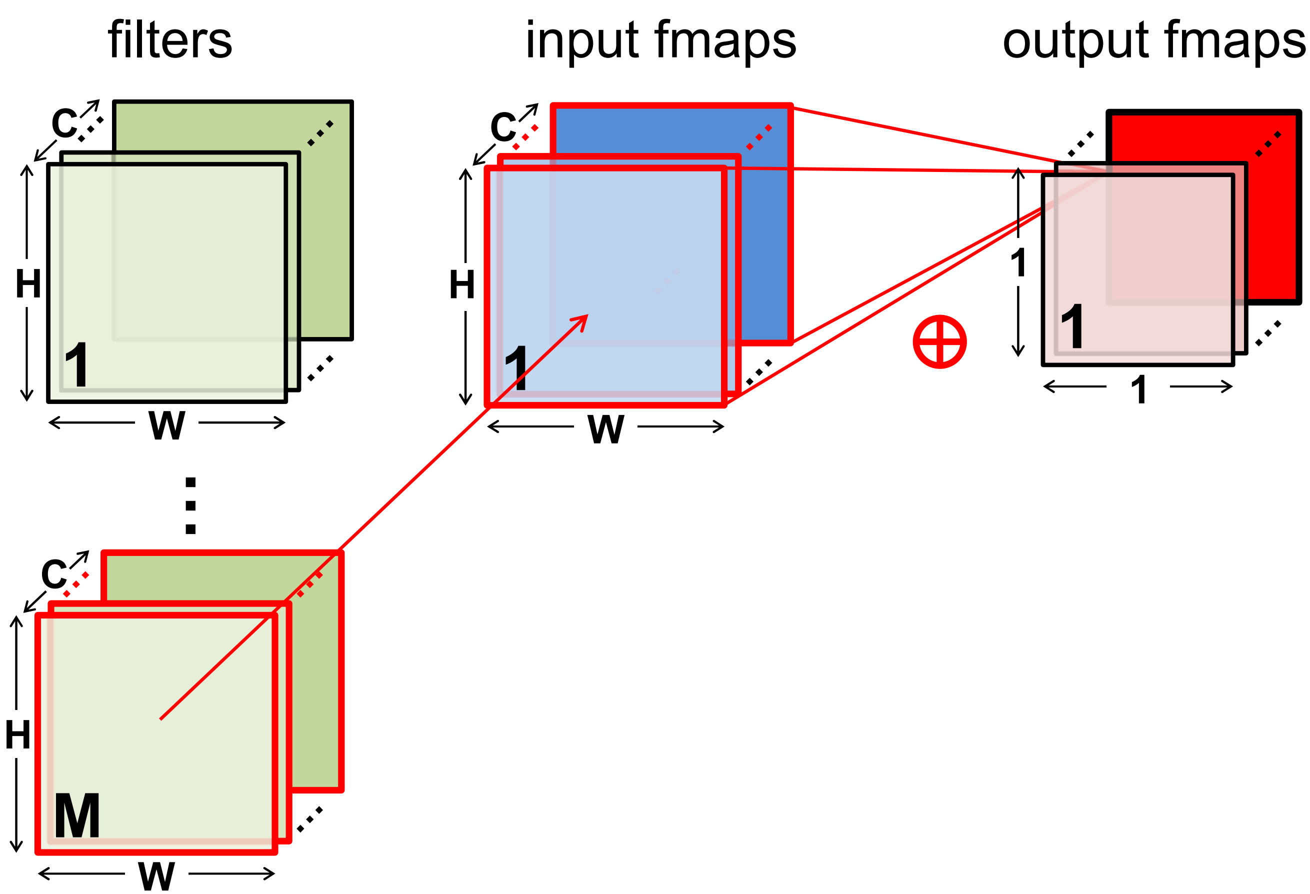
- **Vector architectures**
  - ***Computer Architecture: A Quantitative Approach*, 6th edition, by Hennessy and Patterson**
    - **Ch 4: p282-310, App G**
    - **Ch 4: p262-288, App G**

These books and their online/e-book versions are available through MIT libraries.

Fully Connected Computation

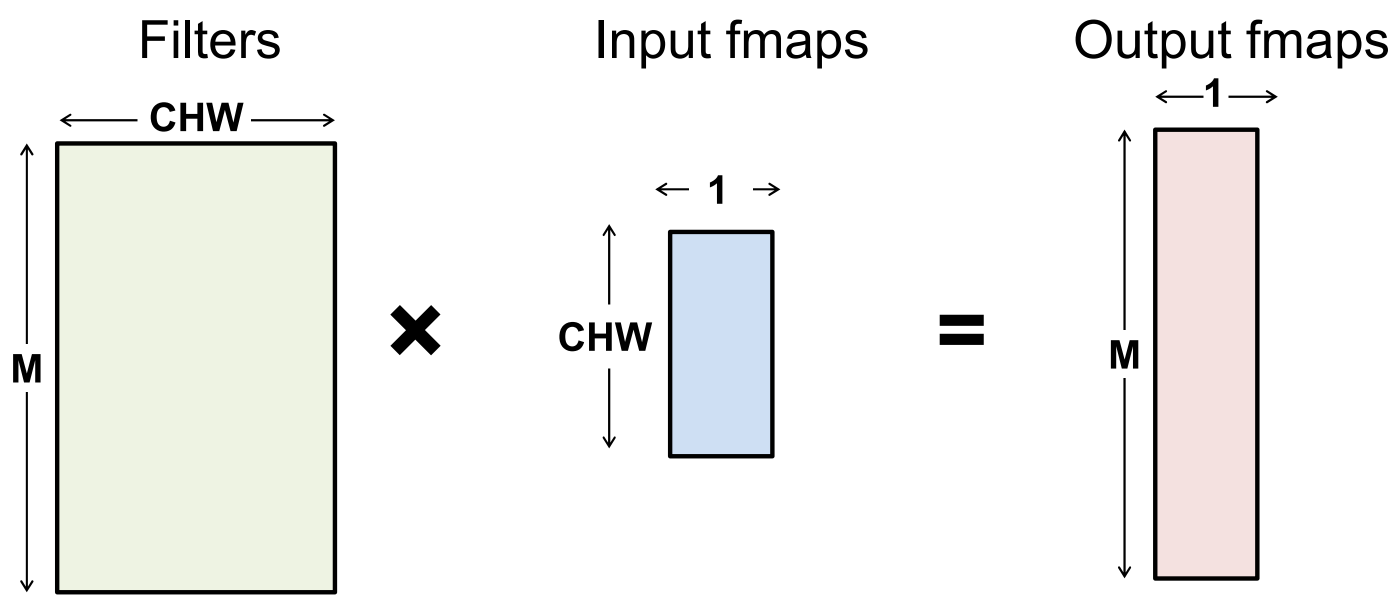


Fully Connected Computation

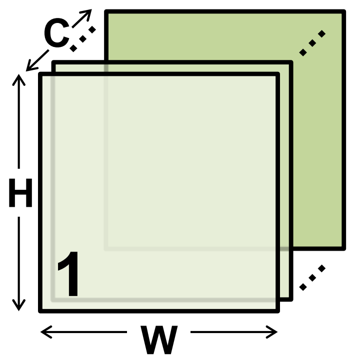


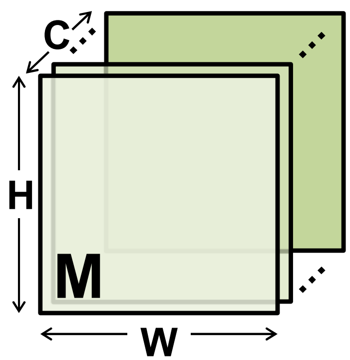
Fully-Connected (FC) Layer - Flattened

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



Filter Memory Layout





Filter weight

Next row

Next input channel

Next output channel

$F[M_0\,C_0\,H_0\,W_0]$ $F[M_0\,C_0\,H_0\,W_1]$ …
$F[M_0\,C_0\,H_1\,W_0]$ $F[M_0\,C_0\,H_1\,W_1]$ …
$F[M_0\,C_0\,H_2\,W_0]$ $F[M_0\,C_0\,H_2\,W_1]$ …
.
.
$F[M_0\,C_1\,H_0\,W_0]$ $F[M_0\,C_1\,H_0\,W_1]$ …
$F[M_0\,C_1\,H_1\,W_0]$ $F[M_0\,C_1\,H_1\,W_1]$ …
$F[M_0\,C_1\,H_2\,W_0]$ $F[M_0\,C_1\,H_2\,W_1]$ …
.
.
.
$F[M_1\,C_0\,H_0\,W_0]$ $F[M_1\,C_0\,H_0\,W_1]$ …
$F[M_1\,C_0\,H_1\,W_0]$ $F[M_1\,C_0\,H_1\,W_1]$ …
$F[M_1\,C_0\,H_2\,W_0]$ $F[M_1\,C_0\,H_2\,W_1]$ …
.
.
.

Flattened FC Loops

Flattened tensors

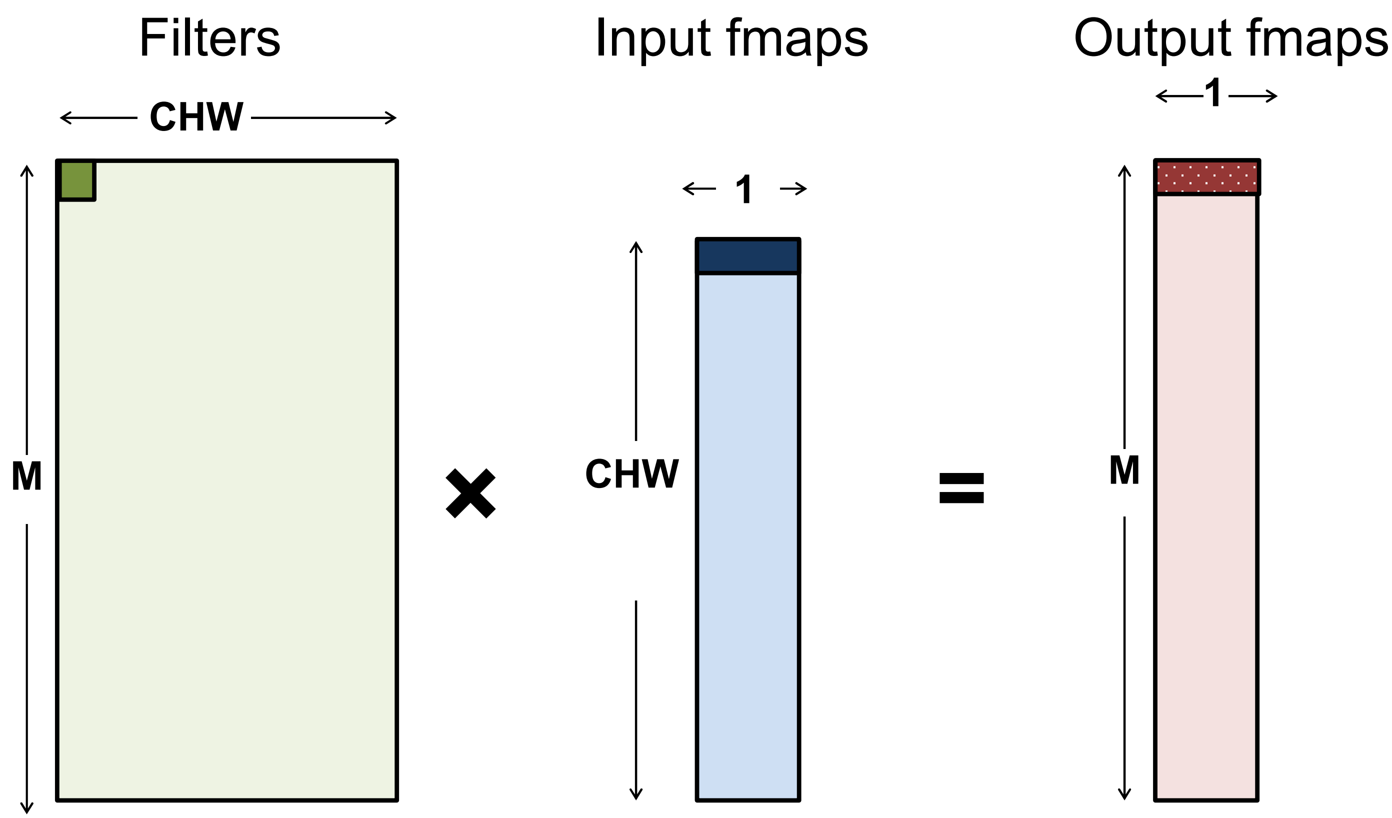
```
int i[CHW];        # Input activations
int f[M*CHW];      # Filter Weights
int o[M];            # Output activations

CHWm = -CHW
for m in [0, M):
  o[m] = 0
  CHWm += CHW
  for chw in [0, CHW):
     o[m] += i[chw]
           * f[CHWm + chw]
```

Loop invariant hoisted and strength reduced

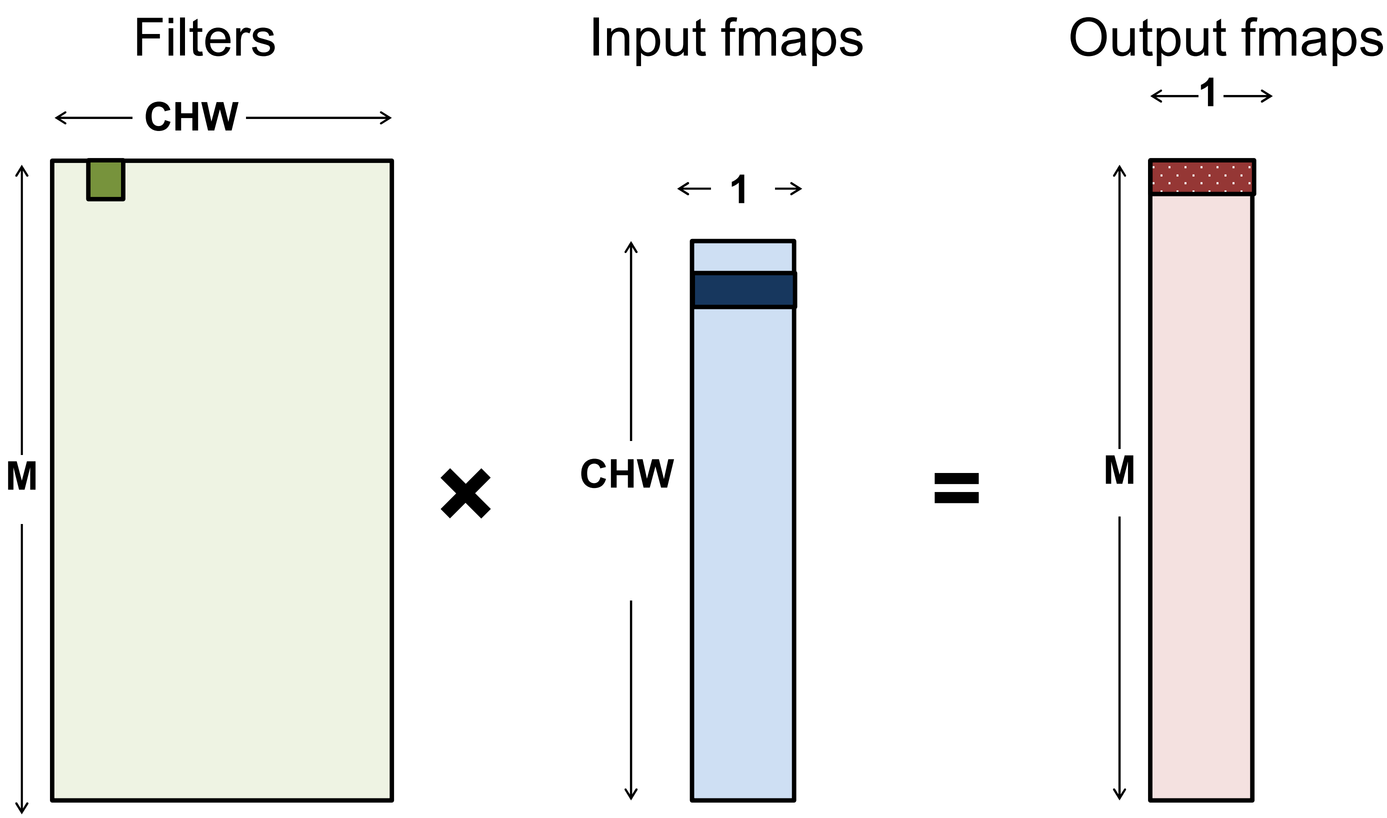
Offset to start of current output filter

Fully-Connected (FC) Layer



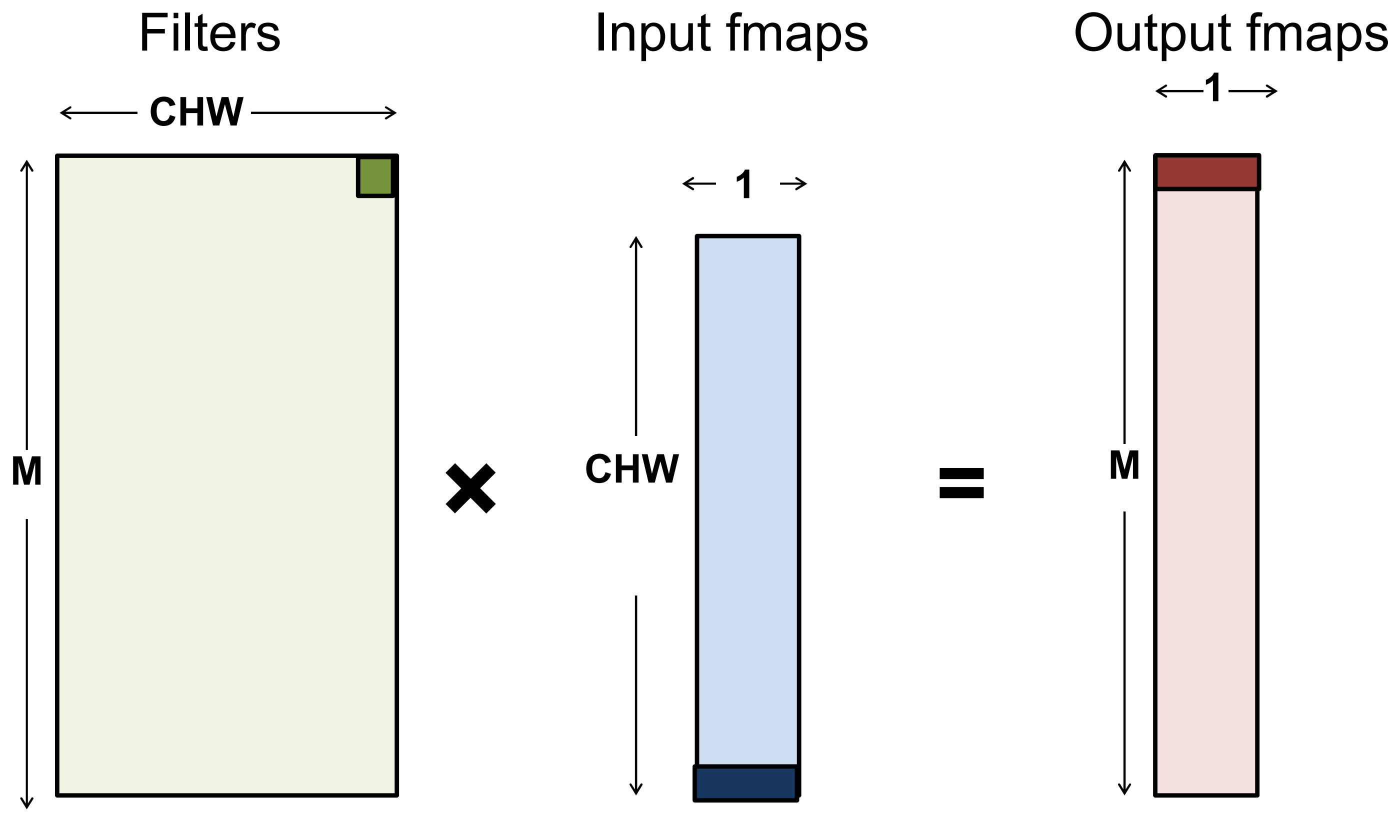
chw = 0

Fully-Connected (FC) Layer



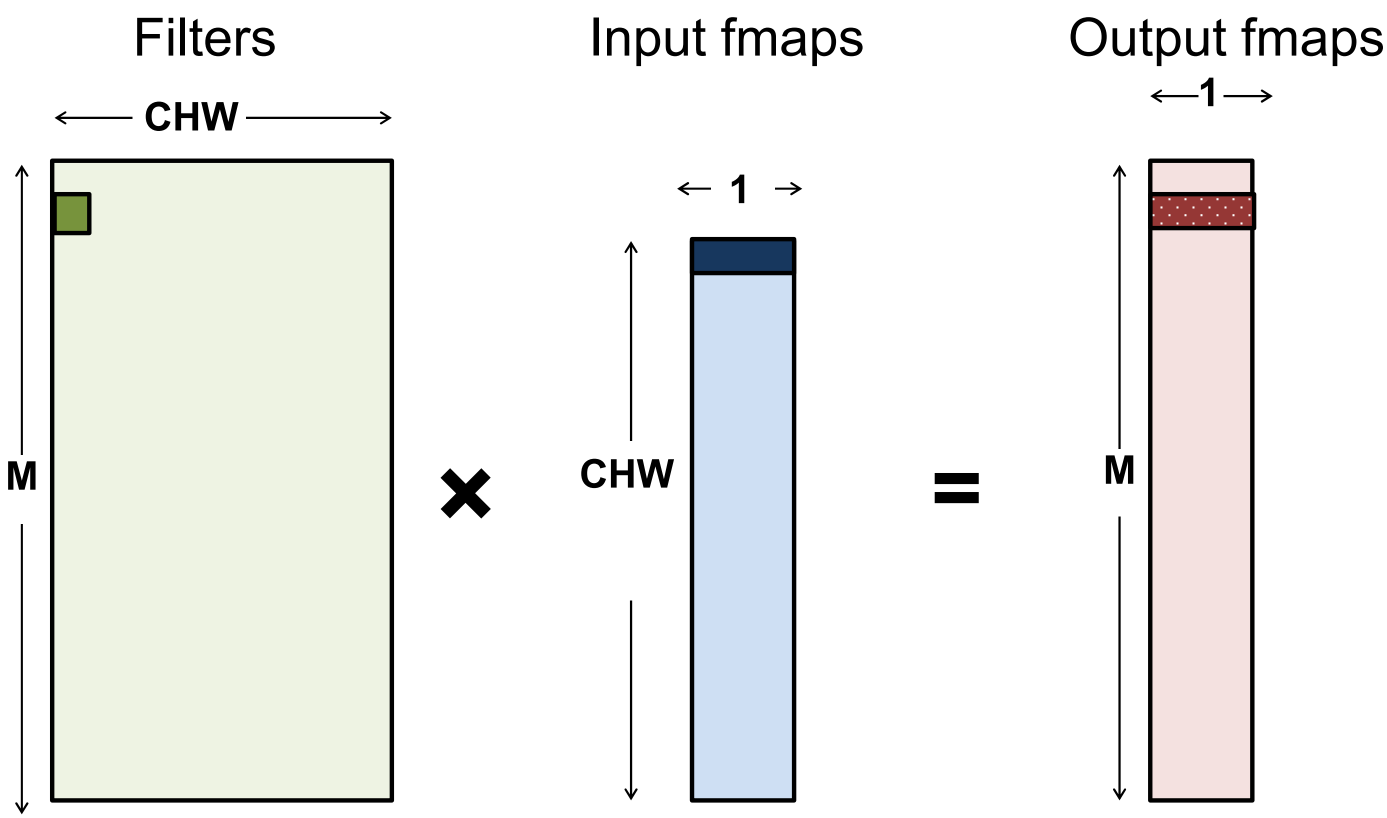
chw = 1

Fully-Connected (FC) Layer



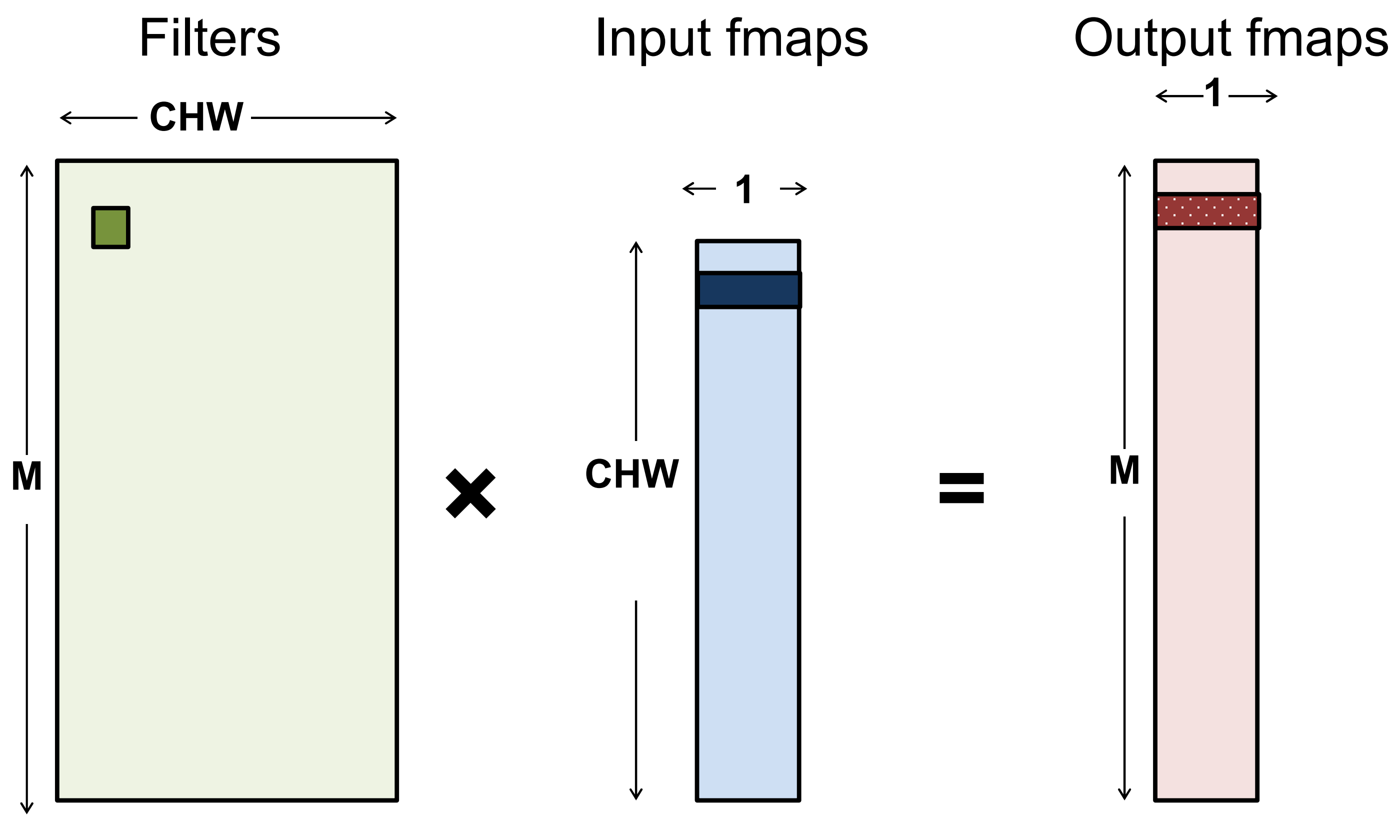
chw = C*H+W-1

Fully-Connected (FC) Layer



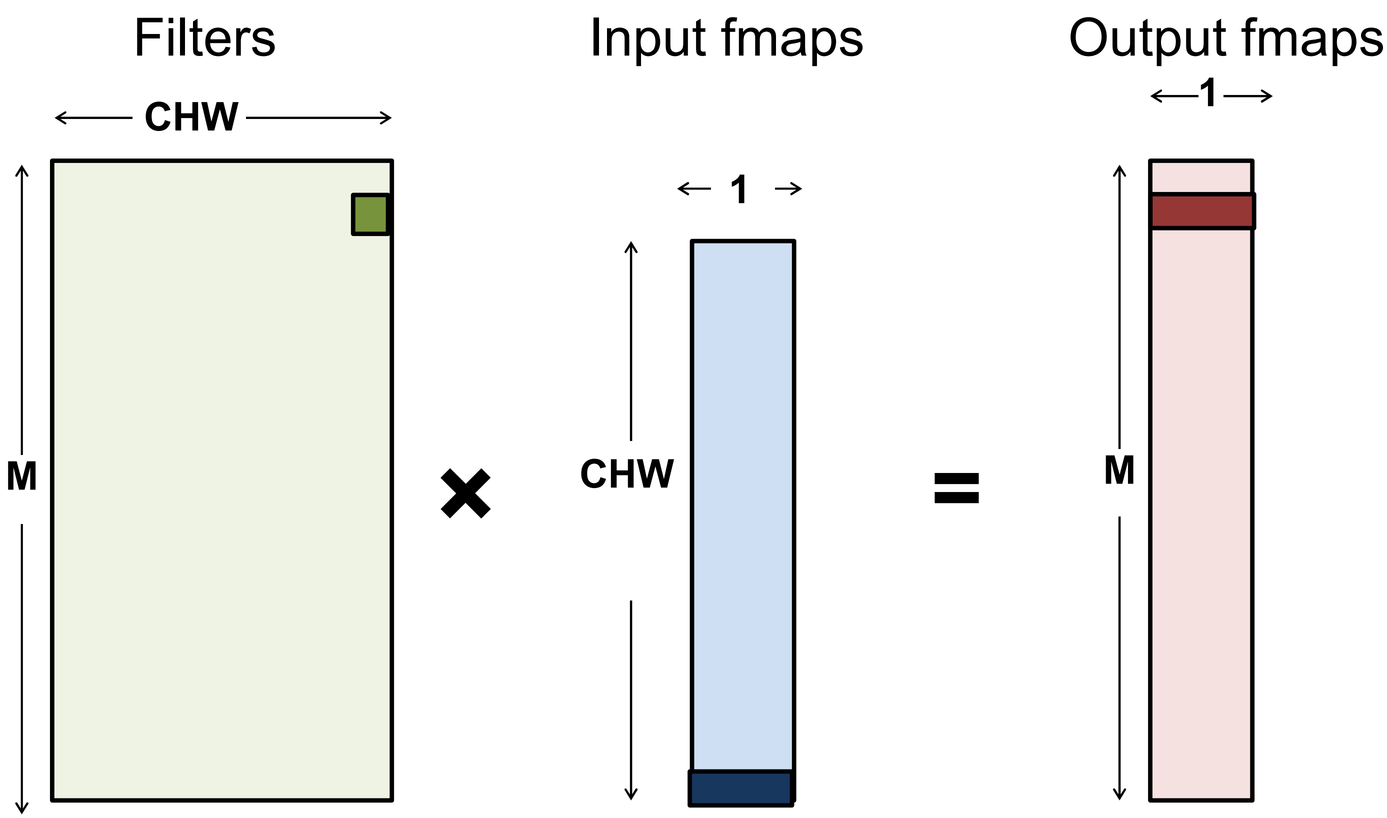
chw = 0

Fully-Connected (FC) Layer



chw = 1

Fully-Connected (FC) Layer



chw = C*H*W-1

Flattened FC Loops

```
int i[C*H*W];          # Input activations
int f[M*C*H*W];        # Filter Weights
int o[M];              # Output activations

CHWm = -C*H*W;
for m in [0, M):
  o[m] = 0;
  CHWm += C*H*W;
  for chw in [0, C*H*W)
     o[m] += i[chw]
           * f[CHWm + chw]
```

Most of the time is spent here!

Loop Iteration Overhead

```
       mv r1, 0                # r1 holds m
mloop: mul r3, r1, CHW         # r3 holds m*CHW (CHWm)
       mv r2, 0                # r2 holds chw
       mv r8, 0                # r8 holds psum (o[m])
xloop: ld r4, i(r2)            # r4 = i[chw]
       add r5, r2, r3          # r5 = CHWm + chw
       ld r6, f(r5)            # r6 = f[CHWm + chw]
       mac r8, r4, r6          # r8 += i[chw] * f[CHWm+chw]
       add r2, r2, 1           # r2 = chw + 1
       blt r2, C*W*H, xloop
       st r8, o(r1)
       add r1, 1
       blt r1, M, mloop
```

Index calculation (f)

Loop count and index calculation (i)

How many MACs/cycle (ignoring stalls)? ~ 1/6

Where is a major source of overhead?

FC scalar computation

Tensor: f_MCHW[['M', 'C', 'H'], W]

Rank: W

Rank: ['M', 'C', 'H']	0	1	2	3	4	5
(0, 0, 0)	9	3	7	4	1	8
(0, 0, 1)	8	8	5	8	5	5
(0, 1, 0)	1	1	7	2	9	7
(0, 1, 1)	4	9	4	5	1	5
(1, 0, 0)	8	5	3	2	5	2
(1, 0, 1)	1	3	9	9	3	4
(1, 1, 0)	5	2	5	2	5	9
(1, 1, 1)	9	6	4	7	5	5
(2, 0, 0)	2	4	4	2	3	2
(2, 0, 1)	2	8	2	2	7	2
(2, 1, 0)	9	2	6	3	2	1
(2, 1, 1)	1	3	8	8	2	1
(3, 0, 0)	6	6	5	1	5	6
(3, 0, 1)	8	5	2	4	3	5
(3, 1, 0)	6	6	9	6	1	3
(3, 1, 1)	8	5	5	6	7	6

Tensor: i_CHW[['C', 'H'], W]

Rank: W

Rank: ['C', 'H']	0	1	2	3	4	5
(0, 0)	4	9	7	7	3	9
(0, 1)	1	6	7	1	8	1
(1, 0)	6	5	7	9	1	5
(1, 1)	8	6	8	6	4	2

Tensor: unknown[M]

Rank: M

0	1	2	3
0	0	0	0

Loop Unrolling (2chw)

```
int i[C*H*W];         # Input activations
int f[M*C*H*W];       # Filter Weights
int o[M];             # Output activations

CHWm = -C*H*W
for m in [0, M):
  CHWm += C*H*W
  for chw in [0, C*H*W, 2):
     o[m] += (i[chw]
           *  f[CHWm + chw])
           + (i[chw + 1]
           *  f[CHWm + chw + 1])
```

Step by 2

Operands accessed in pairs

Loop overhead amortized over more computation

Index calculation amortized since

i[chw+1] => &(i+1)[chw]

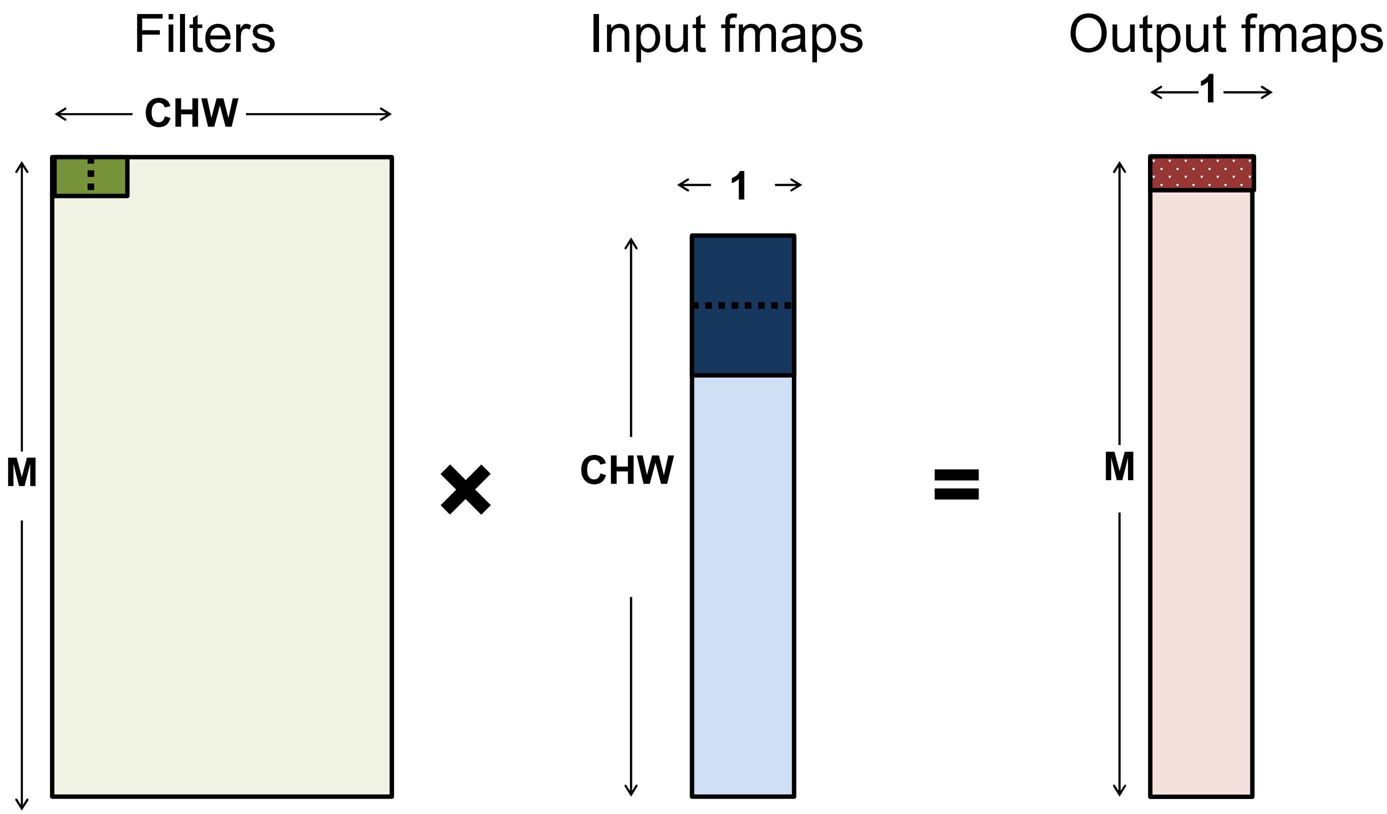
Fully Connected - Unrolled

```
       mv r1, 0               # r1 holds m
mloop: mul r3, r1, C*H*W      # r3 holds m*CHW
       mv r2, 0               # r2 holds chw
       mv r8, 0               # r8 holds psum (o[m])
xloop: ld r4, i(r2)           # r4 = i[chw]
       add r5, r2, r3         # r5 = CHMm + chw
       ld r6, f(r5)           # r6 = f[CHWm + chw]
       mac r8, r4, r6         # r8 += i[chw] * f[CHWm+chw]
       ld r7, i+1(r2)         # r… = i[chw + 1]
       ld r9, f+1(r5)         # r… …[CHWm + chw + 1]
       mac r8, r7, r9         # … …hw] * f[CHWm + chw +1]
       add r2, r2, 2          # r2 = …
       blt r2, C*W*H, xloop
       st r8, o(r1)
       add r1, r1, 1
       blt r1, M, mloop
```

Offset constant reflects constant index increment

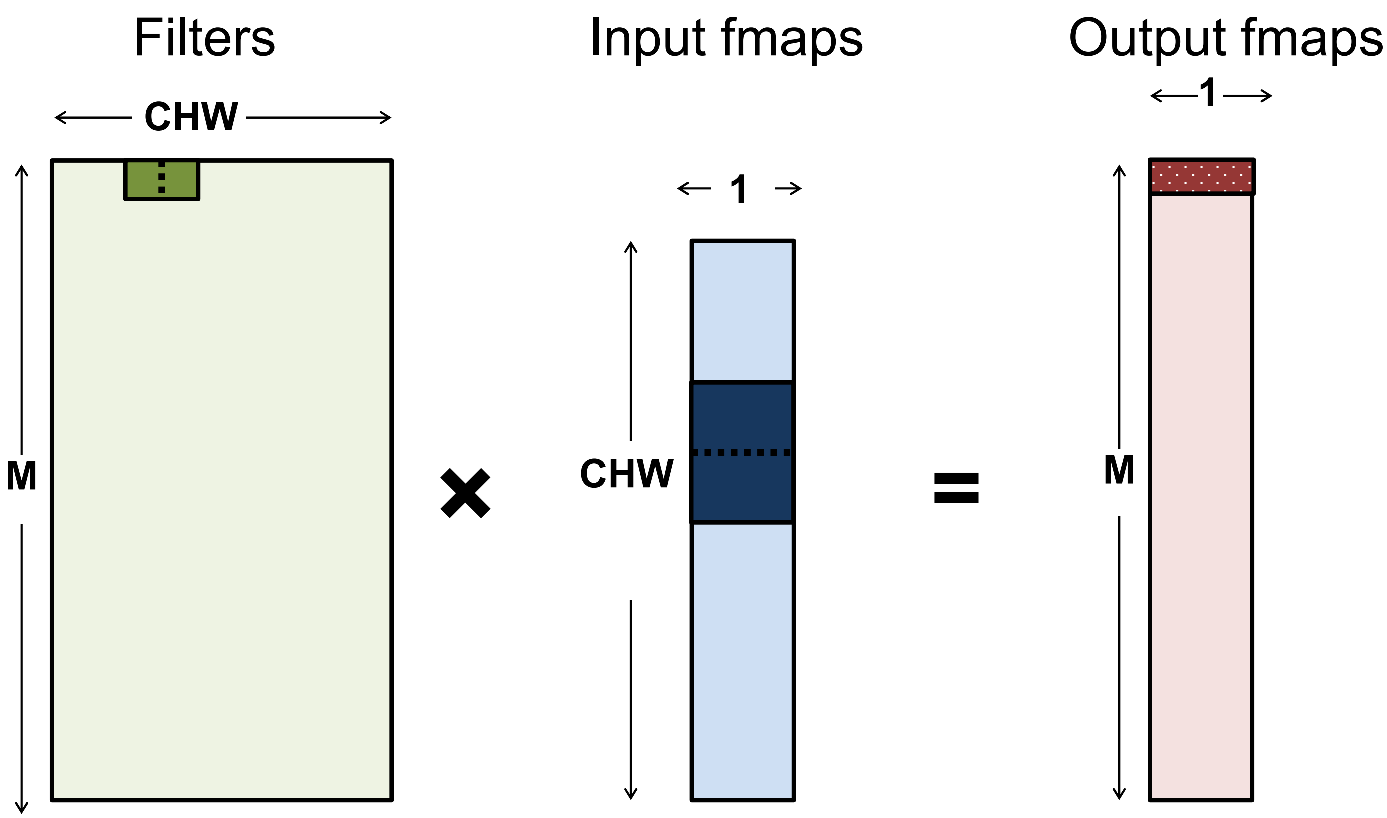
How many MACs/cycle (ignoring stalls)? ~ 2/9

Fully-Connected (FC) Layer



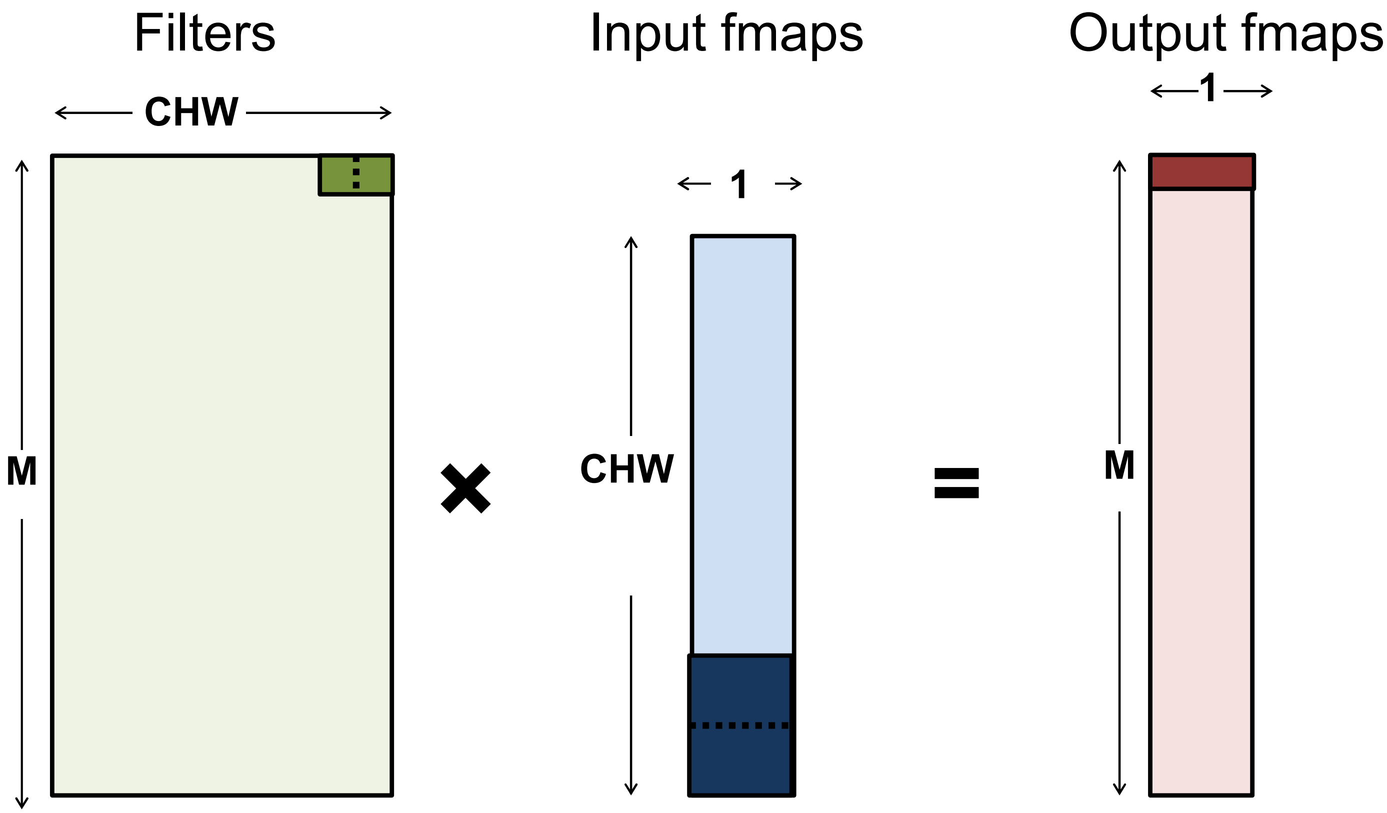
chw = 0,1

Fully-Connected (FC) Layer



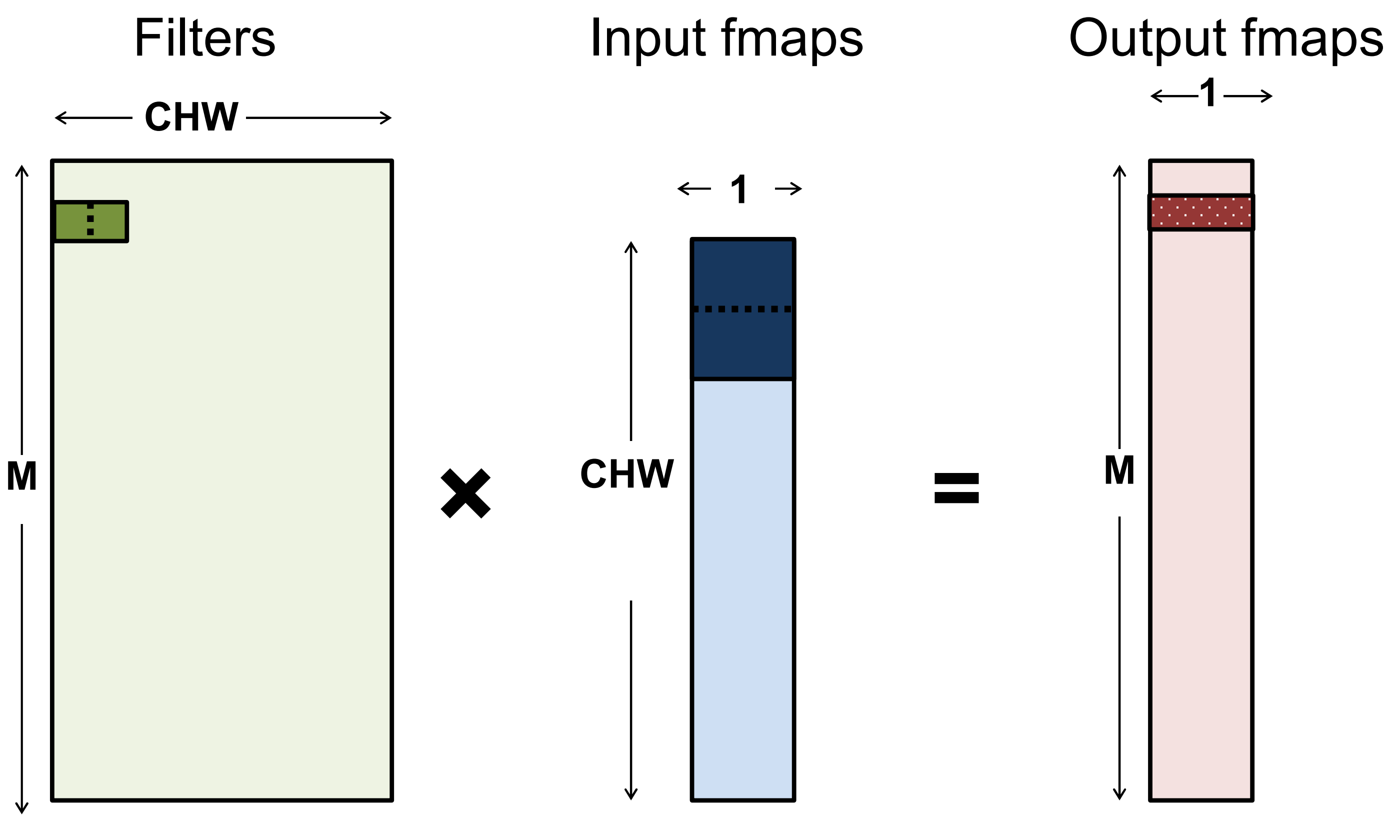
chw = 2,3

Fully-Connected (FC) Layer



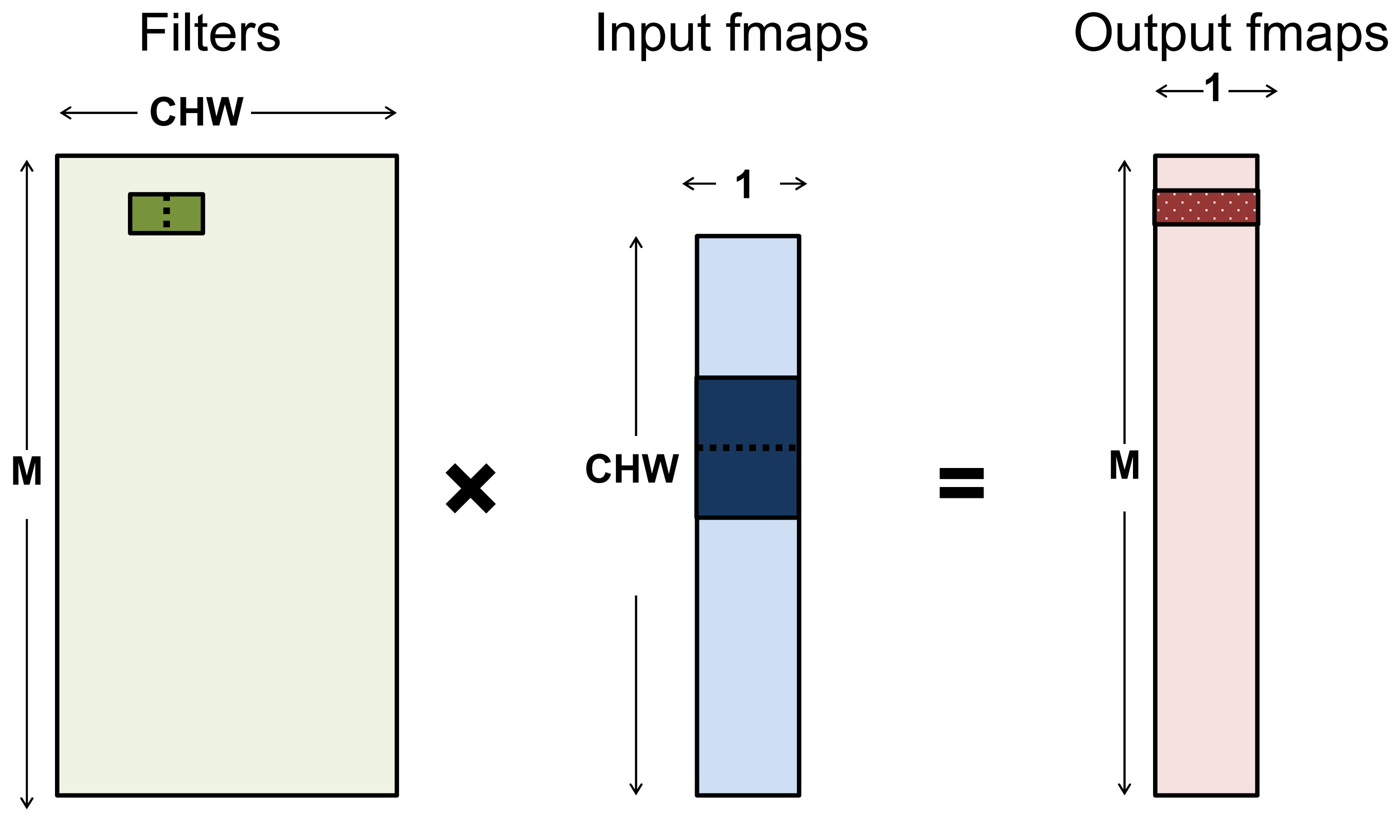
chw = C*H*W-2, C*H*W-1

Fully-Connected (FC) Layer



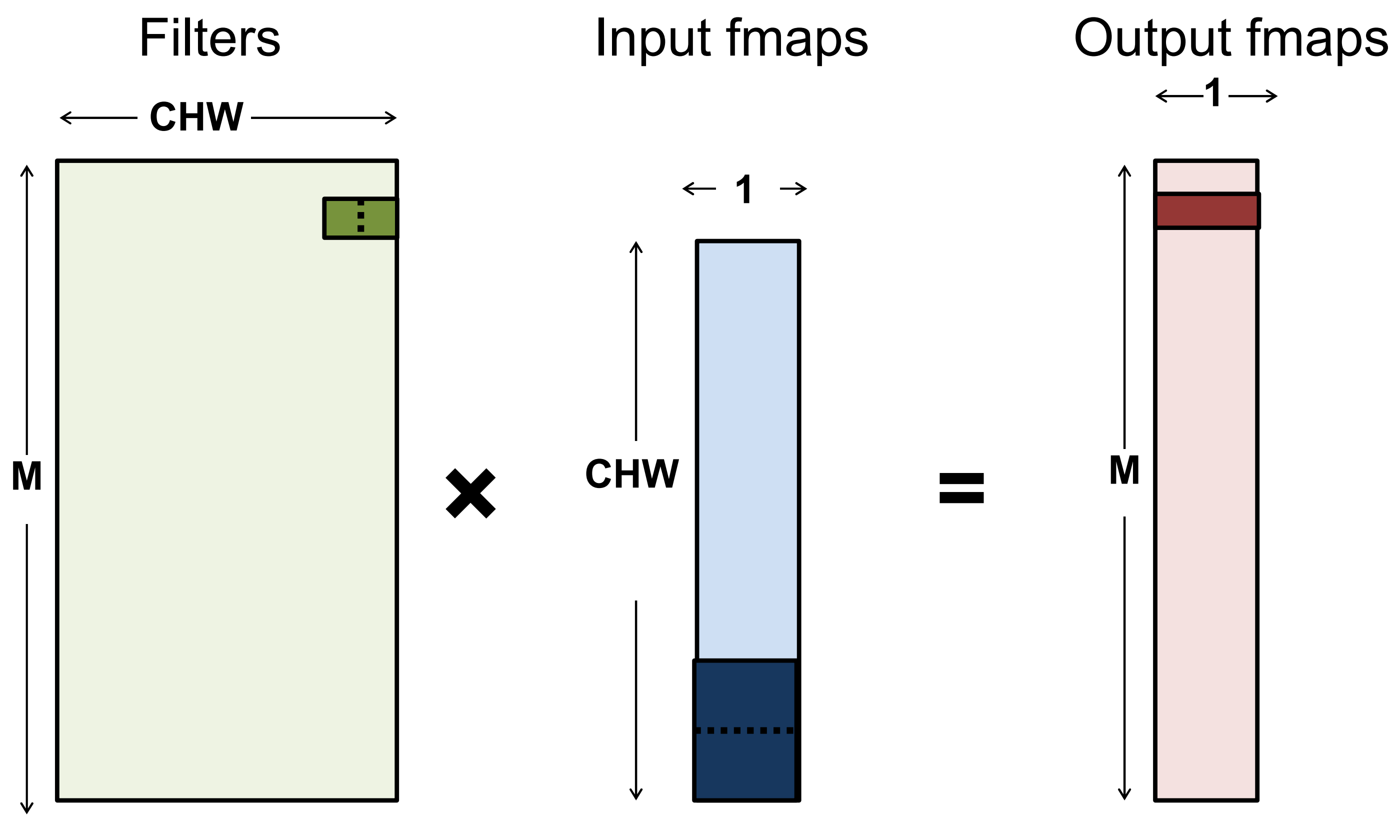
chw = 0,1

Fully-Connected (FC) Layer



chw = 2,3

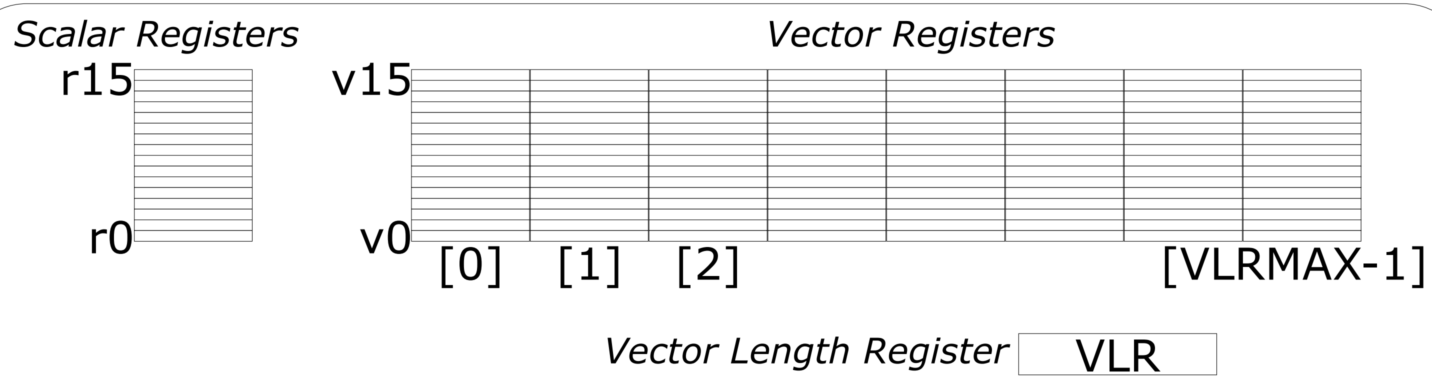
Fully-Connected (FC) Layer



chw = C*H*W-2, C*H*W-1

Can we incorporate this “pairing” into the architecture? Of course

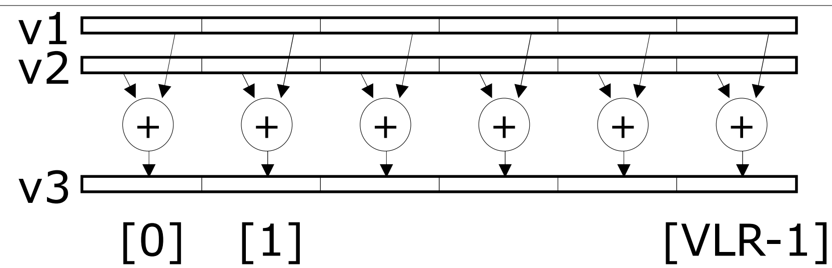
Vector Programming Model



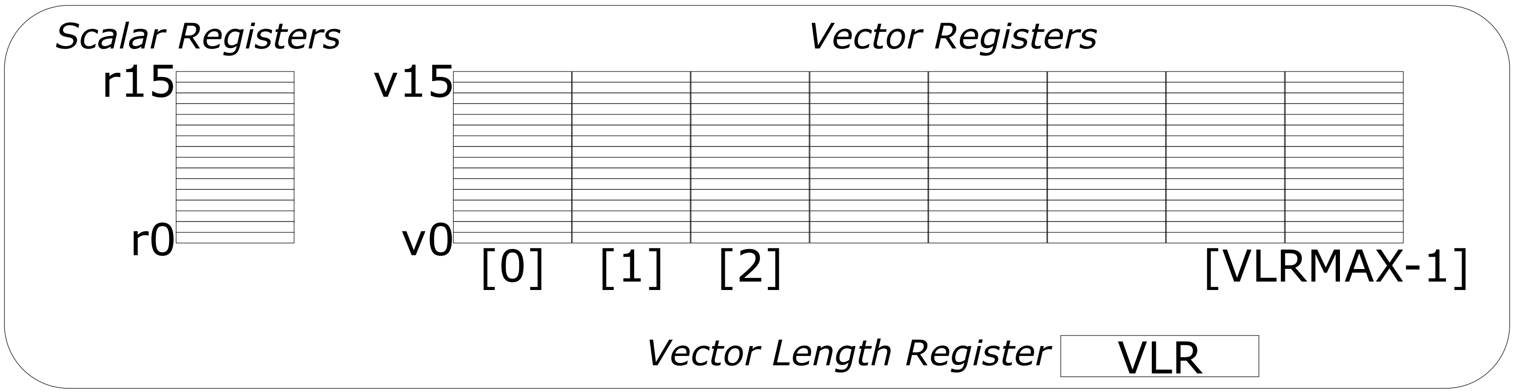
VLRMAX – number of elements in a vector register
VLR – number of elements to use in an instruction

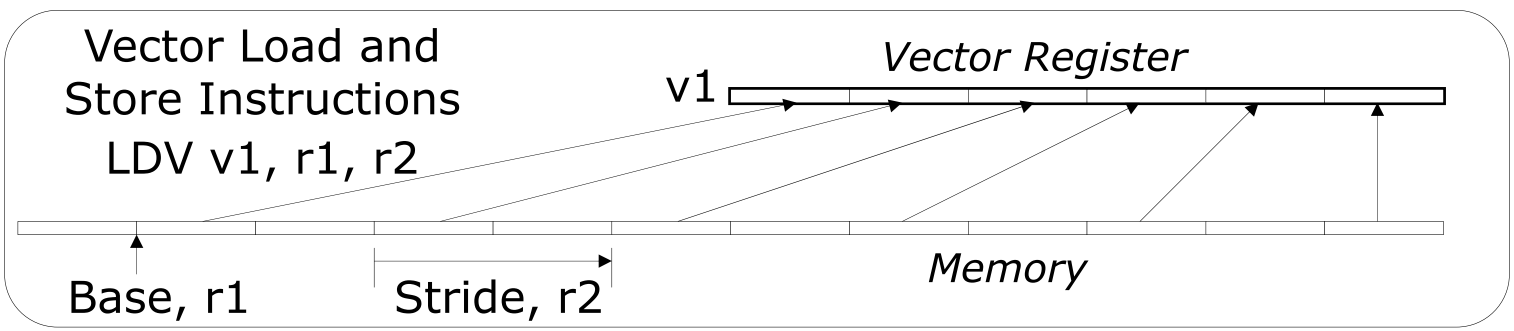
Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Programming Model

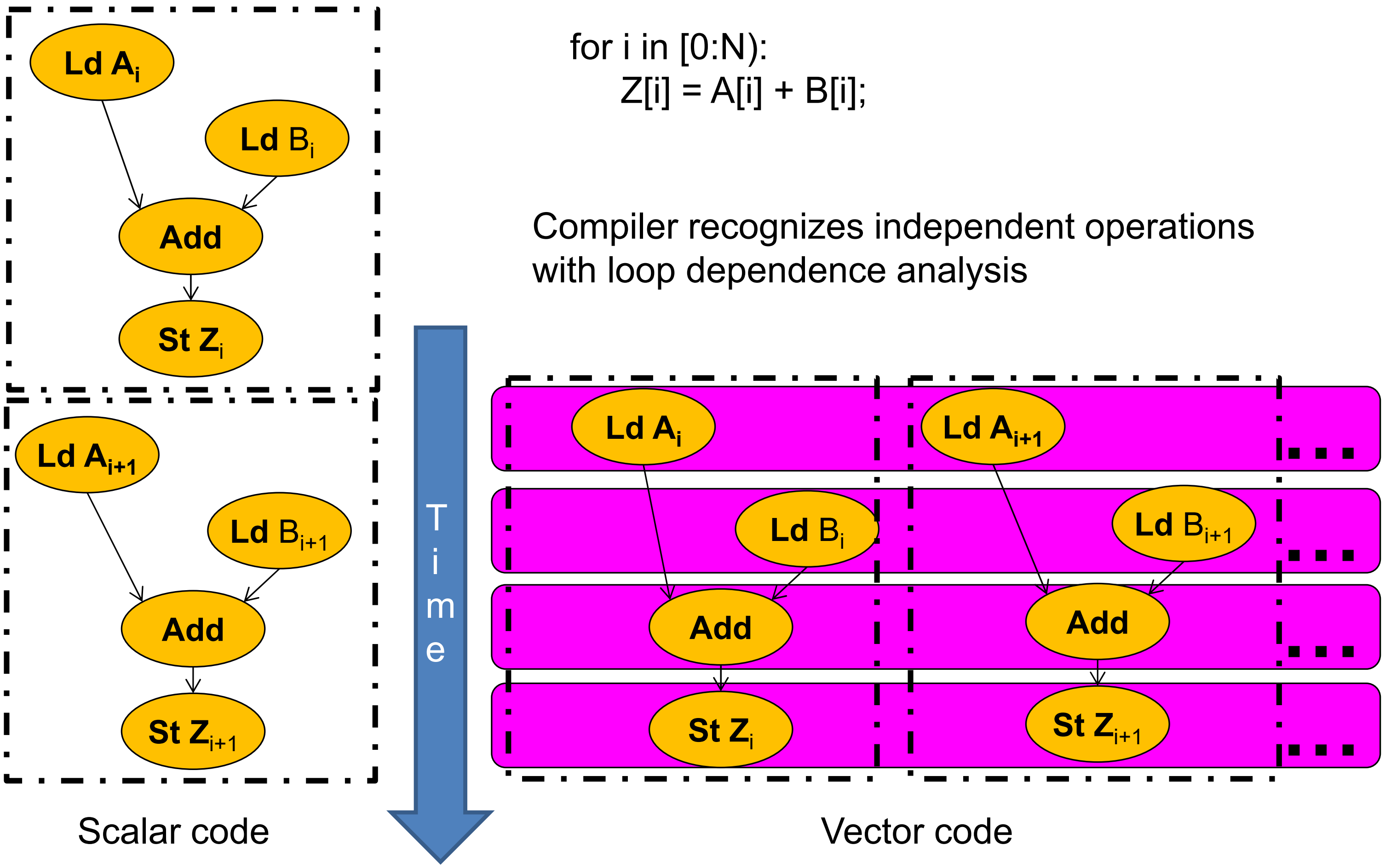




Compiler-based Vectorization

for i in [0:N):
    Z[i] = A[i] + B[i];

Compiler recognizes independent operations with loop dependence analysis



Loop Unrolled

```
int i[C*H*W];          # Input activations
int f[M*C*H*W];        # Filter Weights
int o[M];              # Output activations


for m in [0, M):
  CHWm = C*H*W*m
  for chw in [0, C*H*W, 2):
     o[m] += (i[chw]
           *  f[CHWm + chw])
           + (i[chw + 1]
           *  f[CHWm + chw + 1])

}
```

Sequential loads

Sequential loads

Reduction crosses elements

Will this vectorize? Not in the architecture presented so far

Parallel with animation

Tensor: f_MCHW[['M', 'C', 'H'], W]

Rank: W

Rank: ['M', 'C', 'H']

	0	1	2	3	4	5
(0, 0, 0)	9	3	7	4	1	8
(0, 0, 1)	8	8	5	8	5	5
(0, 1, 0)	1	1	7	2	9	7
(0, 1, 1)	4	9	4	5	1	5
(1, 0, 0)	8	5	3	2	5	2
(1, 0, 1)	1	3	9	9	3	4
(1, 1, 0)	5	2	5	2	5	9
(1, 1, 1)	9	6	4	7	5	5
(2, 0, 0)	2	4	4	2	3	2
(2, 0, 1)	2	8	2	2	7	2
(2, 1, 0)	9	2	6	3	2	1
(2, 1, 1)	1	3	8	8	2	1
(3, 0, 0)	6	6	5	1	5	6
(3, 0, 1)	8	5	2	4	3	5
(3, 1, 0)	6	6	9	6	1	3
(3, 1, 1)	8	5	5	6	7	6

Tensor: i_CHW[['C', 'H'], W]

Rank: W

Rank: ['C', 'H']

	0	1	2	3	4	5
(0, 0)	4	9	7	7	3	9
(0, 1)	1	6	7	1	8	1
(1, 0)	6	5	7	9	1	5
(1, 1)	8	6	8	6	4	2

Tensor: unknown[M]

Rank: M

0	1	2	3
0	0	0	0

Fully Connected – Loop Permutation

```
int i[C*H*W];         # Input activations
int f[M*C*H*W];       # Filter Weights
int o[M];             # Output activations

for m in [0, M):
  for chw in [0, C*H*W, 2):
    o[m] += i[chw] * f[CHW*m + chw]
    o[m] += i[chw + 1] * f[CHW*m + chw + 1]
```

No output is dependent on another output (other than commutative order)

```
for chw in [0, C*H*W, 2):
  for m in [0, M):
    o[m] += i[chw]     * f[CHW*m + chw]
    o[m] += i[chw + 1] * f[CHW*m + chw + 1]
```

FC – Permuted/Unrolled

```
// Loops permuted
for chw in [0, C*H*W):
  for m in [0, M):
    o[m] += i[chw] * f[CHW*m + chw]
```

```
// Unrolled inner loop
for chw in [0, C*H*W):
  for m in [0, M, 2):
    o[m]   += i[chw] * f[CHW*m + chw]
    o[m+1] += i[chw] * f[CHW*(m+1) + chw]
```

Unrolled calculation

Parallel m animation

Tensor: f_MCHW[['M', 'C', 'H'], W]

Rank: W

Rank: ['M', 'C', 'H']	0	1	2	3	4	5
(0, 0, 0)	9	3	7	4	1	8
(0, 0, 1)	8	8	5	8	5	5
(0, 1, 0)	1	1	7	2	9	7
(0, 1, 1)	4	9	4	5	1	5
(1, 0, 0)	8	5	3	2	5	2
(1, 0, 1)	1	3	9	9	3	4
(1, 1, 0)	5	2	5	2	5	9
(1, 1, 1)	9	6	4	7	5	5
(2, 0, 0)	2	4	4	2	3	2
(2, 0, 1)	2	8	2	2	7	2
(2, 1, 0)	9	2	6	3	2	1
(2, 1, 1)	1	3	8	8	2	1
(3, 0, 0)	6	6	5	1	5	6
(3, 0, 1)	8	5	2	4	3	5
(3, 1, 0)	6	6	9	6	1	3
(3, 1, 1)	8	5	5	6	7	6

Tensor: i_CHW[['C', 'H'], W]

Rank: W

Rank: ['C', 'H']	0	1	2	3	4	5
(0, 0)	4	9	7	7	3	9
(0, 1)	1	6	7	1	8	1
(1, 0)	6	5	7	9	1	5
(1, 1)	8	6	8	6	4	2

Tensor: unknown[M]

Rank: M

0	1	2	3
0	0	0	0

FC – Permuted/Unrolled/Hoisted

```
// Unrolled inner loop
for chw in [0, C*H*W):
  for m in [0, M, 2):
    o[m] += i[chw] * f[CHW*m + chw]
    o[m+1] += i[chw] * f[CHW*(m+1) + chw]
```

Same for all calculations

```
// Loop invariant hoisting of i[chw]
for chw in [0, C*H*W):
  i_chw = i[chw]
  for m in [0, M, 2):
    o[m] += i_chw * f[CHW*m + chw]
    o[m+1] += i_chw * f[CHW*(m+1) + chw]
```

Load hoisted out of loop

Fully Connection Computation

```
// Loop invariant hosting of i[chw]
for chw in [0, C*H*W):
  i_chw = i[chw];
  for m in [0, M, 2):
    o[m] += i_chw * f[CHW*m + chw]
    o[m+1] += i_chw * f[CHW*(m+1) + chw]
```

$I[C_0\ H_0\ W_0]$ $I[C_0\ H_0\ W_1]$ …
$I[C_0\ H_1\ W_0]$ $I[C_0\ H_1\ W_1]$ …
$I[C_0\ H_2\ W_0]$ $I[C_0\ H_2\ W_1]$ …
.
.
$I[C_1\ H_0\ W_0]$ $I[C_1\ H_0\ W_1]$ …
$I[C_1\ H_1\ W_0]$ $I[C_1\ H_1\ W_1]$ …
$I[C_1\ H_2\ W_0]$ $I[C_1\ H_2\ W_1]$ …
.
.
.

$F[M_0\ C_0\ H_0\ W_0]$ $F[M_0\ C_0\ H_0\ W_1]$ …
$F[M_0\ C_0\ H_1\ W_0]$ $F[M_0\ C_0\ H_1\ W_1]$ …
$F[M_0\ C_0\ H_2\ W_0]$ $F[M_0\ C_0\ H_2\ W_1]$ …
.
.
$F[M_0\ C_1\ H_0\ W_0]$ $F[M_0\ C_1\ H_0\ W_1]$ …
$F[M_0\ C_1\ H_1\ W_0]$ $F[M_0\ C_1\ H_1\ W_1]$ …
$F[M_0\ C_1\ H_2\ W_0]$ $F[M_0\ C_1\ H_2\ W_1]$ …
.
.
.
$F[M_1\ C_0\ H_0\ W_0]$ $F[M_1\ C_0\ H_0\ W_1]$ …
$F[M_1\ C_0\ H_1\ W_0]$ $F[M_1\ C_0\ H_1\ W_1]$ …
$F[M_1\ C_0\ H_2\ W_0]$ $F[M_1\ C_0\ H_2\ W_1]$ …
.
.
.

Weights needed together are far apart.
What can we do?

Access with stride or rearrange memory

FC – Layered Loops

```
// Unrolled inner loop
for chw in [0, C*H*W):
 i_chw = i[chw]
  for m in [0, M, 2):
    o[m] += i_chw * f[CHW*m + chw]
    o[m+1] += i_chw * f[CHW*(m+1) + chw]
```

```
// Level 2 loops
for chw in [0, C*H*W):
  i_chw = i[chw]
  for m1 in [0, M/VL):
// Level 1 loops
    parallel_for m0 in [0, VL):
      o[m1*VL+m0] += i_chw * f[CHW*(m1*VL+m0) + chw]
```

Limit of m1 (M/VL) times limit of m0 (VL) is M

Level 0 is a set of vector operations

m = m1*VL+m0

Einsum Rank Splitting

$$O_m = I_{chw} \times F_{m,chw}$$

$$O_m \rightarrow O_{m1 \times VL + m0} \rightarrow O_{m1,m0}$$

$$F_{m,chw} \rightarrow F_{m1 \times VL + m0,chw} \rightarrow F_{m1,m0,chw}$$

$$O_m = I_{chw} \times F_{m,chw}$$

$$\Downarrow$$

$$O_{m1,m0} = I_{chw} \times F_{m1,m0,chw}$$

FC – Layered Loops

```
// Level 2 loops
for chw in [0, C*H*W):
    for m1 in [0, M/VL):
    // Level 1 loops
    parallel_for m0 in [0, VL):
      o[m1][m0] += i[chw] * f[m1][m0][chw]
```

Flatten data structures

```
// Level 2 loops
for chw in [0, C*H*W):
    for m1 in [0, M/VL):
    // Level 1 loops
    parallel_for m0 in [0, VL):
      o[m1*VL+m0] += i[chw] * f[VL*CWH*m1+CWH*m0+chw]
```

FC – Layered Loops

```
// Level 2 loops
for chw in [0, C*H*W):
  i_chw = i[chw]
  for m1 in [0, M/VL):
// Level 1 loops
    parallel_for m0 in [0, VL):
      o[m1*VL+m0] += i_chw * f[VL*CWH*m1+CWH*m0+chw]
```

Hoist Loop Invariant!

Invariant in inner loop!

```
// Level 1 loops
    m1VL = m1*VL
    CHWVLm1_chw = CHW*VL*m1 + chw
    parallel_for m0 in [0, VL):
      o[m1VL+m0] += i_chw * f[CHWVLm1_chw + CHW*m0]
```

Stride!

Full Connected - Vectorized

```
        mv r1, 0               # r1 holds chw
        add r4, 0              # r4 holds CHWVLm1_chw
xloop:  ldv v1, i(r1), 0       # fill v1 with i[cwh]
        mv r2, 0               # r2 holds m1VL
mloop:  ldv v3, f(r4), CWH     # v3 holds f[]
        ldv v5, o(r2), 1       # v5 holds o[]
        macv v5, v1, v3        # multiply f[] * i[]
        stv v5, o(r2), 1       # store o
        add r2, r2, VL         # update m1VL
        add r4, r4, CHWVL      # update CHWVLm1_chw
        blt r2, M, mloop
        add r1, r1, 1          # update chw
        add r4, r4, r1         # update CHWVLm1_chw
        blt r1, CWH, xloop
```

Strength reduced

How many MACs/cycle (ignoring stalls)? ~ VL/7

Can we unroll this to get even more? Yes

FC – Layered Loops

```
// Level 2 loops
for chw in [0, C*H*W):
  for m1 in [0, M/VL):
// Level 1 loops
    parallel_for m0 in VL):
      o[m1*VL+m0] += i[chw] * f[VL*CWH*m1+CWH*m0+chw]
```

No constraints on loop permutations!

```
// Level 2 loops
for m1 in [0, M/VL):
  for chw in [0, C*H*W):
// Level 1 loops
    parallel_for m0 in [0, VL):
      o[m1*VL+m0] += i[chw] * f[VL*CWH*m1+CWH*m0+chw]
```

Loop order affects where loop invariants can be moved

Vector ISA Attributes

- **Compact**
  - **one short instruction encodes N operations**
  - **many implicit bookkeeping/control operations**
- **Expressive, hardware knows the N operations:**
  - **are independent**
  - **use the same functional unit**
  - **access disjoint registers**
  - **access registers in same pattern as previous instructions**
  - **access a contiguous block of memory (unit-stride load/store)**
  - **access memory in a known pattern (strided load/store)**

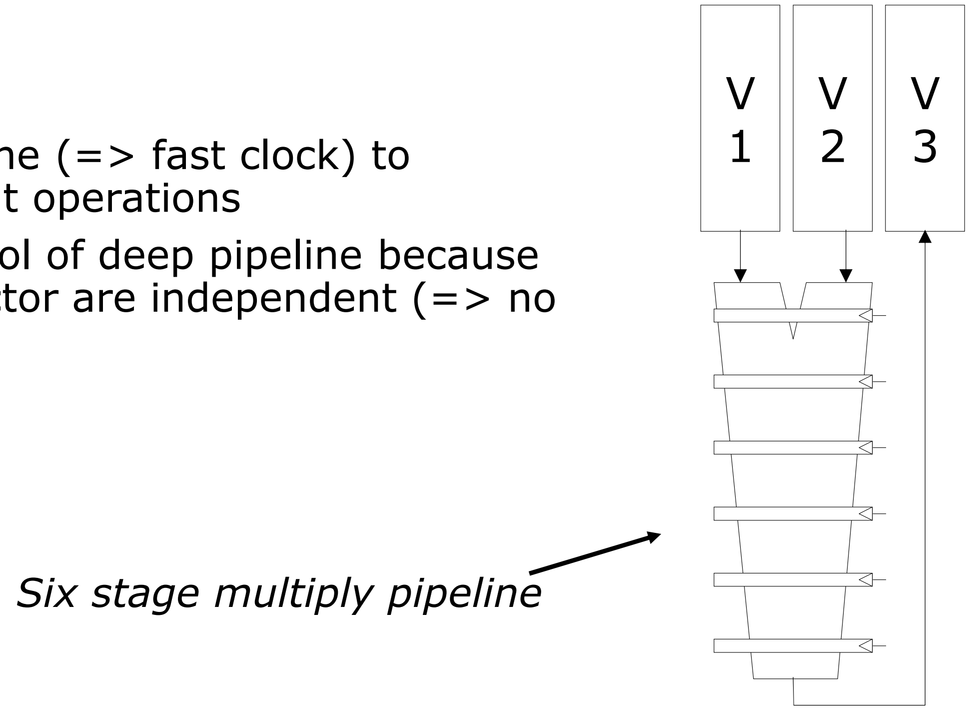
Vector instructions make “explicit” many things that are “implicit” with standard instructions

Vector ISA Hardware Implications

- **Large amount of work per instruction**
  - **-> Less instruction fetch bandwidth requirements**
  - **-> Allows simplified instruction fetch design**
- **Architecturally defined bookkeeping operations**
  - **-> Bookkeeping can run in parallel with main compute**
- **Disjoint vector element accesses**
  - **-> Banked rather than multi-ported register files**
- **No data dependence within a vector**
  - **-> Amenable to deeply pipelined/parallel designs**
- **Known regular memory access pattern**
  - **-> Allows for banked memory for higher bandwidth**

Vector Arithmetic Execution

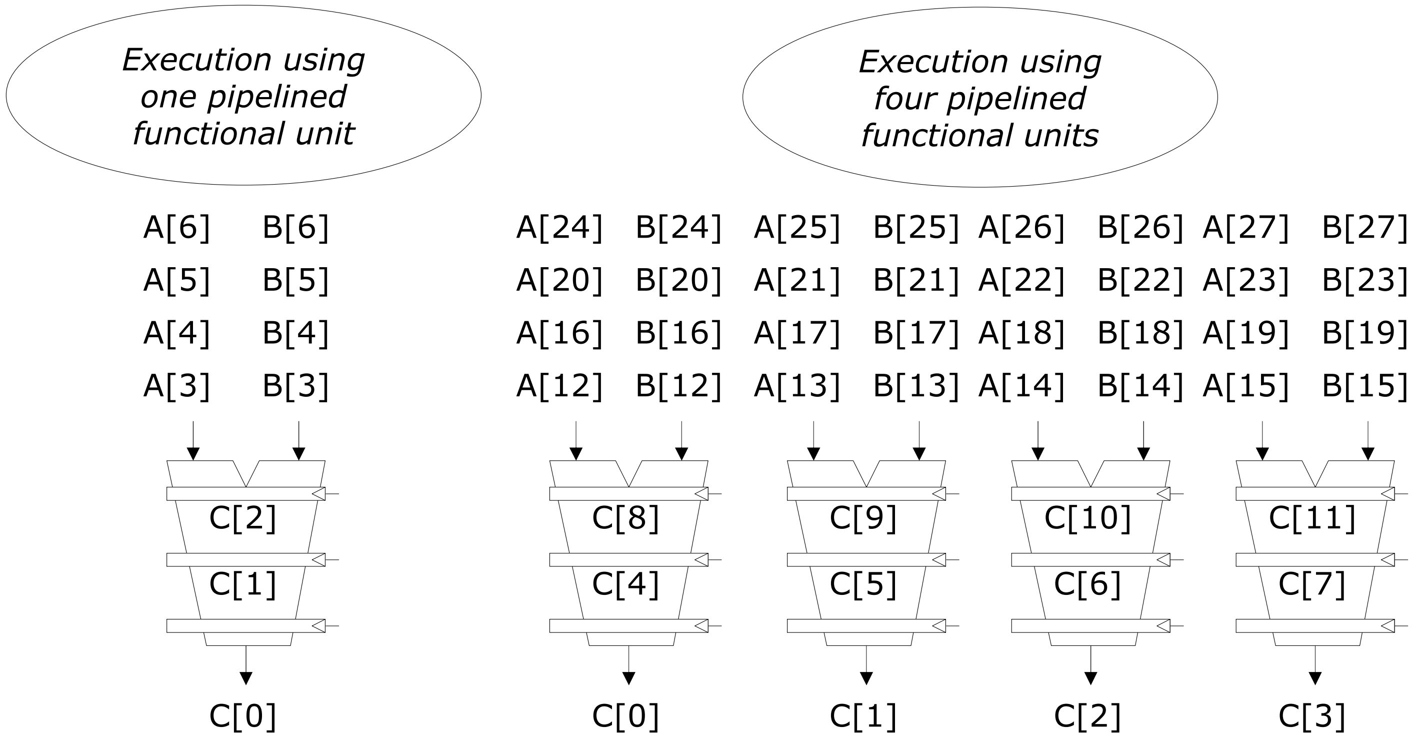
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



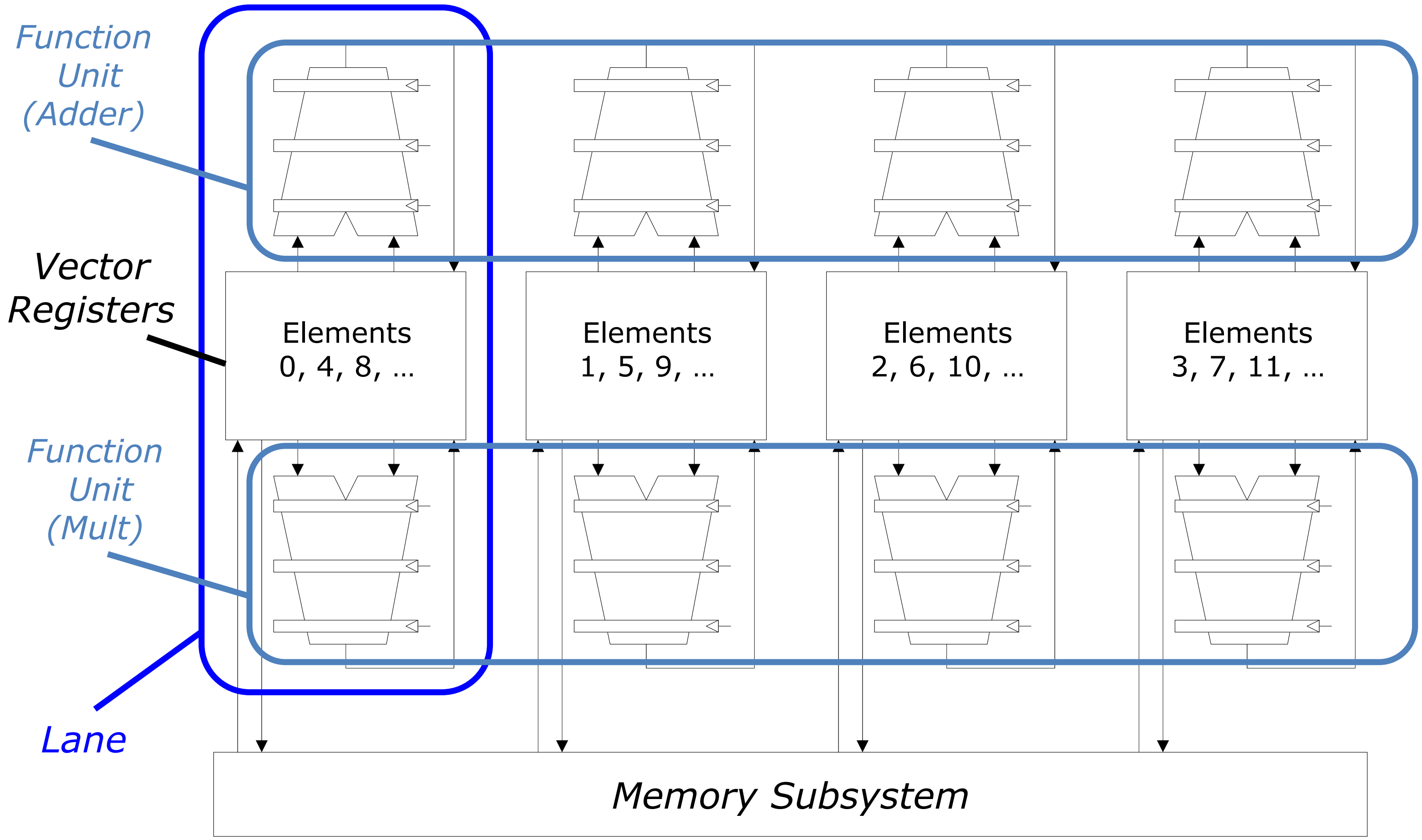
V3 <- V1 * V2

Vector Instruction Execution

ADDV C,A,B, where A, B, C are registers, e.g., V3, V1 and V2



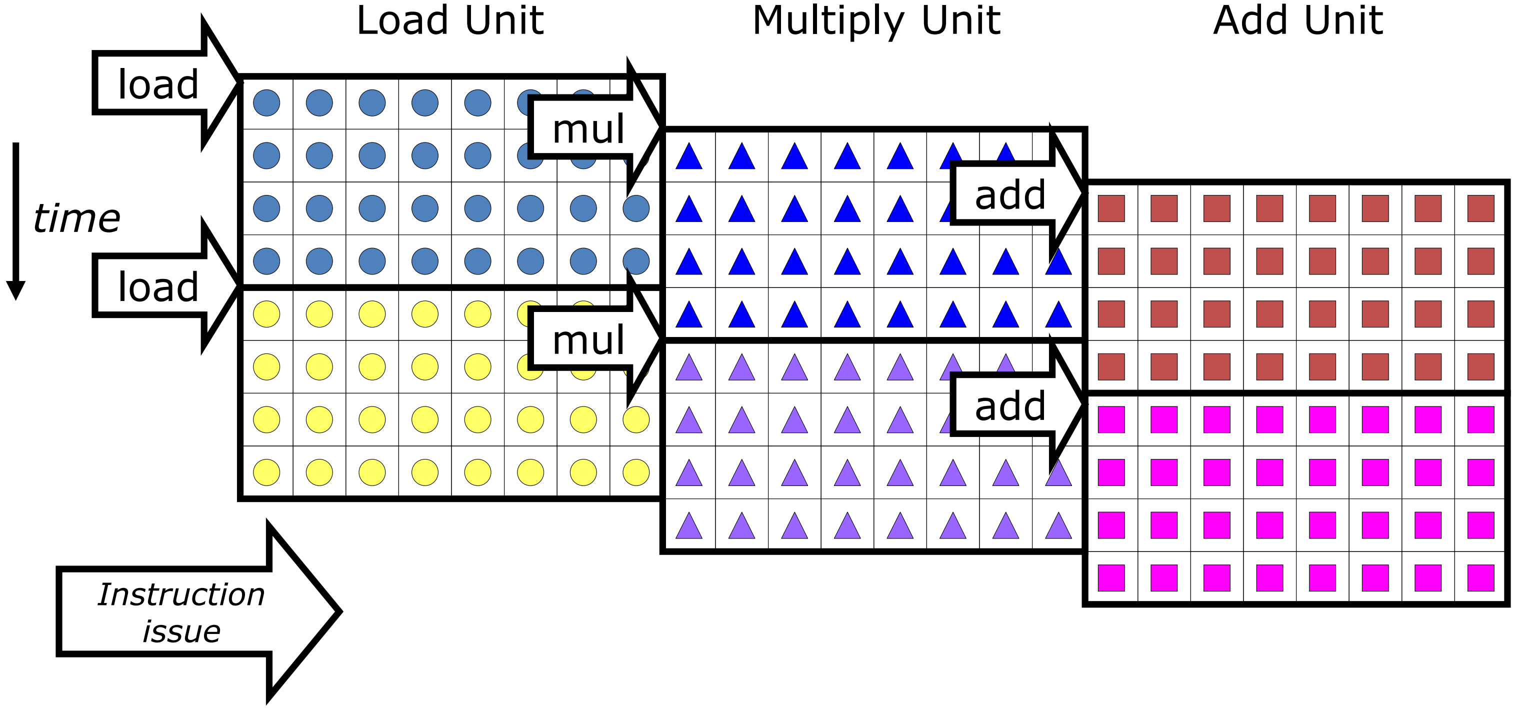
Vector Unit Structure



Vector Instruction Parallelism

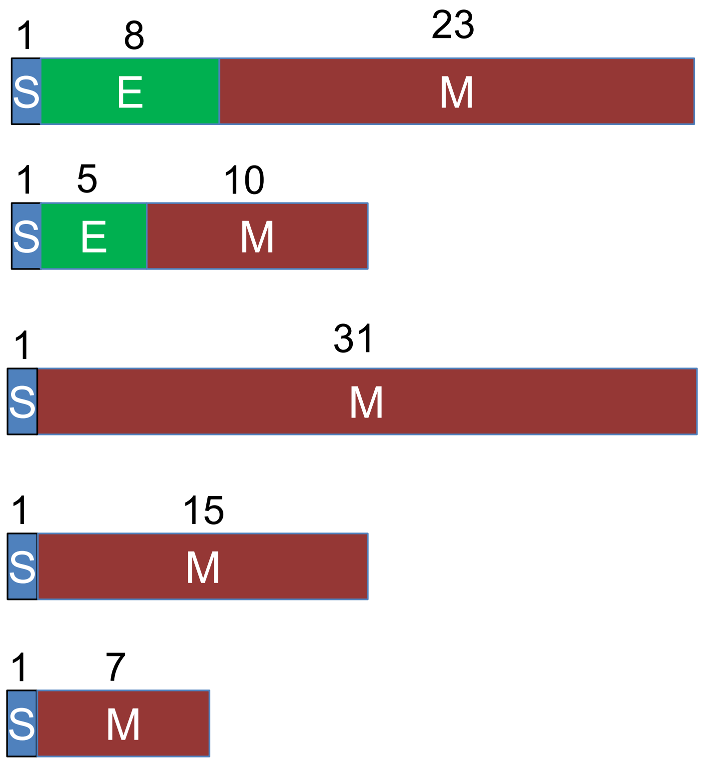
Can overlap execution of multiple vector instructions

- **example machine has 32 elements per vector register and 8 lanes**



Complete 24 operations/cycle while issuing 1 short instruction/cycle

ISA Datatypes



	Range	Accuracy
FP32	$10^{-38} – 10^{38}$	.000006%
FP16	$6x10^{-5}$ - $6x10^{4}$	.05%
Int32	0 – $2x10^{9}$	½
Int16	0 – $6x10^{4}$	½
Int8	0 – 127	½

Image Source: B. Dally

Intel – MMX/SSE/AVX

	Width	Int8	Int16	Int 32	Int64	FP16	FP32	FP64	Features
MMX	64	8	4	2	1				
SSE	128						4		
SSE2	128	16	8	4	2		4	2	
SSE3	128	16	8	4	2		4	2	R
AVX	256	32	16	8	4	16	8	4	
AVX2	256	32	16	8	4	16	8	4	GUMR
AVX3	512	64	32	16	8	?	16	8	GUMRP

G: gather
U: unaligned
M: MAC

R: reductions/permutations
P: Predicate masks

Source: Myriad non-authoritative sources on web

Python to C++ Chart

Version	*Implementation*	*Running time (s)*	*GFLOPS*	*Absolute speedup*	*Relative speedup*	*Fraction of peak*
1	Python	25,552.48	0.005	1	—	0.00%
2	Java	2,372.68	0.058	11	10.8	0.01%
3	C	542.67	0.253	47	4.4	0.03%
4	Parallel loops	69.80	1.969	366	7.8	0.24%
5	Parallel divide-and-conquer	3.80	36.180	6,727	18.4	4.33%
6	+ vectorization	1.10	124.914	23,224	3.5	14.96%
7	+ AVX intrinsics	0.41	337.812	62,806	2.7	40.45%
8	Strassen	0.38	361.177	67,150	1.1	43.24%

[**Leiserson**, There's plenty of room at the top, *Science*, 2020]

Next Lecture: Roofline Analysis and Transforms

Thank you!