Accelerators-II

Tushar Krishna
Associate Professor @ Georgia Tech
Visiting Professor @ MIT EECS and CSAIL

Slide Acknowledgments: *Michael Pellauer, Angshuman Parashar, Joel Emer, Vivienne Sze, Hyoukjun Kwon, Felix Kao*

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows

Recap

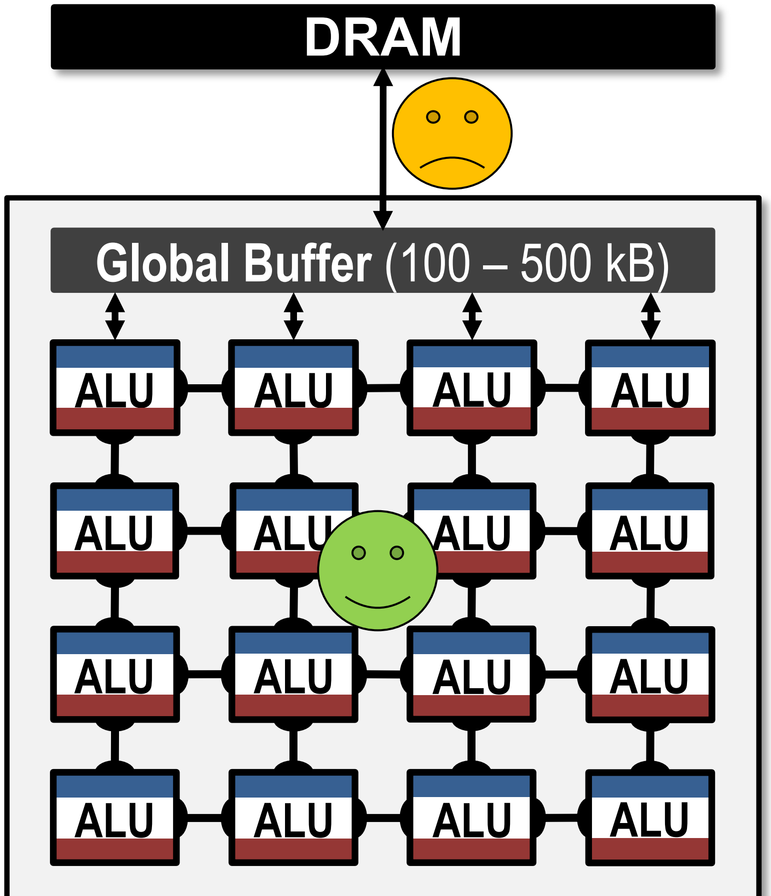
- Why domain-specific accelerators?
  - High Throughput requirements (workload constraint)
  - Energy costs of Data Movement (technology constraint)

- Why do accelerators help?
  - custom datapaths for the operator(s) of interest (e.g., matrix multiplication)
  - remove control overheads that Turing-complete engines (e.g., CPUs) have such as instruction fetch/decode, speculation, caches, ..

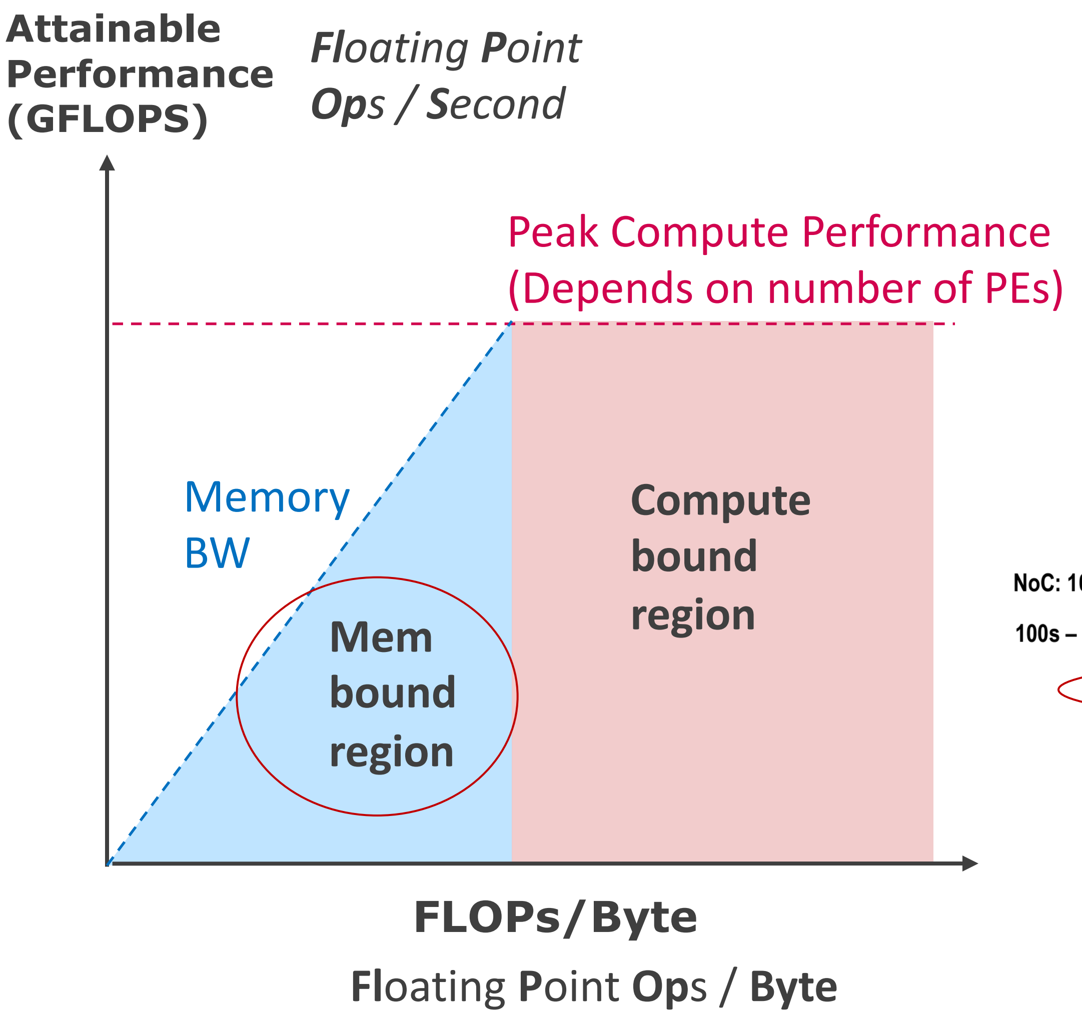
Accelerators

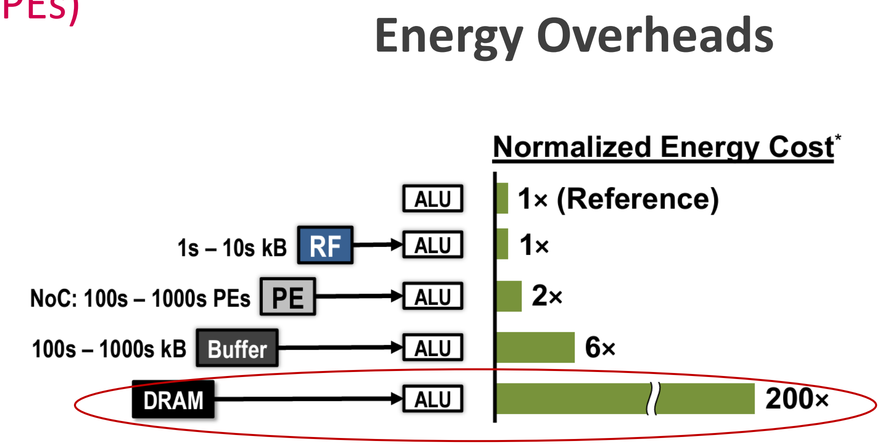
Off-Chip Memory

Custom Datapath



Why does this matter?





How to reduce BW requirement?

VGG16 conv 3_2	
Multiply Add Ops	1.85 Billion
Weights	590 K
Inputs	803 K
Outputs	803 K

Data Reuse

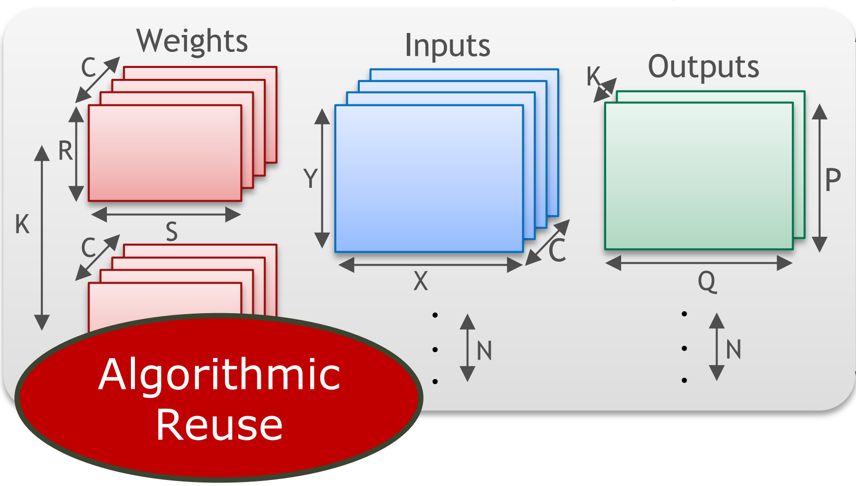
How to exploit reuse? “Dataflow”

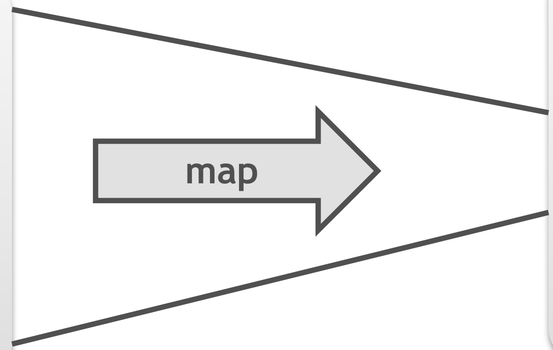
i.e., fine-grained schedule of computations within DNN accelerators

- Computation Order
- Parallelization Strategy

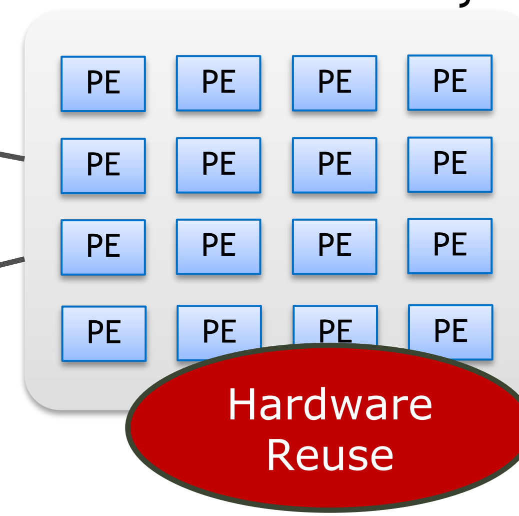
Dataflow Implication: Algorithm Reuse → HW Reuse

7-dimensional network layer





2D hardware array

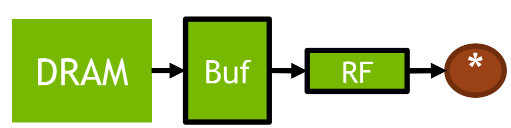


- **7D Computation Space**
  - R * S * X * Y * C * K * N
- **4D Operand/Result Data Spaces -**
  - Weights - R * S * C * K
  - Inputs - X * Y * C * N
  - Outputs - P * Q * K * N

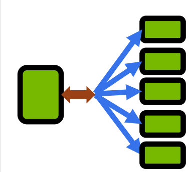
- **HW Design-space**
  - Number of PEs
  - Memory Hierarchy (Sizes and Bandwidths)
  - Interconnect Bandwidth

- **HW Reuse Structures**

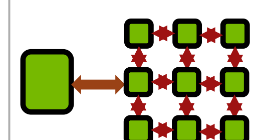
Temporal



Spatial (Multicast)



Spatial (Forwarding)

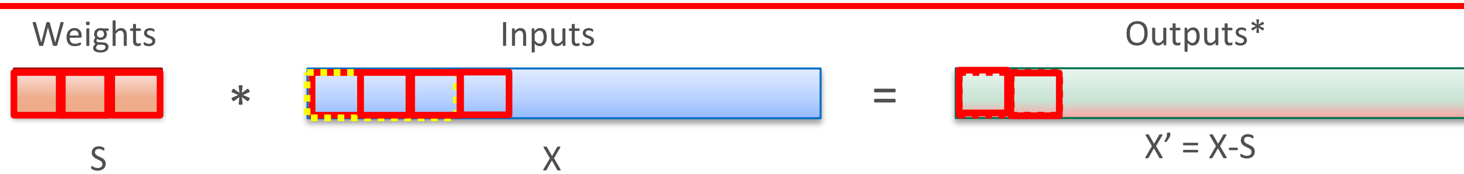


How to describe and navigate?

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows

Output Stationary (OS) Dataflow



Computation

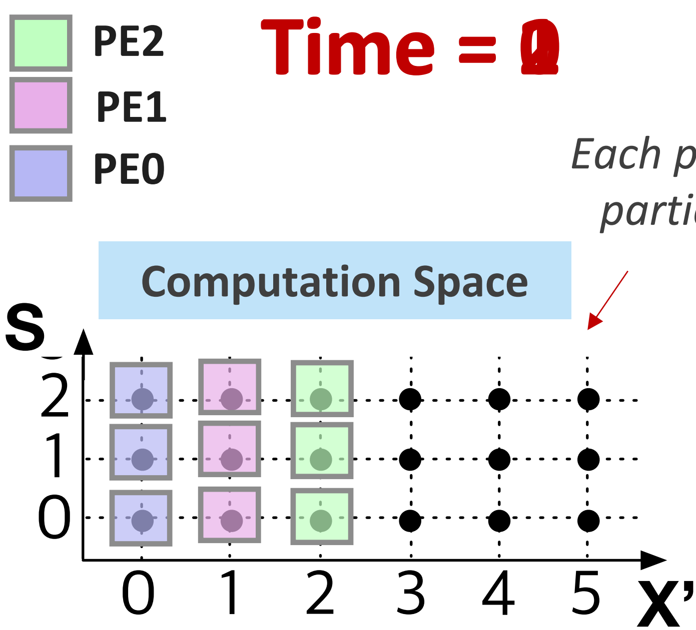
```
for(int x = 0; x < X’; x++)
  for(int s = 0; s < S; s++)
Output[x] += Weight[s] * Input[x+s]
```

Data

PartialSum[X’][S] needs to access:

- Weight[s]
- Output[x’]
- Input[x’+s]

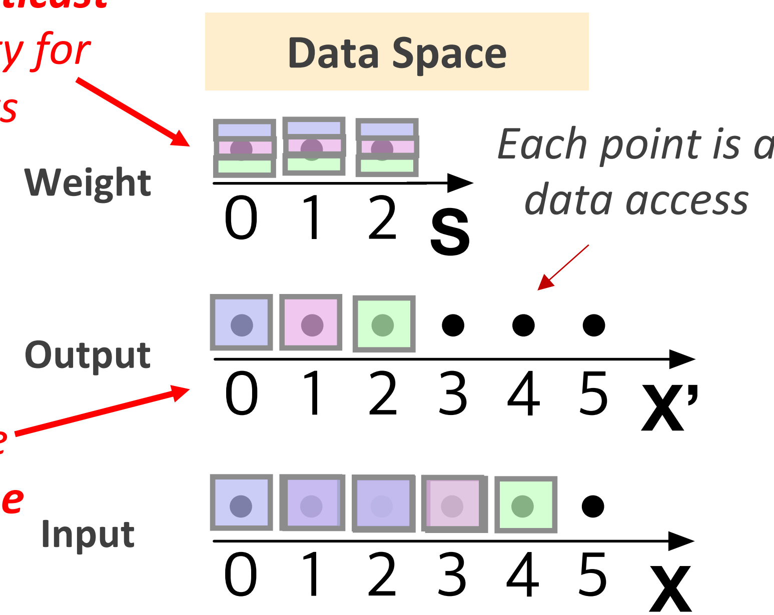
Time = 0



Each point is a partial sum

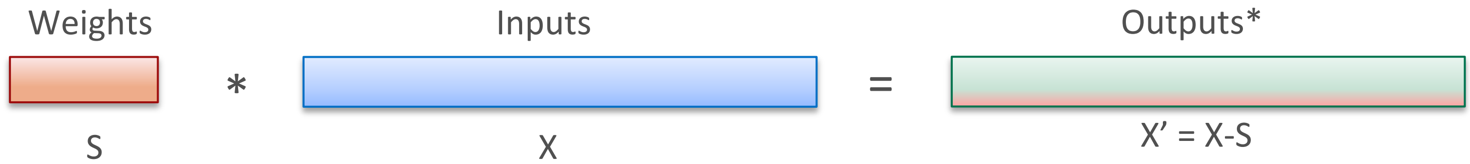
Spatial multicast *opportunity for weights*

Output does not change over time
=> ***Temporal reuse*** *opportunity*



Each point is a data access

Describing OS dataflow



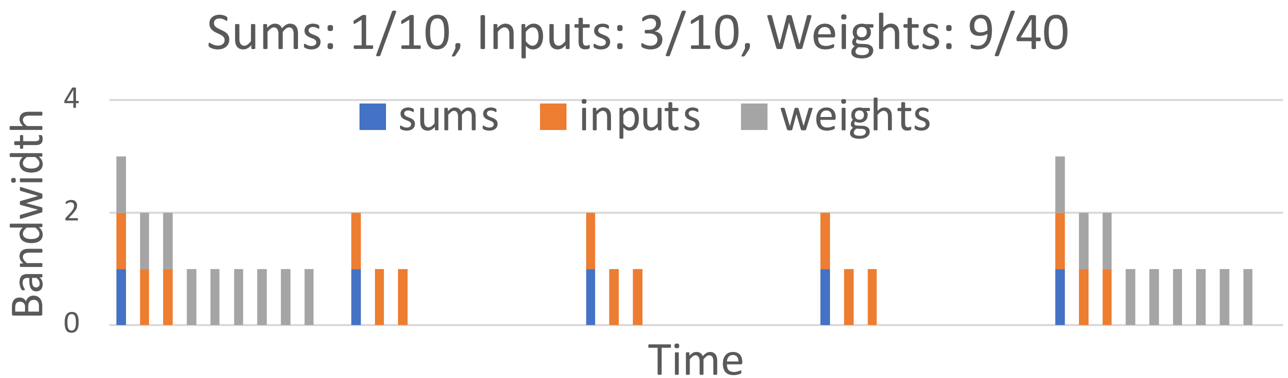

```
int i[X];       # Input activations
int w[S];       # Filter weights
int o[X’];      # Output activations

for (x = 0; x < X’; x++) {
       for (s = 0; s < S; s++) {
          o[x] += i[x+s]*w[s];
}
```

How often does the datapath change the weight and input? — Every cycle

Output? — Every S cycles: “Output stationary”

What do we mean by “stationary”?

The datatype (and dimension) that changes most slowly



- Imprecise analogy: think of data transfers as a wave with “amplitude” and “period”
  - The stationary datatype has the **longest** period (locally held tile changes most slowly)
    - Note: like waves, may have harmful “interference” (bursts)
  - intermediate staging buffers reduce both bandwidth and energy
- Often corresponds to datatype that is “done with” earliest without further reloads
- ***Note:*** *the “stationary” name is meant to give intuition, not to be a complete specification of all the behavior of a dataflow*

“Done with” vs “Needs Reload”

```
int i[X];       # Input activations
int w[S];       # Filter weights
int o[X’];      # Output activations

for (x = 0; x < X’; x++) {
       for (s = 0; s < S; s++) {
          o[x] += i[x+s]*w[s];
}
```

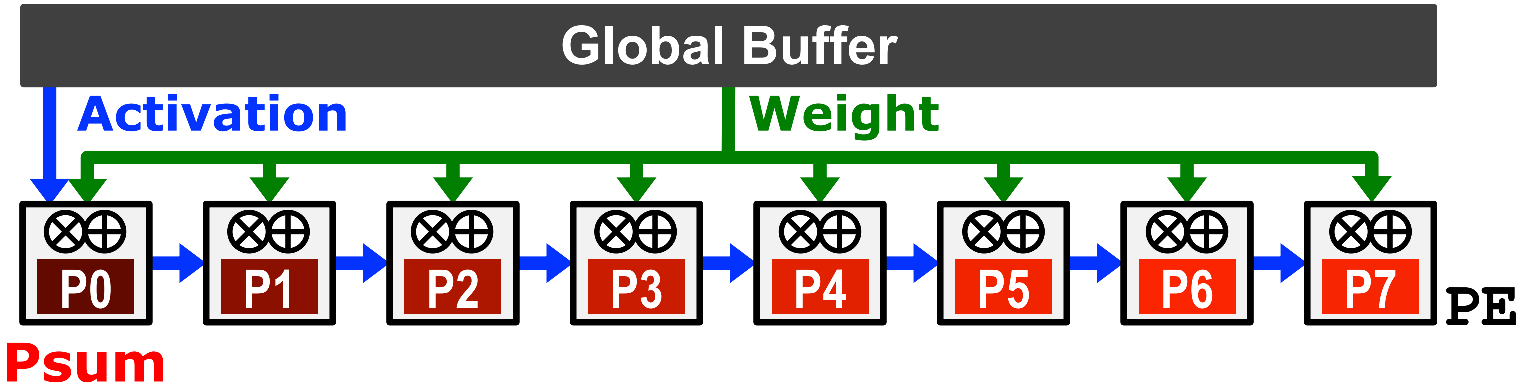
How many times will x == 2?

How many times will x+s == 2?

How many times will s == 2?

- Temporal distance between re-occurrence dictates buffer size to avoid re-load
- How do you know if a buffer that size is worth it?

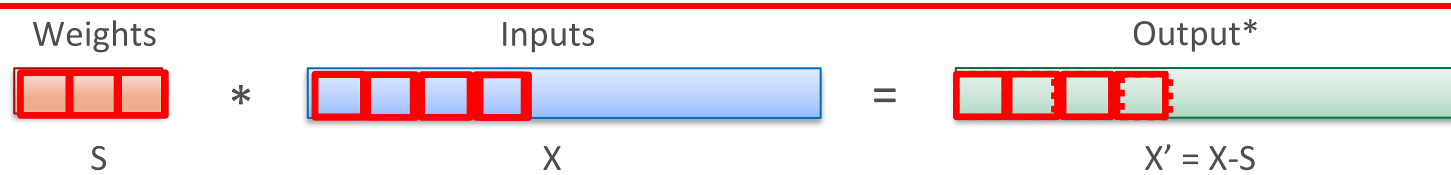
OS Dataflow Implementation



- **Minimize partial sum** R/W energy consumption
  - maximize local accumulation

- **Broadcast/Multicast filter weights** and **reuse activations spatially** across the PE array

Weight Stationary (WS) Dataflow



Computation

```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
Output[x] += Weight[s] * Input[x+s]
```

Data

PartialSum[X'][S] needs to access:

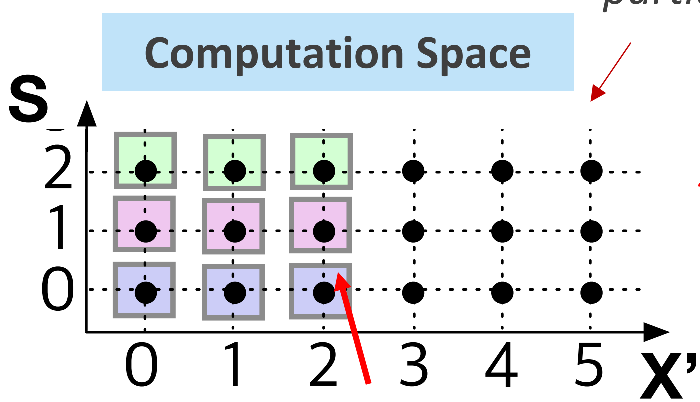
- Weight[s]
- Output[x']
- Input[x'+s]

PE2

PE1

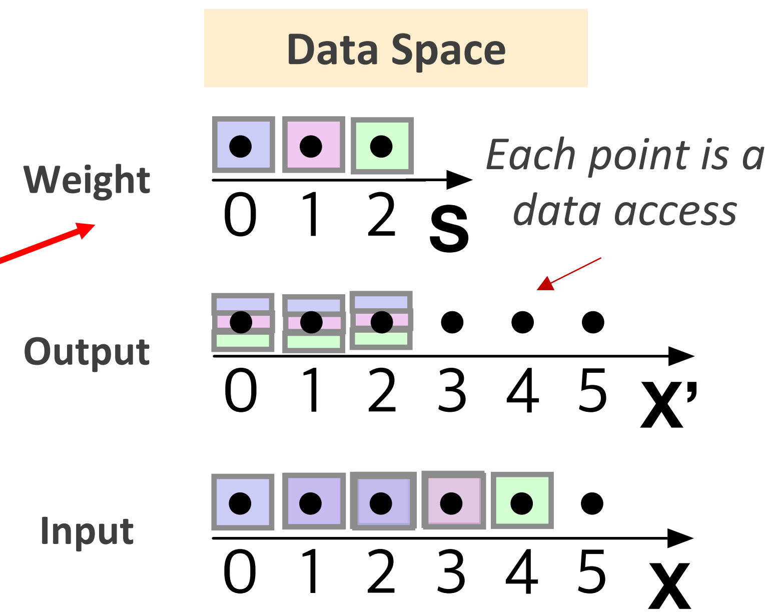
PE0

Time = 0

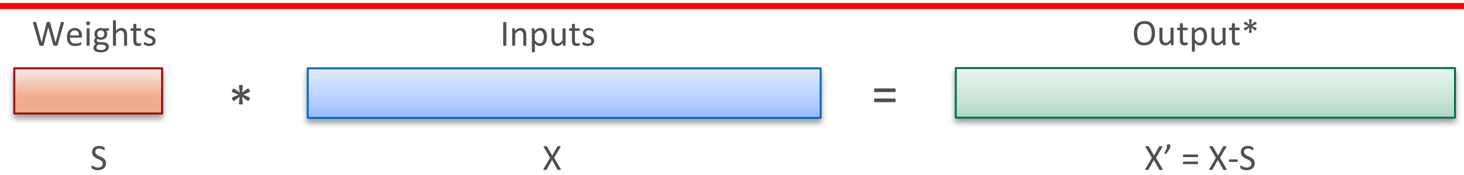


*Need **Spatial reduction** for output*

*Weight does not change over time => **Temporal reuse** opportunity*



Describing WS Dataflow



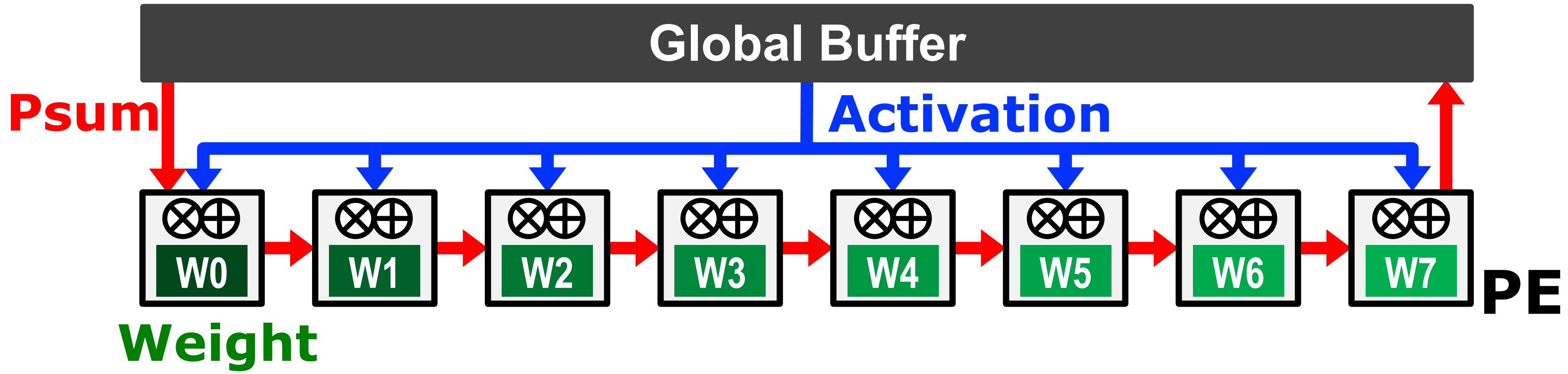
Computation

```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X’];     # Output activations

for (s = 0; s < S; s++) {
       for (x = 0; x < X’; x++) {
          o[x] += i[x+s]*w[s];
}
```

What about the loop nest makes it weight stationary?

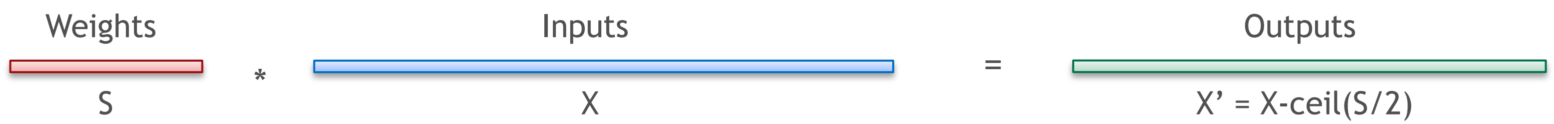
outermost loop is S rank

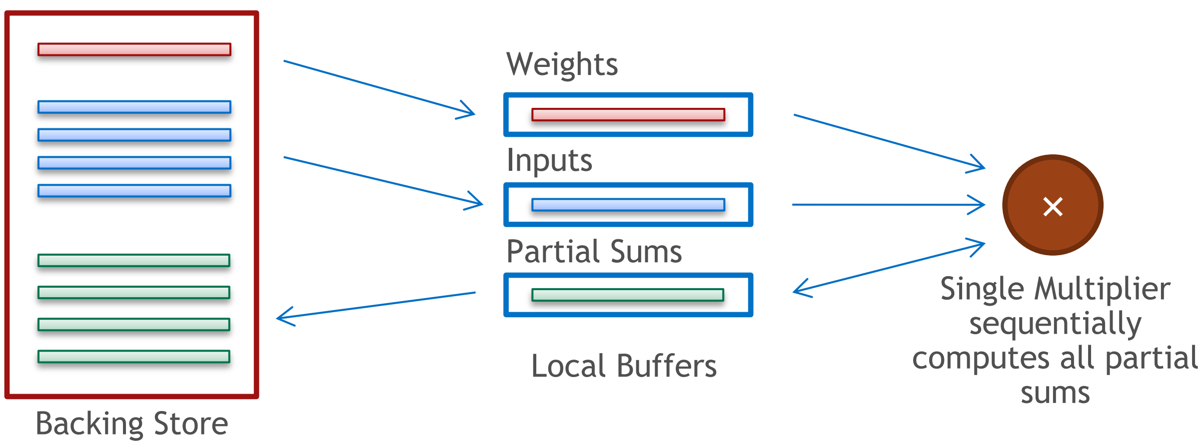
WS Dataflow Implementation



- **Minimize** **weight** read energy consumption
  - maximize convolutional and filter reuse of weights
- **Broadcast** **activations** and **accumulate** **psums** **spatially** across the PE array.

Simple Model for Mapping Dataflows to HW





Common metric	Weights	Inputs	Outputs / Partial Sums
Alg. Min. accesses to backing store (MINALG)	S	X	X’
Maximum operand uses (MAXOP)	SX'	SX'	SX'

1D Convolution Summary

Hardware Structure	Per Data Type	OS Dataflow Implication	WS Dataflow Implication
Bandwidth to MAC	Weight Fetch Rate	Every Cycle	Every S Cycles
	Input Fetch Rate	Every Cycle	Every Cycle
	Output Fetch Rate	Every S Cycles	Every Cycle
Local Buffer Sizes for Temporal Reuse	Weight Buffer Size	S	1
	Input Buffer Size	S	X’
	Output Buffer Size	1	X’
Total Local Buffer Accesses	Weight Buffer	X’	SX’
	Input Buffer	X’	S
	Output Buffer	SX’	S

Why is product always SX’?

Total computations same

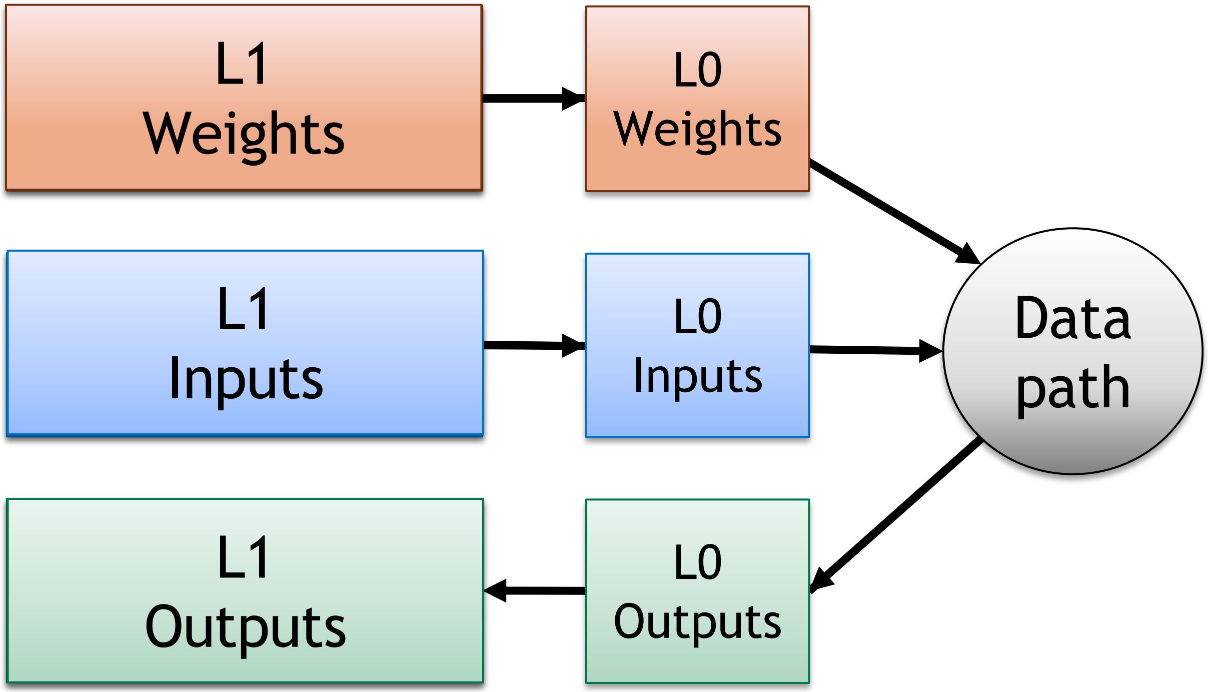
Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
  - Multi-layer Buffering
  - Multiple PEs
  - Full Convolution
- Advanced Dataflows

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
  - Multi-layer Buffering
  - Multiple PEs
  - Full Convolution
- Advanced Dataflows

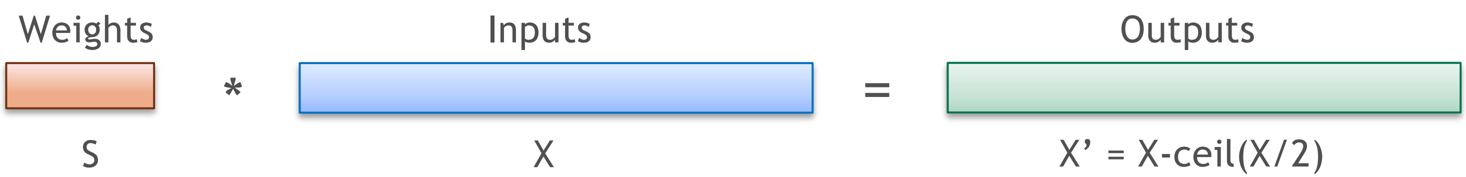
Multi-layer Buffering



How will this be reflected in the loop nest?

New 'level' of loops

1D Convolution – "Tiled"




```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
          x = x1 * X'0 + x0;
          s = r1 * R0 + r0;
          o[x] += i[x+s] * w[s];
}
```

Note X' and S are factored so:
X'0 * X'1 = X'
S0 * S1 = S

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
         o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
}
```

Constant over each level 1 iteration

- Level 0 buffer size is volume needed in each Level 1 iteration.
- Level 1 buffer size is volume needed to be preserved and re-delivered in future (usually successive) Level 1 iterations.

- A **legal mapping** will fit into the hardware's buffer sizes

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
         o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
}
```

Constant over each level 1 iteration

	Level 0	Level 1
Weights	S0	S
Inputs	X'0+S0	S
Outputs	X'0	1

Energy Costs

```
// Level 1
for (x1 = 0; x1 < X’1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X’0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
         o[x1*X’0+x0] += i[x1*X’0+x0 + s1*S0+s0] * w[s1*S0+s0];
}
```

Constant over each level 1 iteration

Energy of a buffer access is a function of the size of the buffer

Each buffer level’s energy is proportional the number of accesses at that level

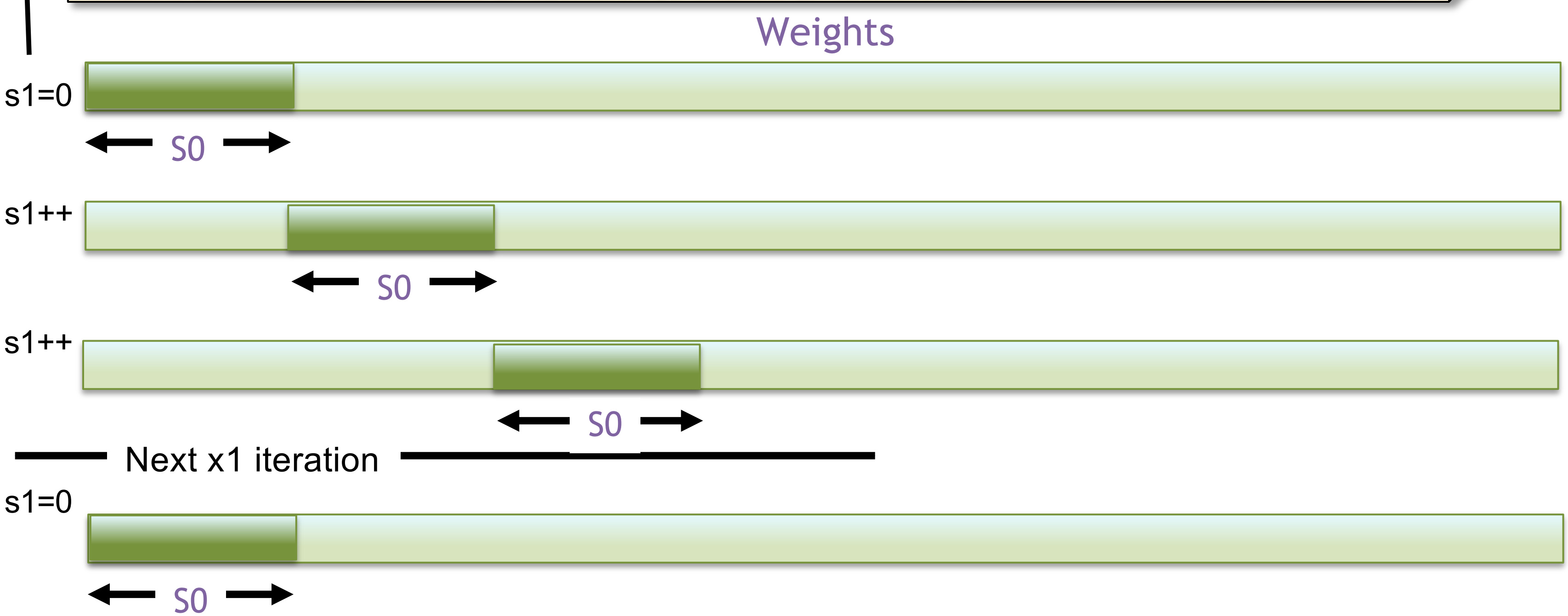
- For level 0 that is all the operands to the Datapath

For level L>0 there are three components:

- Data arriving from level L+1
- Data that needs to be transferred to level L-1
- Data that is returned from level L-1

Mapping – Weight Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
          o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S'0+s0];
}
```



Mapping – Weight Access Costs

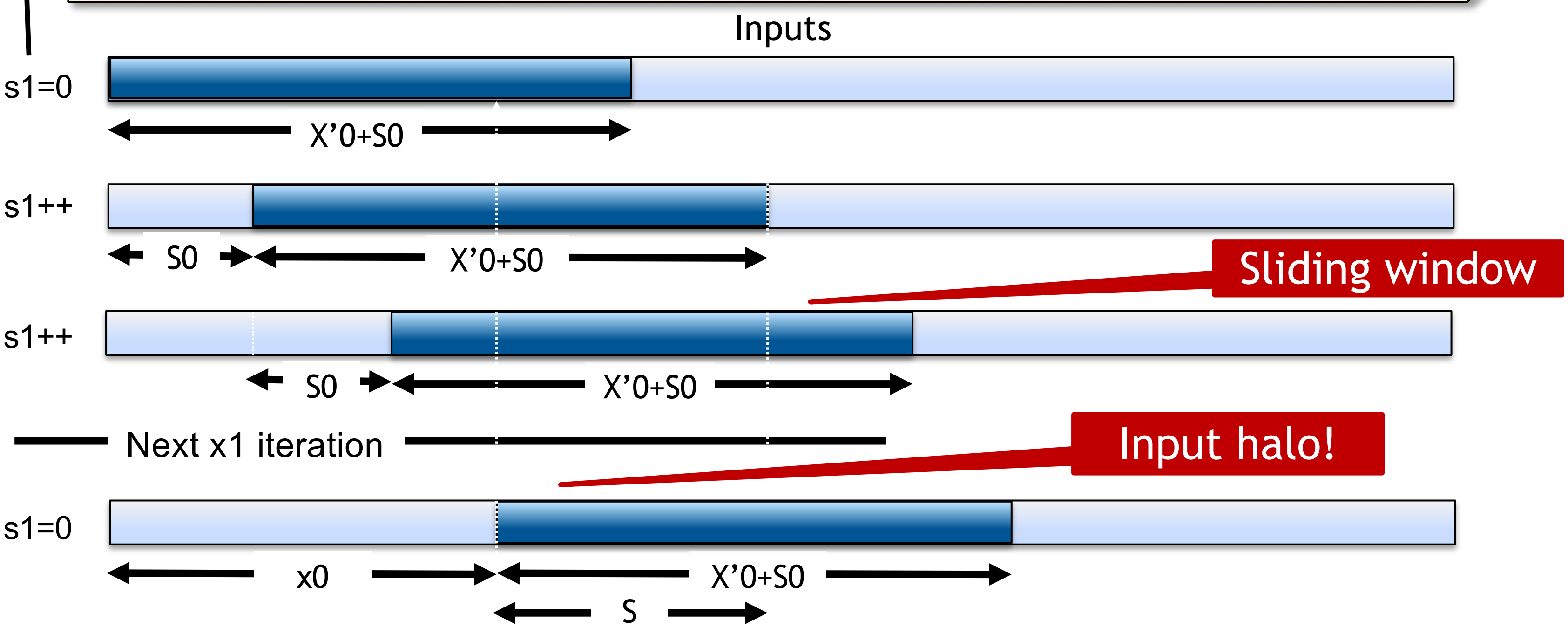
- Level 0 reads
  - Per level 1 iteration -> X'0*S0 weight reads
  - Times X'1*S1 level 1 iterations
  - Total reads = (X'0*S0)*(X'1*S1) = (X'0*X'1)*(S0*S1) = SX' reads

- Level 1 to 0 transfers
  - Per level 1 iteration -> S0 weights transferred
  - Times same number of level 1 iterations = X'1 * S1
  - Total transfers -> S0*(X'1*S1) = X'1*(S0*S1) = SX'1

Disjoint/partitioned reuse pattern

Mapping – Input Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
          o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
}
```



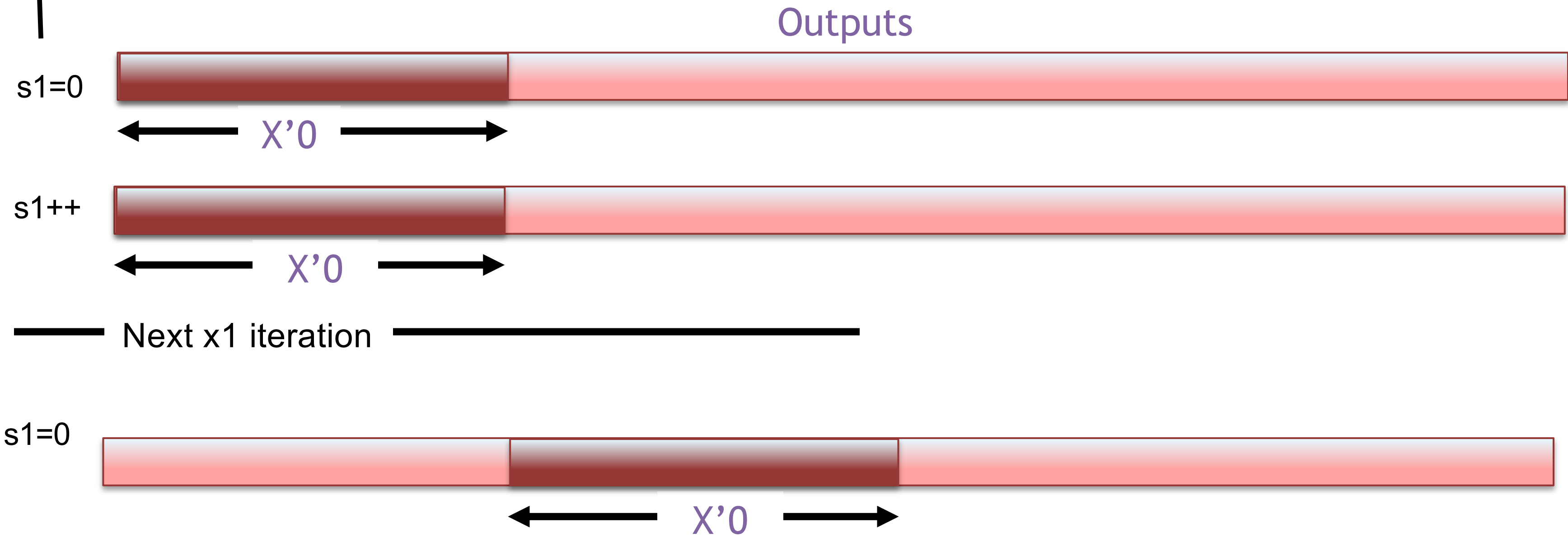
Mapping – Input Access Costs

- Level 0 reads
  - Per level 1 iteration -> X'0+S0 inputs reads
  - Times X'1*S1 level 1 iterations
  - Total reads = X'1*S1*(X'0+S0) = ((X'1*X'0)*S1)+(X'1*(S1*S0)) = X'*S1+X'1*S reads
- –
- Level 1 to 0 transfers
  - For s=0, X'0+S0 inputs transferred
  - For each of the following S1-1 iterations another S0 inputs transferred
  - So total per x1 iteration is: X'0+S0*S1 = X'0+S inputs
  - Times number of x1 iterations = X'1
  - So total transfers = X'1*(X'0+S) = (X'1*X'0)+X'1*S = X'+X'1*S

Sliding window/partitioned reuse pattern

Mapping – Output Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
}
```



Mapping – Output Access Costs

- Level 0 writes
  - Due to level 0 being ‘output stationary’ only X’0 writes per level 1 iteration
  - Times X’1*S1 level 1 iterations
  - Total writes = X’0*(X’1*S1) = (X’0*X’1)*S1 = X’*S1 writes
  -
- Level 0 to 1 transfers
  - After each S1 iterations a completed partial sum for X’0 outputs are transferred
  - There are X’1 chunks of S1 iterations
  - So total is X’1*X’0 = X’ transfers

Mapping Data Cost Summary

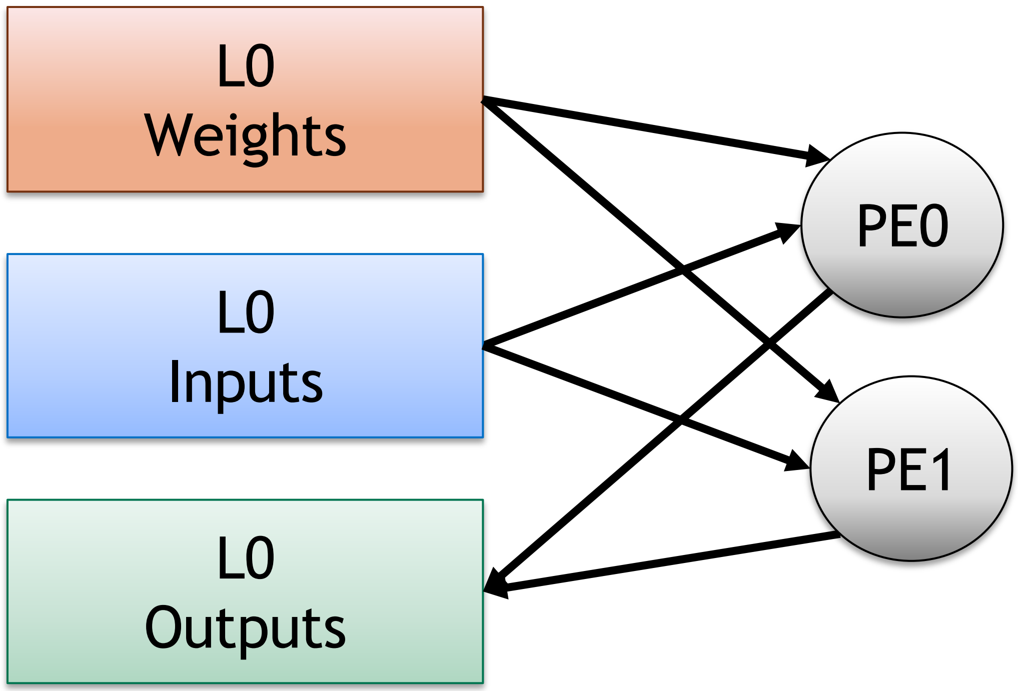
```
// Level 1
for (x1 = 0; x1 < X’1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X’0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
          o[x1*X’0+x0] += i[x1*X’0+x0 + s1*S0+s0]* w[s1*S0+s0];
}
```

	Level 0	Level 1 to 0 transfers
Weight Reads	SX’	SX’1
Input Reads	X’ * S1+ X’1 * S	X’+X’1*S
Output Reads	N/A	N/A
Output Writes	X’ * S1	X’

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
  - Multi-layer Buffering
  - Multiple PEs
  - Full Convolution
- Advanced Dataflows

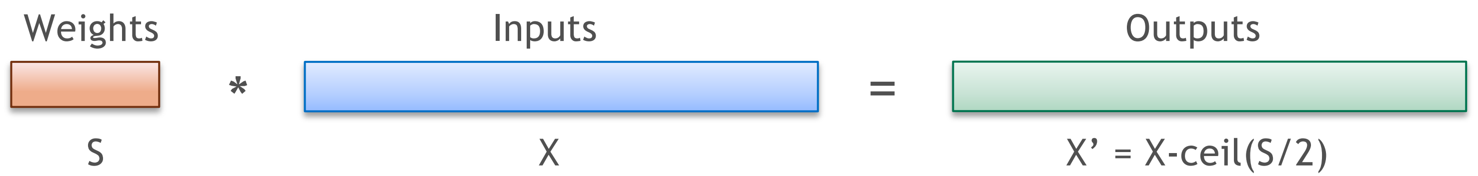
Spatial PEs



How will this be reflected in the loop nest?

New 'level' of loops

1D Convolution – Partition Outputs



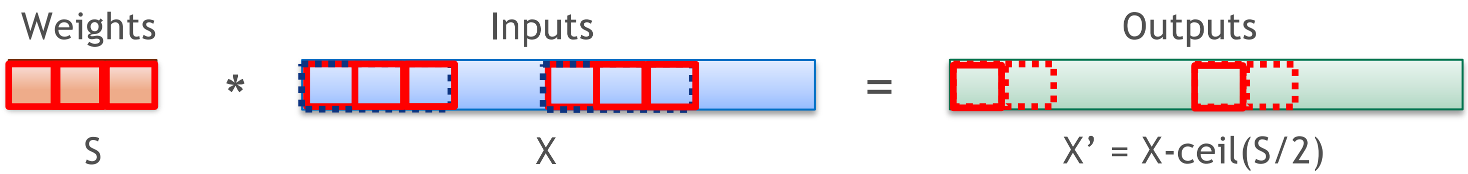

```
int i[X];       # Input activations
int w[S];       # Filter Weights
int o[X’];      # Output activations

// Level 1
parallel-for (x1 = 0; x1 < X’1; x1++) {
 ~~parallel-for (s1 = 0; s1 < S1; s1++) {~~
    // Level 0
    for (x0 = 0; x0 < X’0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
          o[x1*X’0+x0] += i[x1*X’0+x0 + s1*S0+s0]
                          * w[s1*S0+s0];
}
```

Note:
X’0*X’1 = X’
S0*S1 = S

X’1 = 2

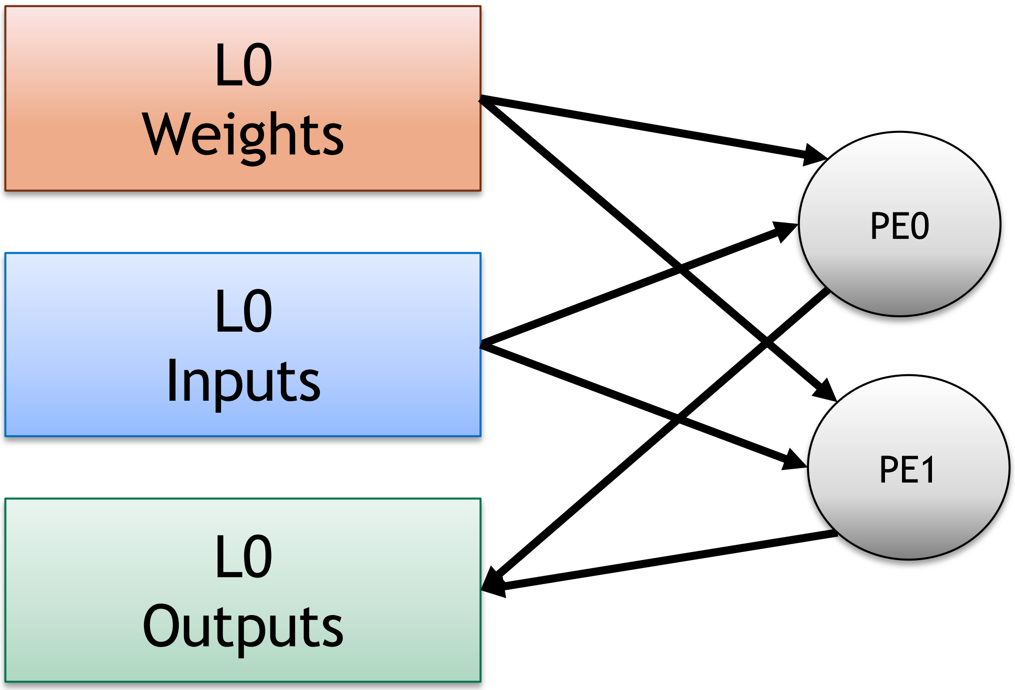
S1 = 1 => s1 = 0

1D Convolution – Partition Outputs




```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
          o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                        * w[s1*S0+s0];
}
```

Spatial PEs




```
PE0       PE1
>>w[0]    >>w[0]
```

Implementation opportunity?

Yes, single fetch and multicast

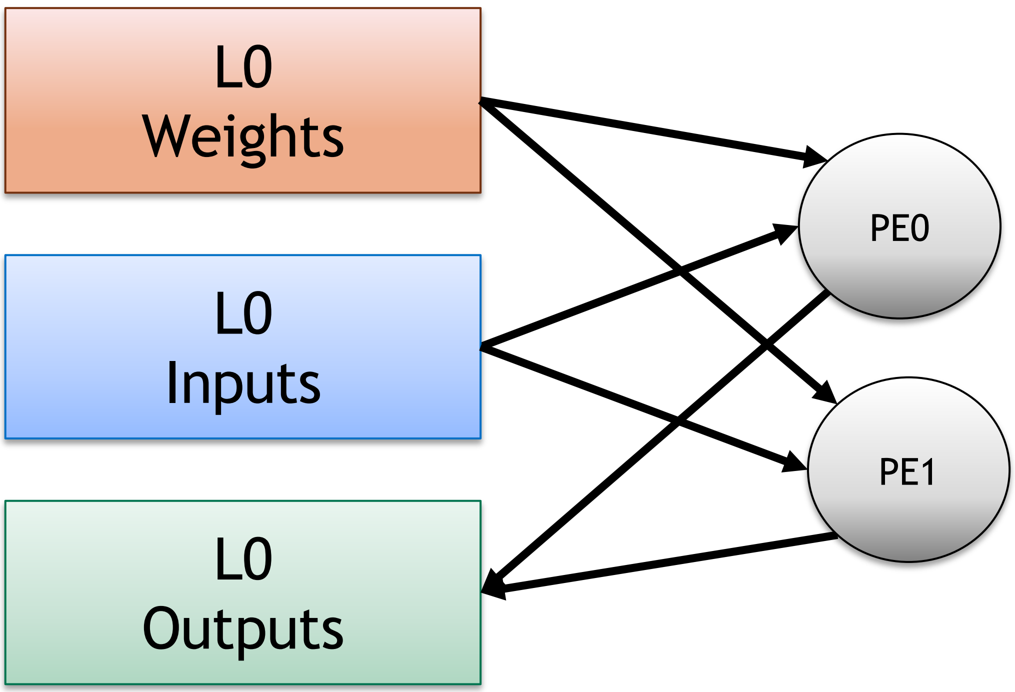
1D Convolution – Partition Outputs

```
// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X’0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
         o[x1*X’0+x0] += i[x1*X’0+x0 + s1*S0+s0]
                        * w[s1*S0+s0];
}
```

How do we recognize multicast opportunities?

Indices independent of spatial index

Spatial PEs: Partitioned Outputs



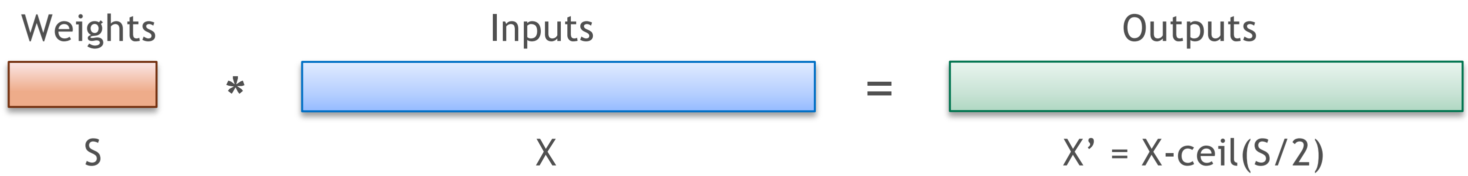
PE0	PE1
>>w[0]	>>w[0]
>>i[0]	>>i[X’0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X’0+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X’0+2]
<<o[0]	<<o[X’0+0]
>>w[0]	>>w[0]
>>i[1]	>>i[X’0+1]
>>w[1]	>>w[1]
>>i[2]	>>i[X’0+2]
>>w[2]	>>w[2]
>>i[3]	>>i[X’0+3]
<<o[1]	<<o[X’0+1]

Implementation opportunity?

Parallel fetch

Assuming S=3

1D Convolution – Partition Weights




```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X’];     # Output activations

// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
    // Level 0
    for (x0 = 0; x0 < X’0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X’0+x0] += i[x1*X’0+x0 + s1*S0+s0]
                      * w[s1*S0+s0];
}
```

Note:
X’0*X’1 = X’
S0*S1 = S

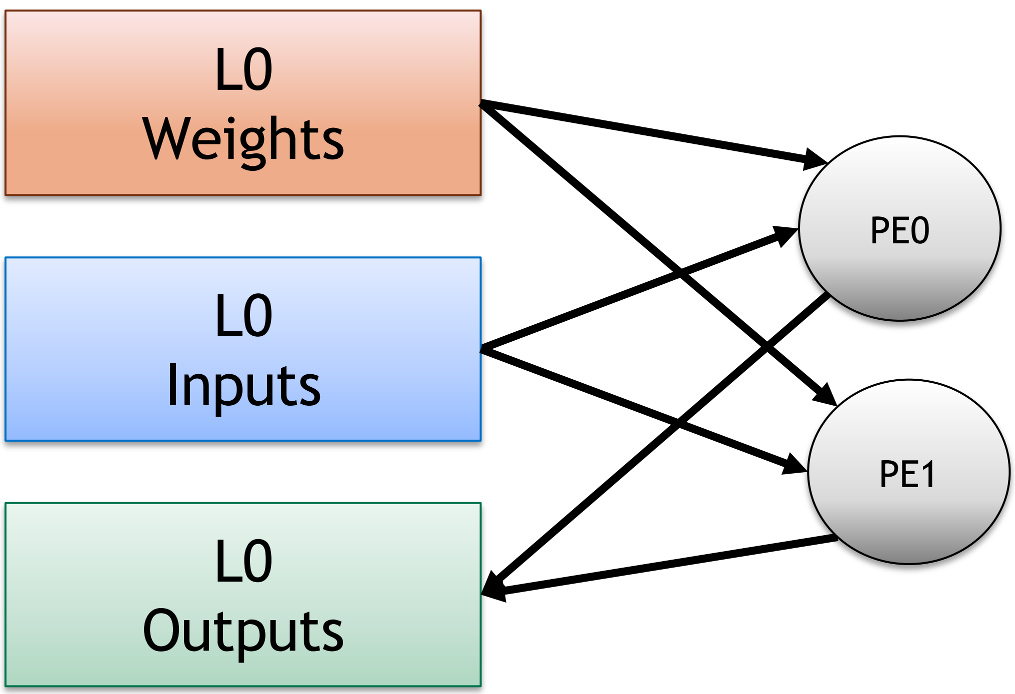
1D Convolution – Partition Weights

```
// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                      * w[s1*S0+s0];
}
```

How do we handle same index for output in multiple PEs? Spatial reduction

Other multicast opportunities? No

Spatial PEs: Partitioned Weights

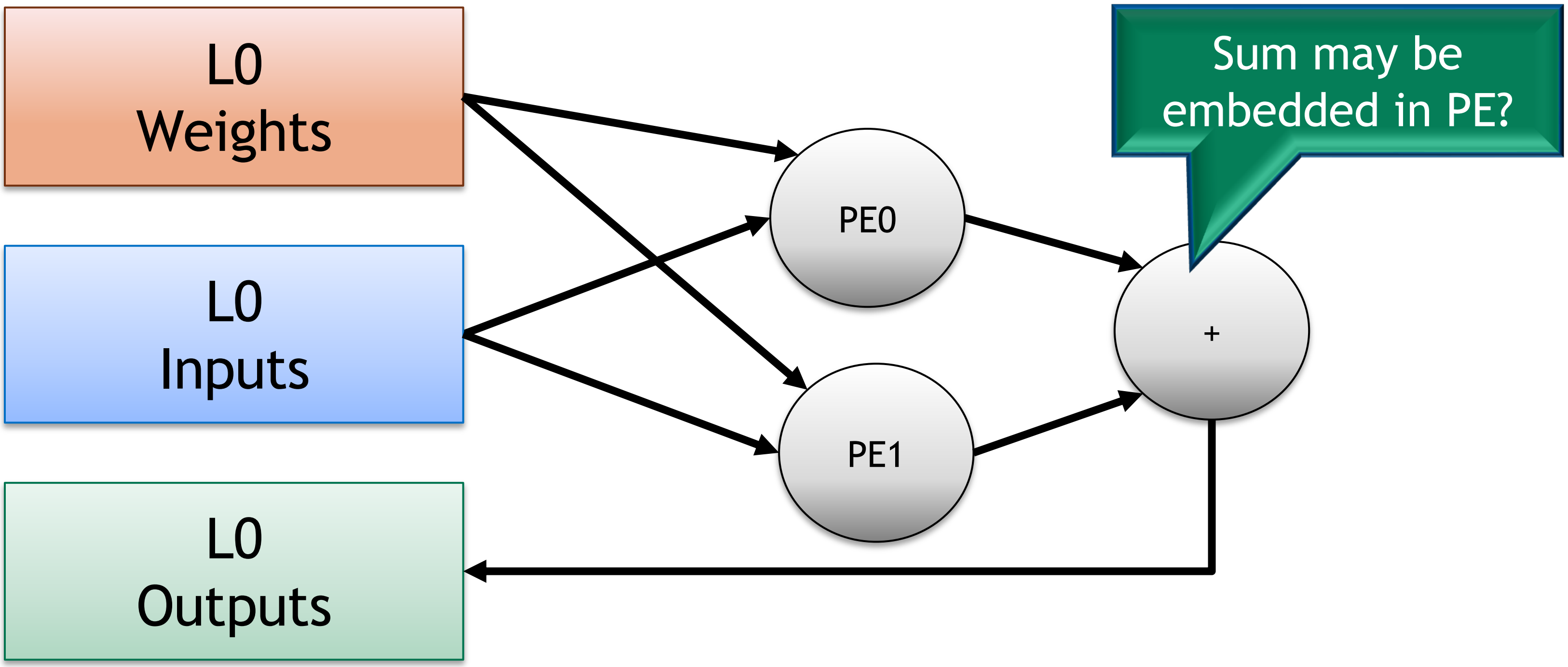


PE0	PE1
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]
>>w[0]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[1]	>>w[S0+2]
>>i[2]	>>i[S0+2]
>>w[2]	>>w[S0+3]
>>i[3]	>>i[S0+3]
<<o[1]	<<o[1]

Spatial sum needed? Yes

Assuming S=3

Spatial PEs with Spatial Summation



What if hardware cannot do a spatial sum?

Illegal mapping!

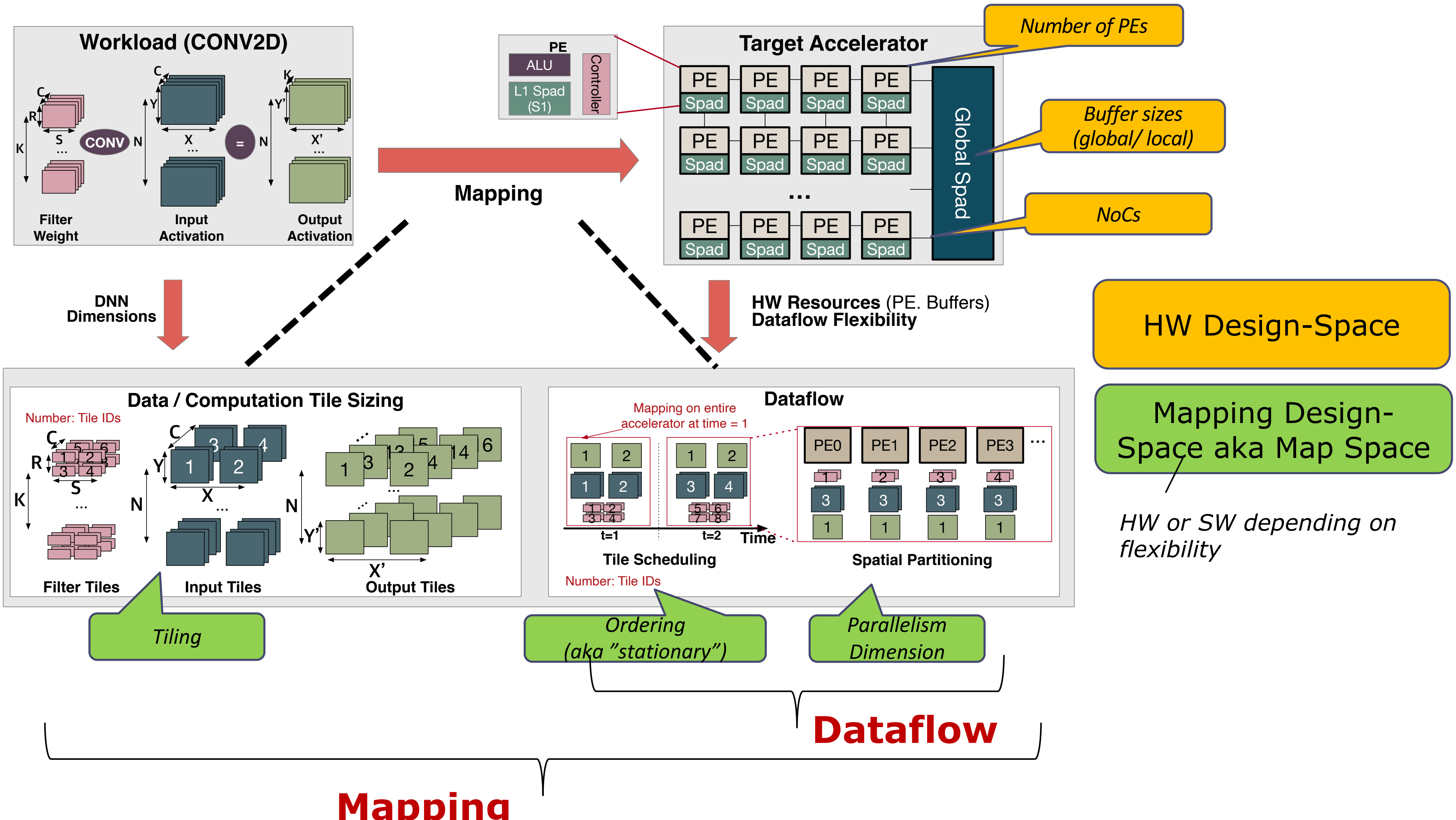
NoC Support

Hardware Structure	Per Data Type	Output-partitioned Dataflow Implication	Weight-partitioned Dataflow Implication
Network-on-Chip for Spatial Reuse	Weight Distribution	Spatial Multicast	Unicast
	Input Distribution	Unicast/Spatial Multicast	Unicast
	Output Collection	Temporal Reduction	Spatial Reduction

More Realistic Loop Nest

```
int i[W];       # Input activations
int w[R];       # Filter Weights
int o[E];       # Output activations

// Level 2
for (x2 = 0; x2 < X'2; x2++) {
  for (s2 = 0; s2 < S2; s2++) {
    // Level 1
    parallel-for (x1 = 0; x1 < X'1; x1++) {
      parallel-for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X'0; x0++) {
          for (s0 = 0; s0 < S0; s0++) {
            ...
```

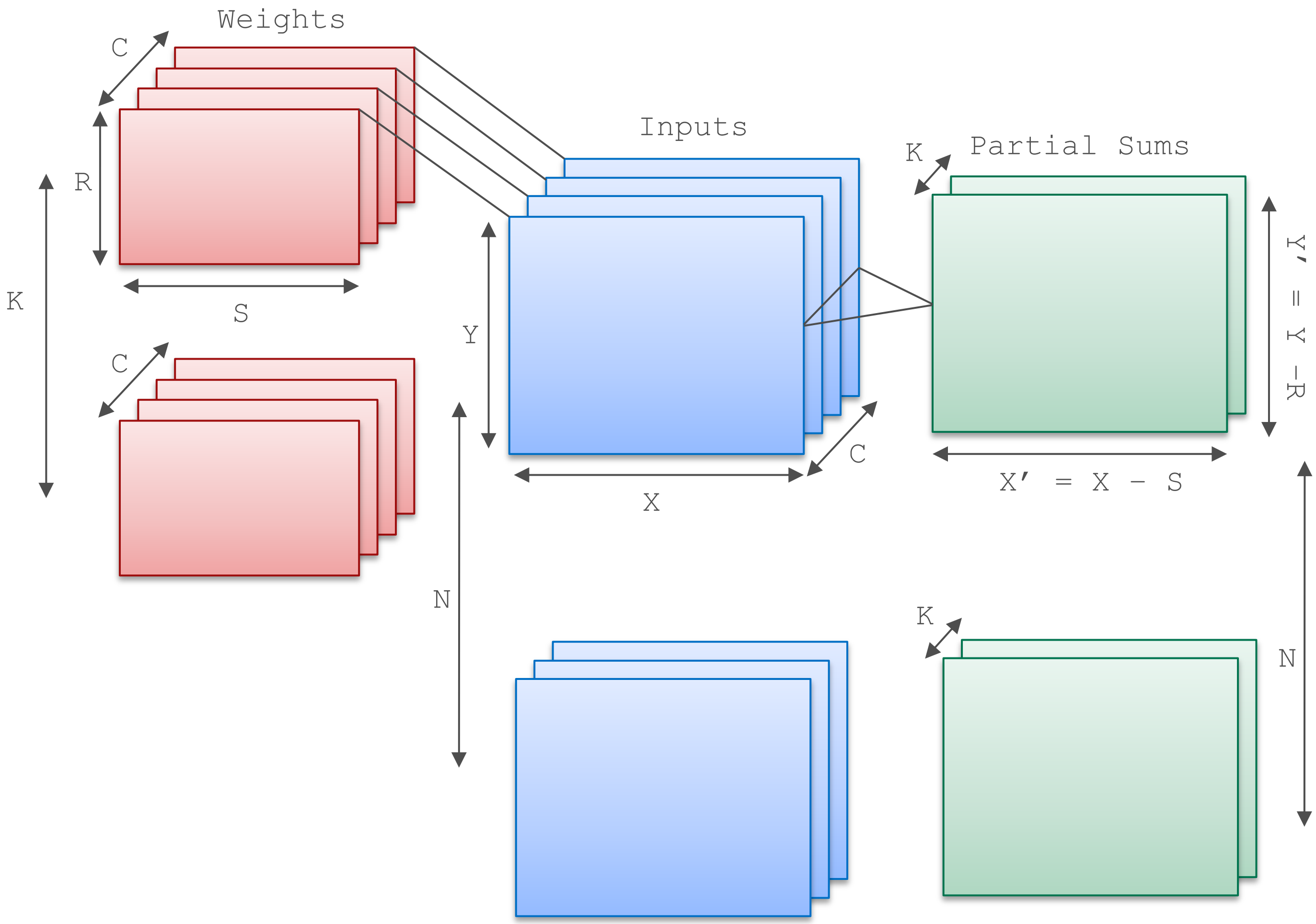
Design-space of a DNN Accelerator



Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
  - Multi-layer Buffering
  - Multiple PEs
  - Full Convolution
- Advanced Dataflows

Mapping a Full Convolution



Reference Convolution Layer

```
int i[C][Y][X];       # Input activation channels
int w[K][C][R][S];    # Filter weights (per channel pair)
int o[K][Y’][X’];     # Output activation channels

for (k = 0; k < K; k++) {
  for (y = 0; y < Y’; y++) {
    for (x = 0; x < X’; x++) {
      for (c = 0; c < C; c++) {
        for (r = 0; r < R; r++) {
          for (s = 0; s < S; s++) {
            o[k][y][x] += i[c][y+r][x+s]*w[k][c][r][s];
```

Describing a full accelerator

```
Input Fmaps:    I[G][N][C][H][W]
Filter Weights: W[G][M][C][R][S]
Output Fmaps:   O[G][N][M][E][F]

// DRAM levels
for (g3=0; g3<G3; g3++) {
 for (n3=0; n3<N3; n3++) {
  for (m3=0; m3<M3; m3++) {
   for (f3=0; f3<F3; f3++) {
    // Global buffer levels
    for (g2=0; g2<G2; g2++) {
     for (n2=0; n2<N2; n2++) {
      for (m2=0; m2<M2; m2++) {
       for (f2=0; f2<F2; f2++) {
        for (c2=0; c2<C2; c2++) {
         for (s2=0; s2<S2; s2++) {
          // NoC levels
          parallel-for (g1=0; g1<G1; g1++) {
           parallel-for (n1=0; n1<N1; n1++) {
            parallel-for (m1=0; m1<M1; m1++) {
             parallel-for (f1=0; f1<F1; f1++) {
               parallel-for (c1=0; c1<C1; c1++) {
                parallel-for (s1=0; s1<S1; s1++) {
                 // SPad levels
                 for (f0=0; f0<F0; f0++) {
                  for (n0=0; n0<N0; n0++) {
                   for (e0=0; e0<E; e0++) {
                    for (r0=0; r0<R; n++) {
                     for (c0=0; c0<C0; c++) {
                      for (m0=0; m0<M0; m++) {
                       O += I × W;
}}}}}}}}}}}}}}}}}}}}}}
```

DRAM to Accelerator

Inter-PE Global Buffer

Inter-PE NoC

Intra-PE

A Mapping Representation

- For each temporal and spatial level:
  - Permutation order of all indices
  - Partitioning of data volume for all indices (factoring)
    - $\forall X \in indices \left(\prod_{l=0}^{maxlevel} X_l\right) \geq X_{total}$
  - Data bypass flag per datatype (for temporal layers)

$(N_l\ K_l\ C_l\ X'_l\ Y'_l\ R_l\ S_l)\ [I_l, W_l, O_l]$

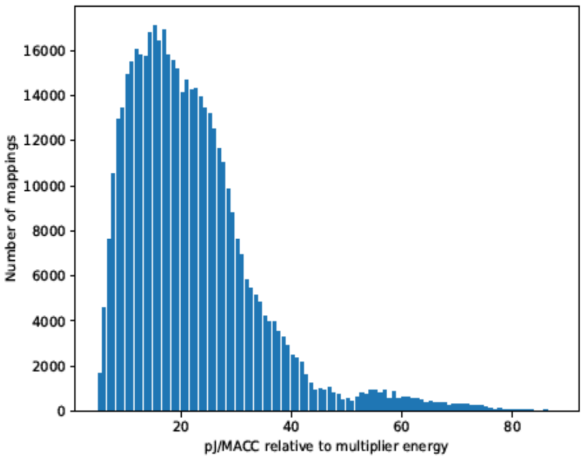
How many permutations per level? (# Indices)!

How many bypass combinations per level? 2^N

Total choices per temporal level? (# Indices)! * 2^N * factorings

Impact of Mappings

VGG conv3 2 Layer. Source: Timeloop



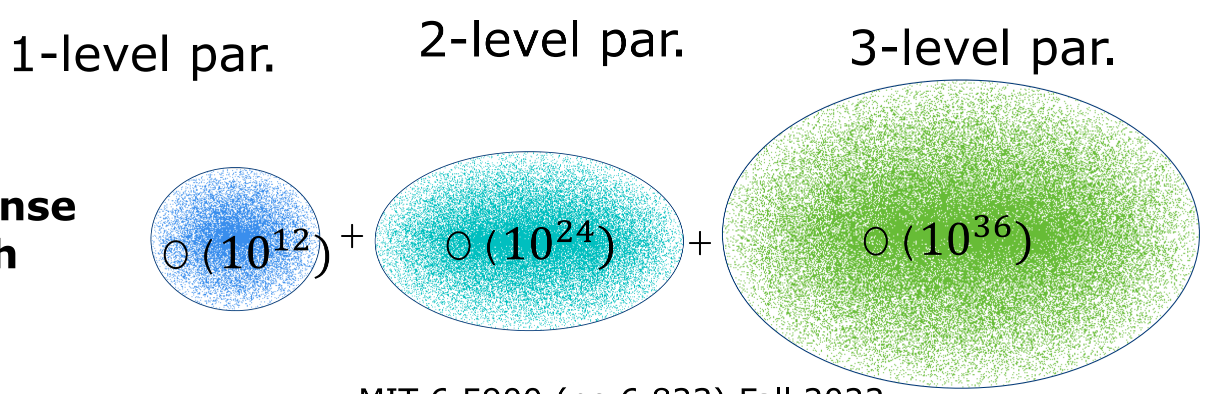
480,000 mappings shown

Spread: 19x in energy efficiency

Only 1 is optimal, 9 others within 1%

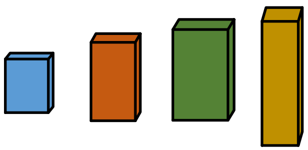
6,582 mappings have min. DRAM accesses but vary 11x in energy efficiency

Immense Search space



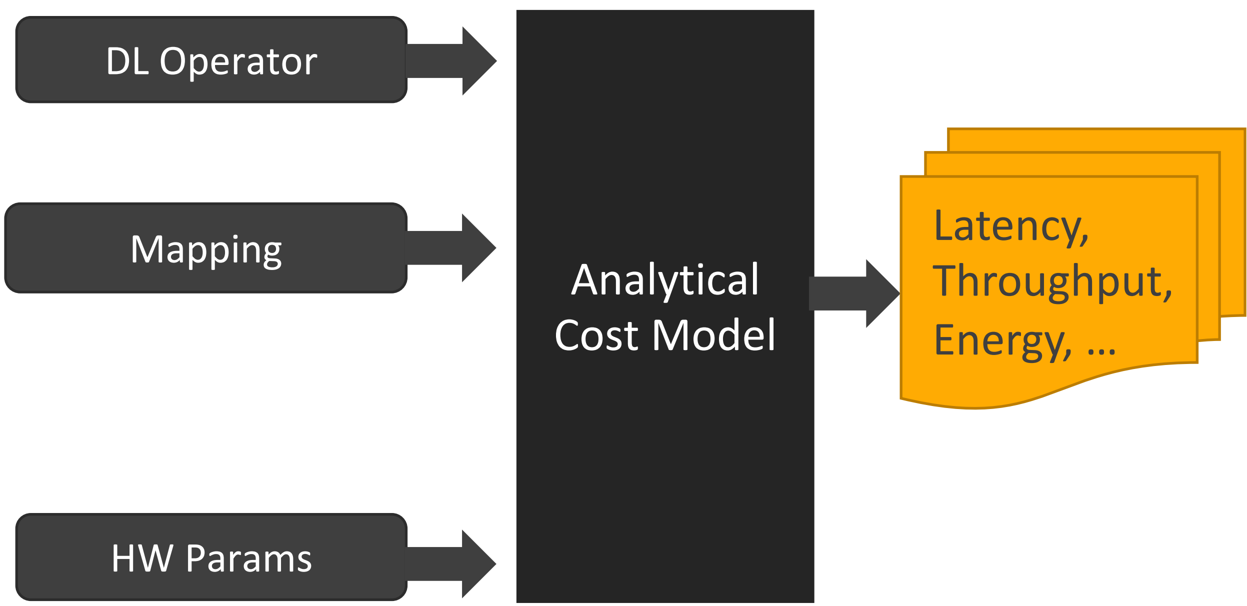
Exploring Mappings

- Gigantic space of potential loop orders & factorizations
- Cycle-accurate modeling of realistic dimensions and fabric sizes too slow
- Solution: use an analytic modeling



```
int i[C][Y][X];     # Input activation channels
int w[K][C][R][S];  # Filter weights (per channel pair)
int o[K][Y’][X’];   # Output activation channels

for (k = 0; k < K; k++) {
  for (y = 0; y < Y’; y++) {
    for (x = 0; x < X’; x++) {
      for (c = 0; c < C; c++) {
        for (r = 0; r < R; r++) {
          for (s = 0; s < S; s++) {
            o[k][y][x] += i[c][y+r][x+s]*w[k][c][r][s];
```

⋮



e.g.,: Timeloop (ISPASS 2019), MAESTRO (MICRO 2019), ..

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows
  - Fusion
  - Sparsity

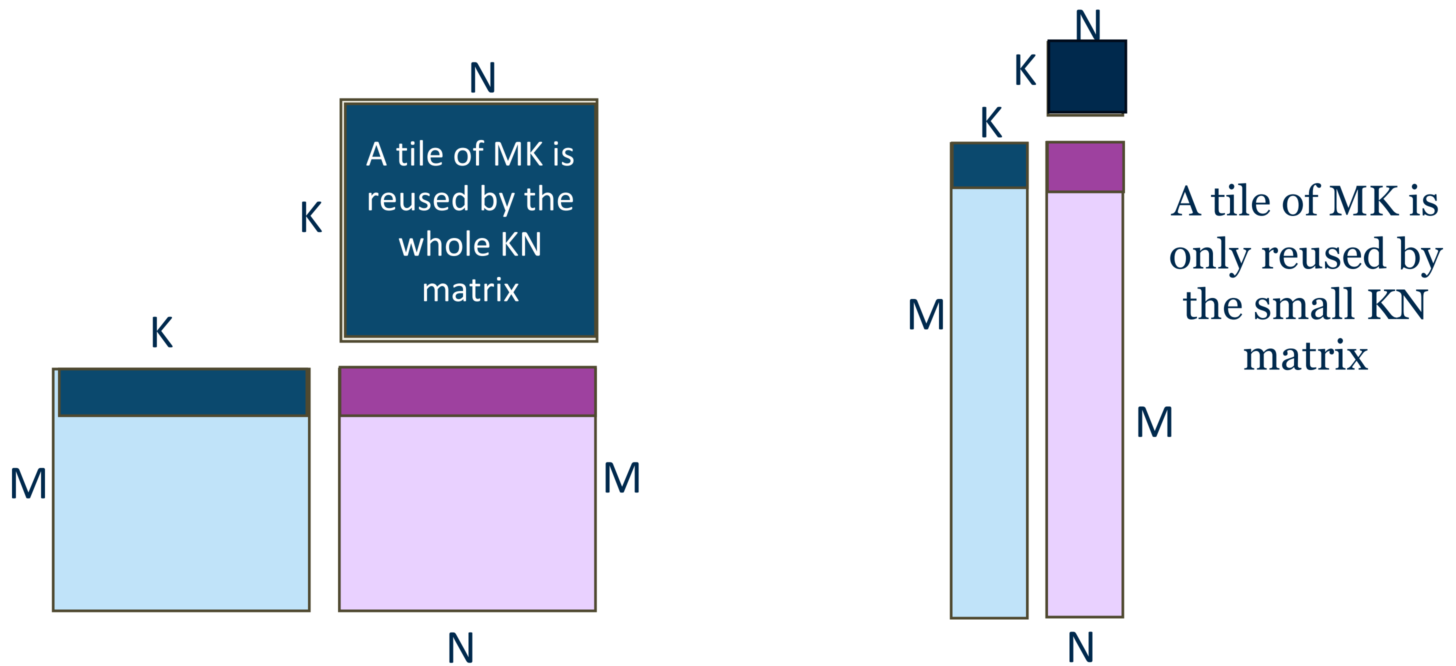
Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows
  - Fusion
  - Sparsity

Not All GEMMs are Compute Bound

Even in the best case with infinite on-chip storage and large number of PEs.

$$AI_{best\_GEMM} = \frac{M \times K \times N}{M \times K + K \times N + M \times N}$$



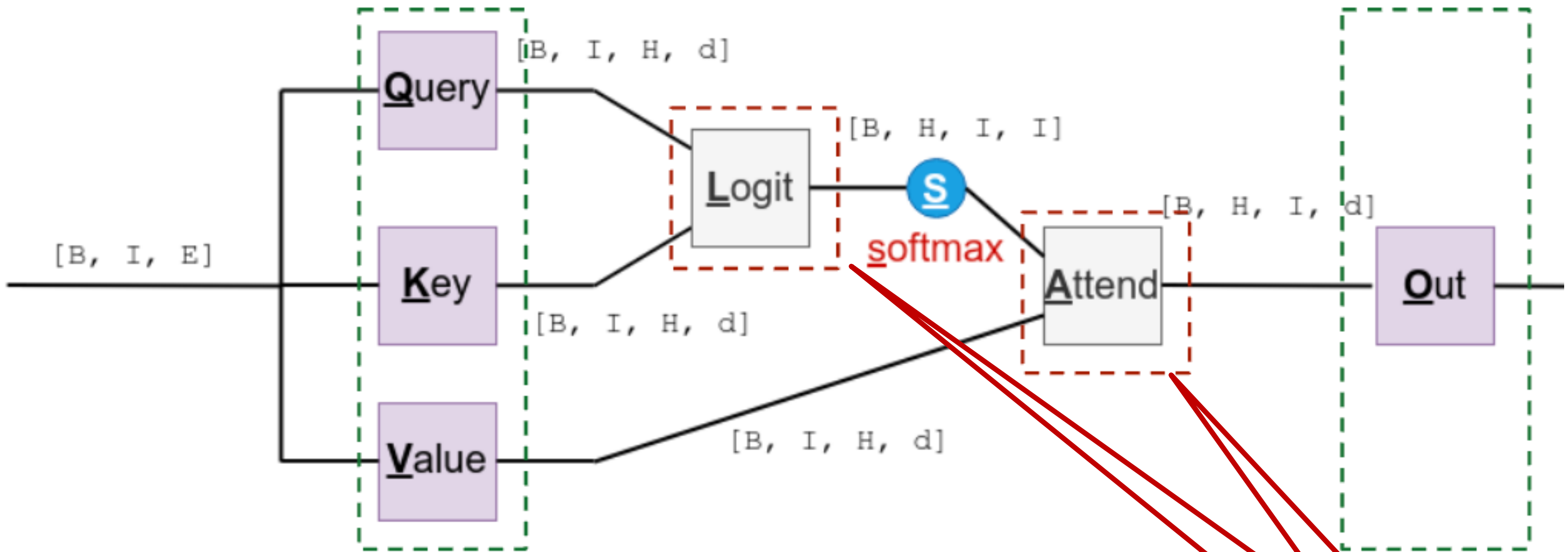
Regular GEMM (M=1024, K=1024, K=1024)

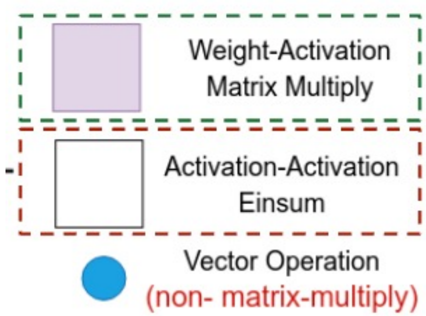
$AI_{best\_GEMM} = 341.33\ ops/word$

Skewed GEMM (M=1048576, N=32, K=32)

$AI_{best\_GEMM} = 16\ ops/word$

GEMMs in Attention Layers





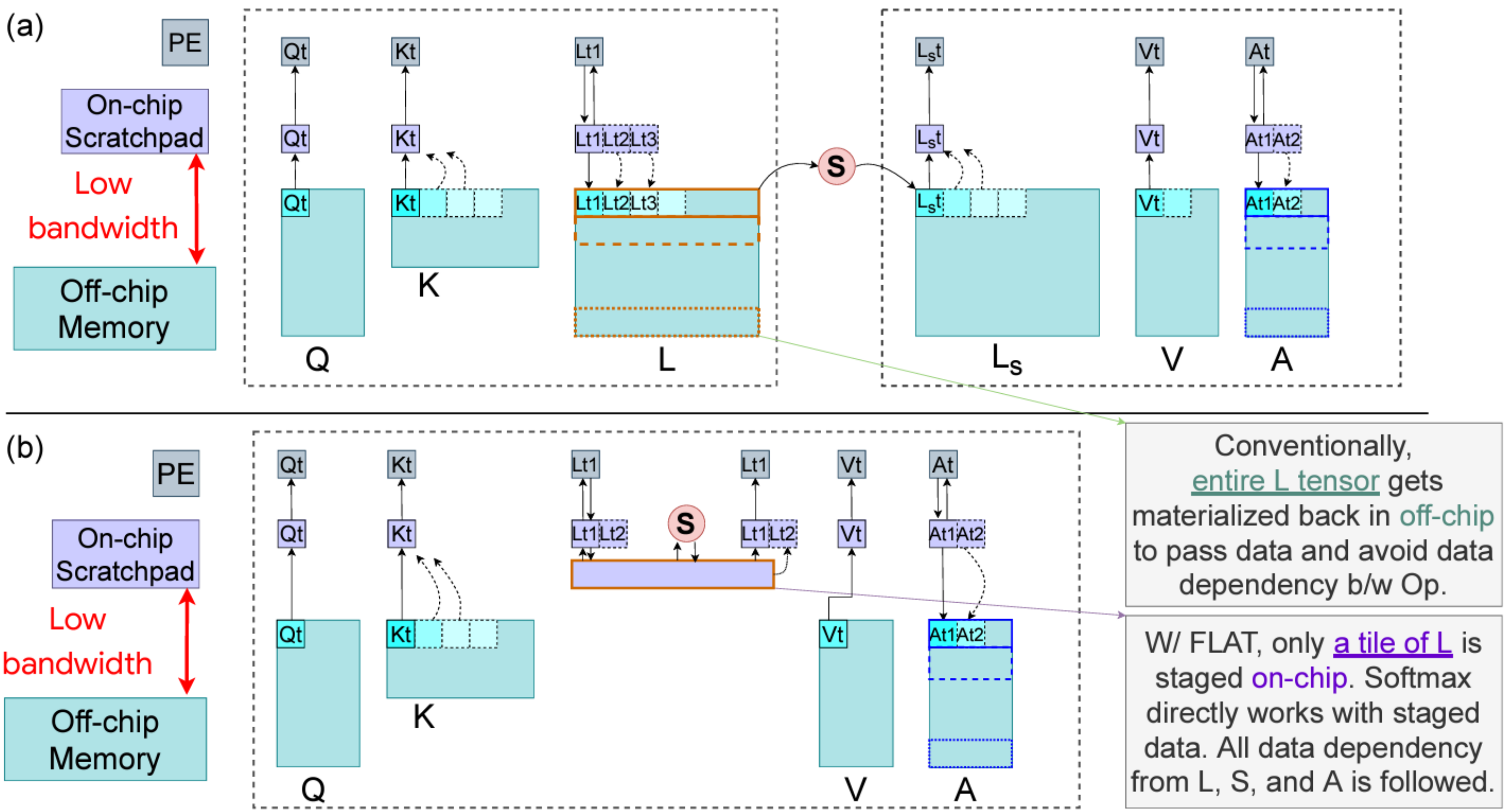
Compute-bound ⇒ Intra-operator dataflow to improve reuse is effective

Memory-bound ⇒ Intra-operator dataflow is **not** effective

Activation – Activation GEMMs

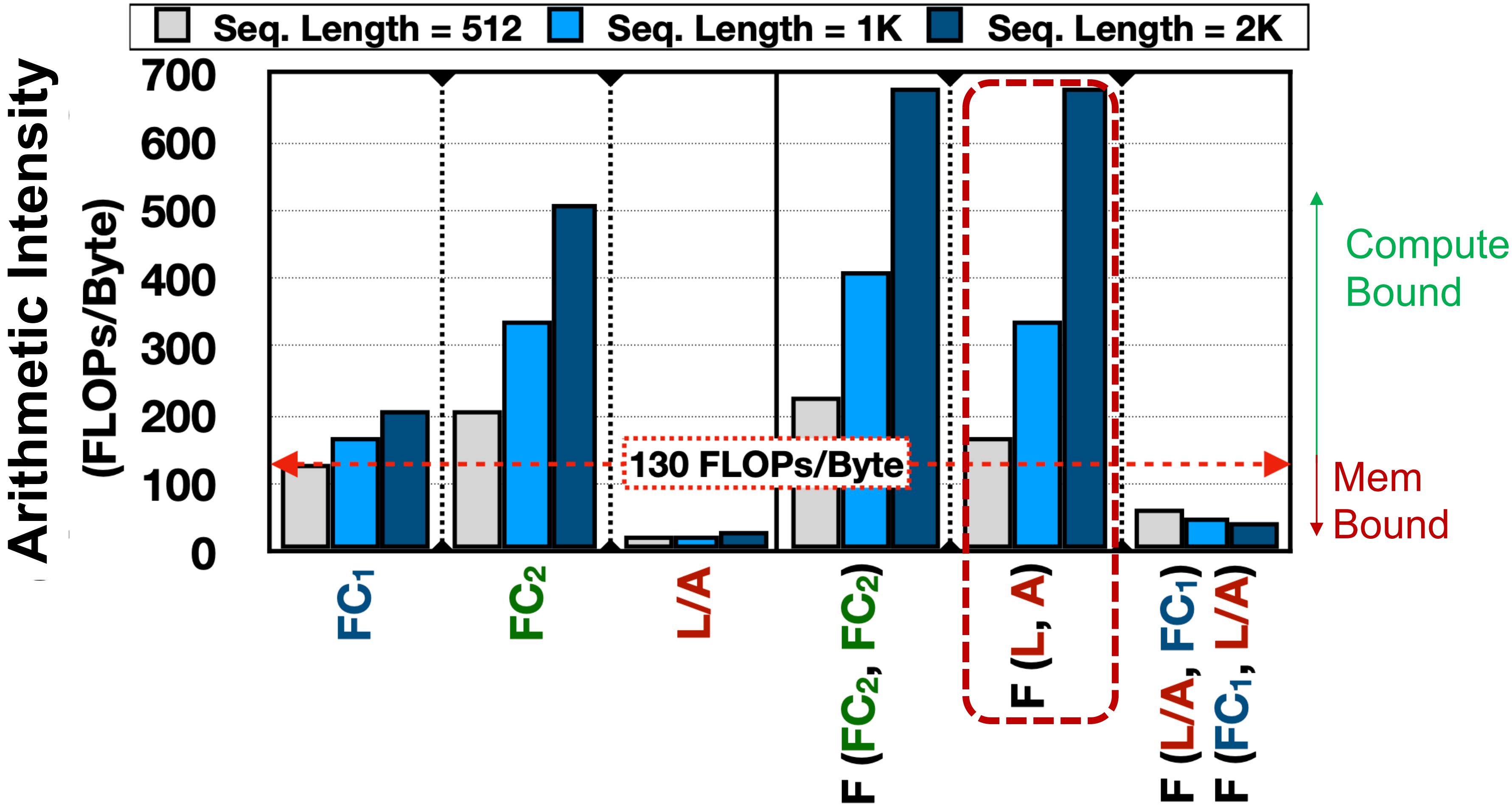
Opportunity: Fusion

- Key Intuition: "Reuse" the intermediate output immediately



Kao et al, "FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks", ASPLOS 2023

Opportunity: Fusion

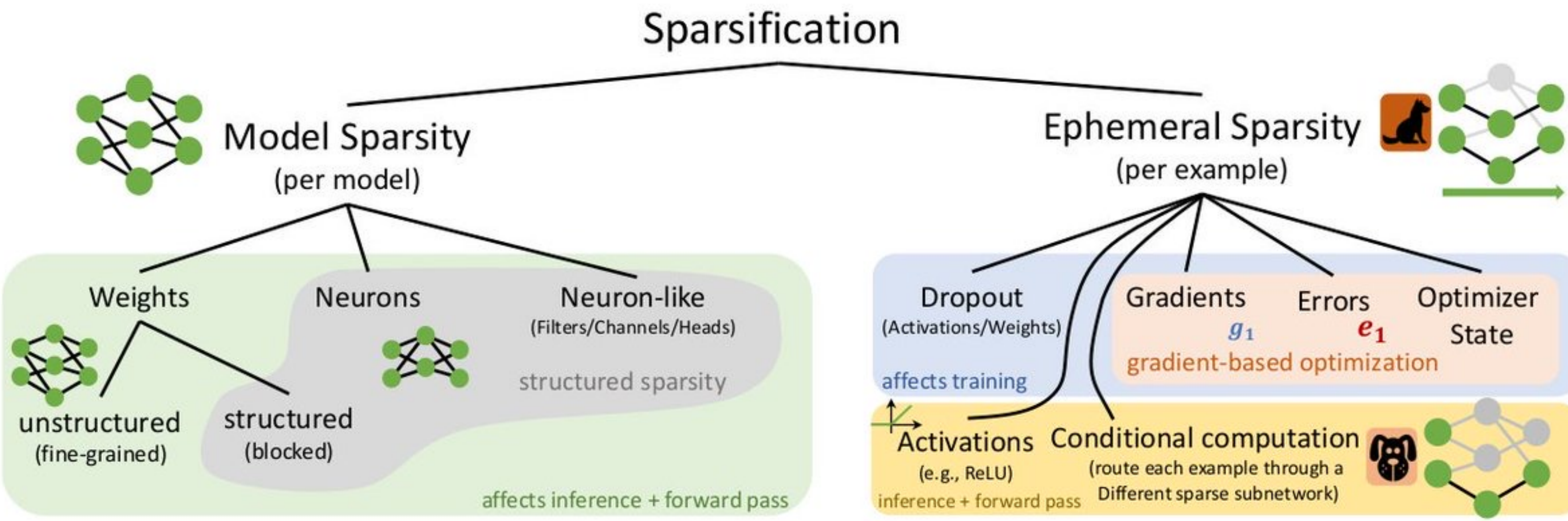


Kao et al, "FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks", ASPLOS 2023

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows
  - Fusion
  - Sparsity

Sparsity in DNNs

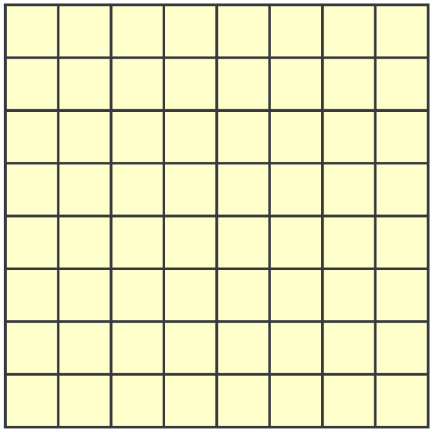


Source: *Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks*

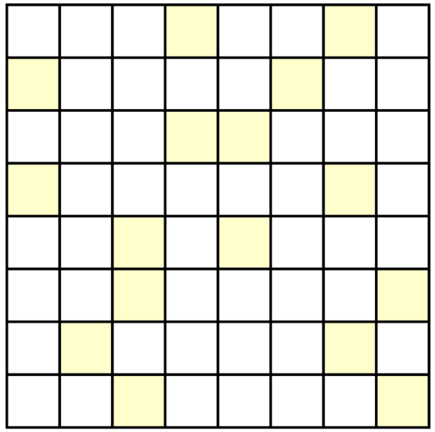
Figure source: https://htor.inf.ethz.ch/sparsity-in-dl/

10-90% sparsity across ML Models today

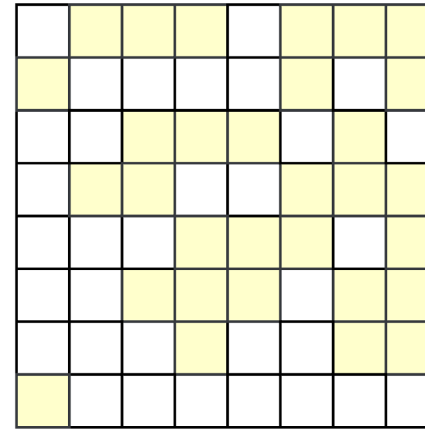
Sparsity Patterns



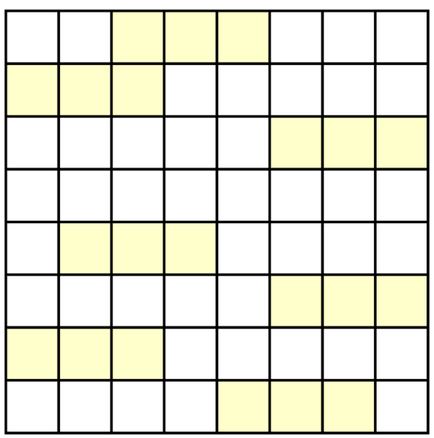
DENSE



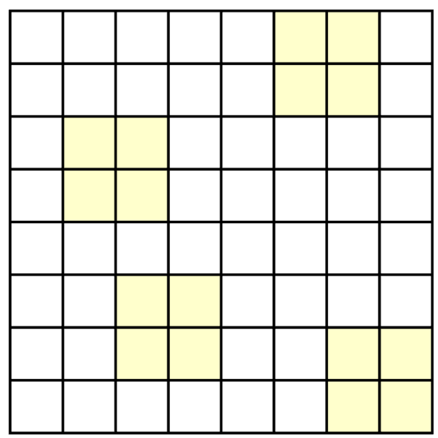
Block Balanced (Eg: N:M)



Unstructured

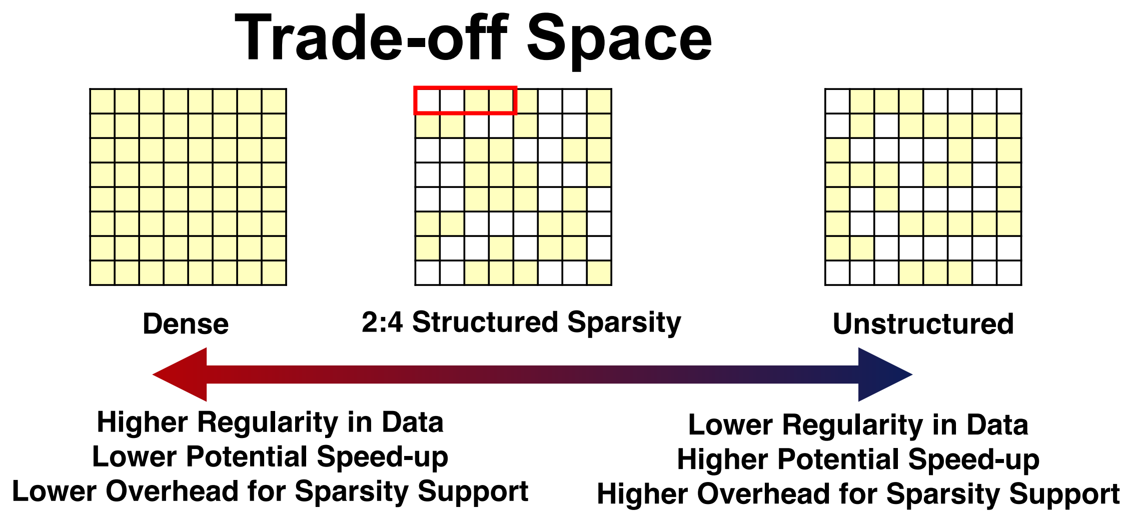


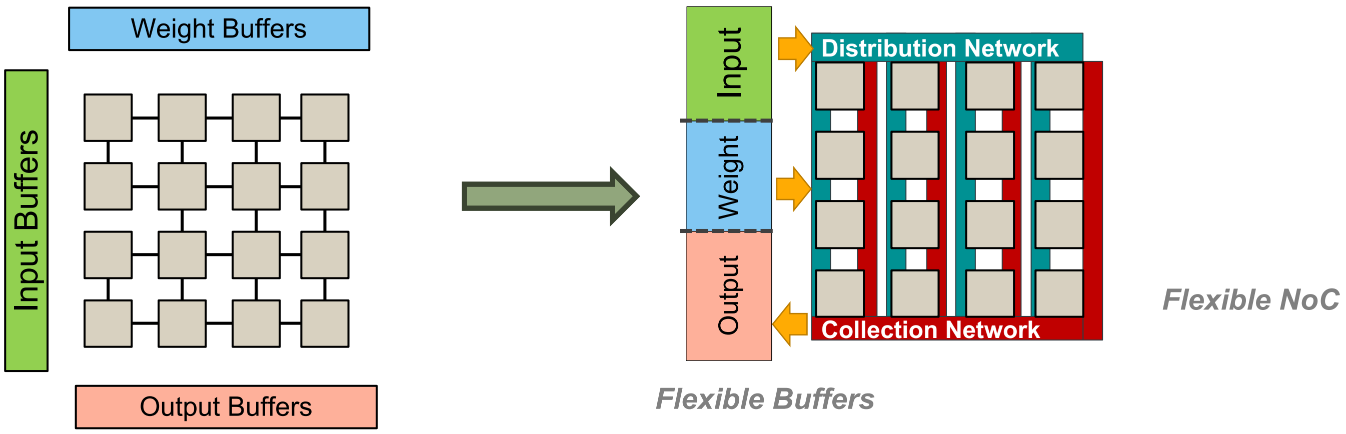
1D Blocks



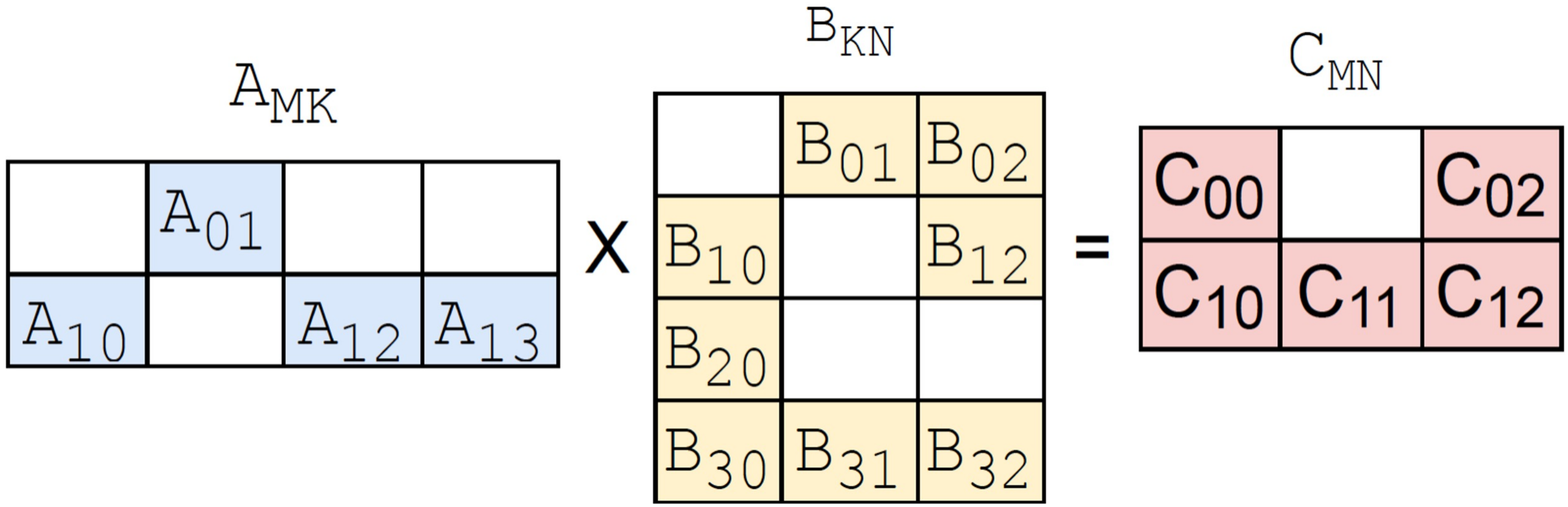
2D Blocks

Sparse Accelerators





Sparse Dataflows



- Inner Product
- Outer Product
- Gustavson's

Active area of research!

Thank you!