INTRODUCTION TO DATAFLOW FOR DEEP LEARNING ACCELERATOR DESIGN

Michael Pellauer*, Hyoukjun Kwon** and Tushar Krishna**

*Architecture Research Group, NVIDIA

**Georgia Institute of Technology

ACKNOWLEDGMENTS

Many slides from: Professors Joel Emer (NVIDIA, MIT) and Vivienne Sze's (MIT) class

- 6.S082/6.888 Hardware Architectures for Deep Learning (Fall, 2017)

Also includes slides from: Angshuman Parashar, Senior Research Scientist (NVIDIA)

Also influenced by:

Steve Keckler (NVIDIA)
Jason Clemons (NVIDIA)
Sophia Shao (NVIDIA)
Cristopher Fletcher (UIUC)

Former Nvidia interns:

Yu-Hsin Chen (MIT)
Anurag Mukkara (MIT)
Animesh Jain (U Mich.)

TALK OUTLINE

Motivation and Background

- Why architecture researchers should care about dataflow

Concepts in Tiling, Blocking, and Scheduling

- Basic example: 1D-convolution
- Adding intermediate staging buffers
- Adding spatial parallelism

Extending to full CNN layers

- The need for analytic modeling

ACCELERATORS ARE GREAT…. BUT!

Custom Datapath

Off-Chip Memory

MOTIVATION: DATA MOVEMENT

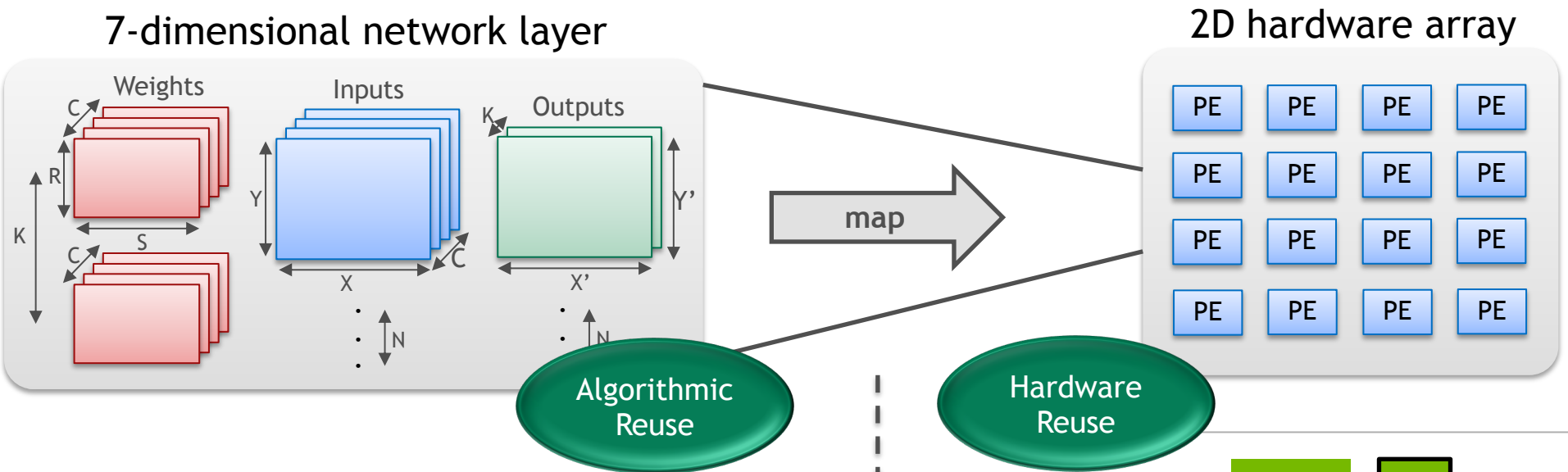
Why it's important

Energy costs	
8-bit Integer Multiply	1x
Fetch two 8-bit operands from DRAM	~100x
Fetch two 8-bit operands from large SRAM	~10x

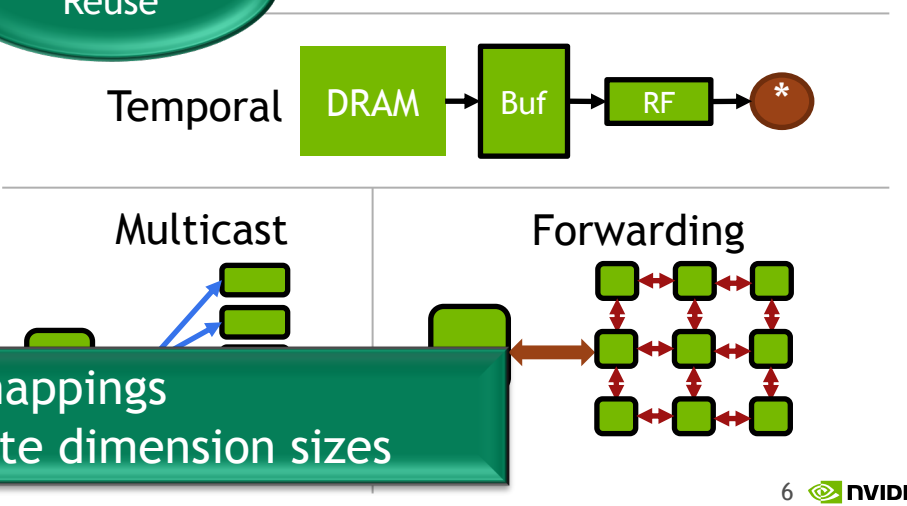
Fortunately…

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

MAPPING REUSE TO HARDWARE



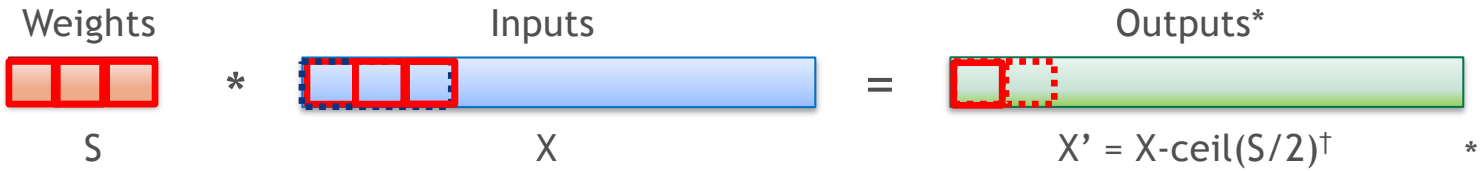
- **7D Computation Space**
  - R * S * X * Y * C * K * N
- **4D Operand / Result Spaces -**
  - Weights - R * S * C * K
  - Inputs - X * Y * C * N
  - Outputs



Millions of non-trivial mappings
Efficiency is dependent on concrete dimension sizes

LEARNING ABOUT DATAFLOWS

PEDAGOGICAL EXAMPLE: 1-D CONVOLUTION




```
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];
}
```

* The terms "Output" and "Partial Sum" used interchangeably

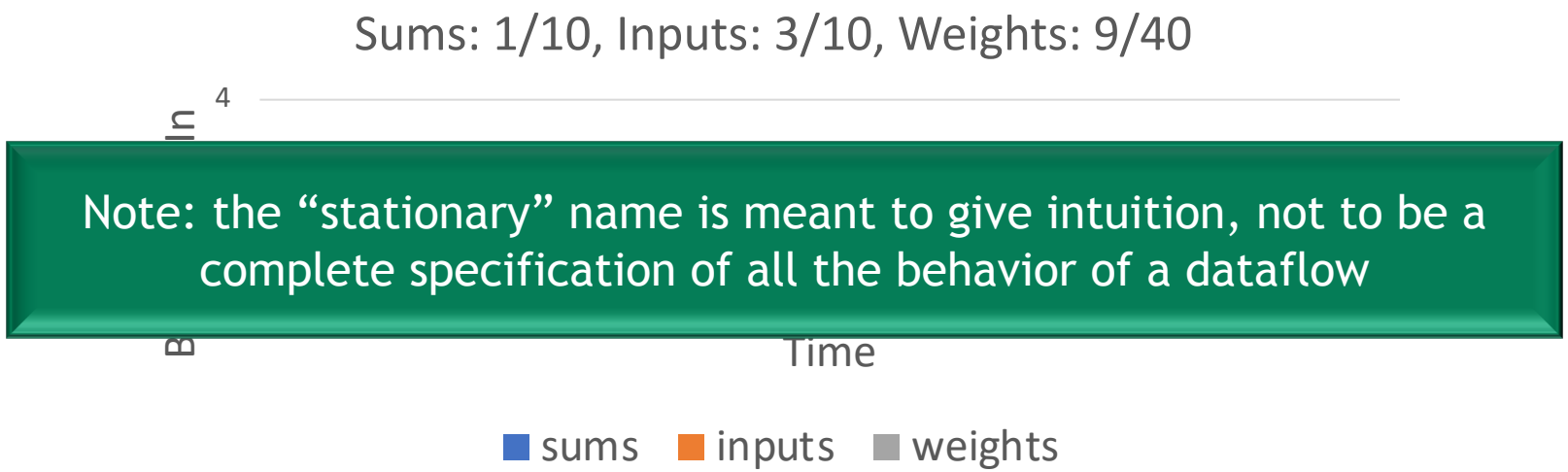
† Assuming: 'valid' style convolution

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

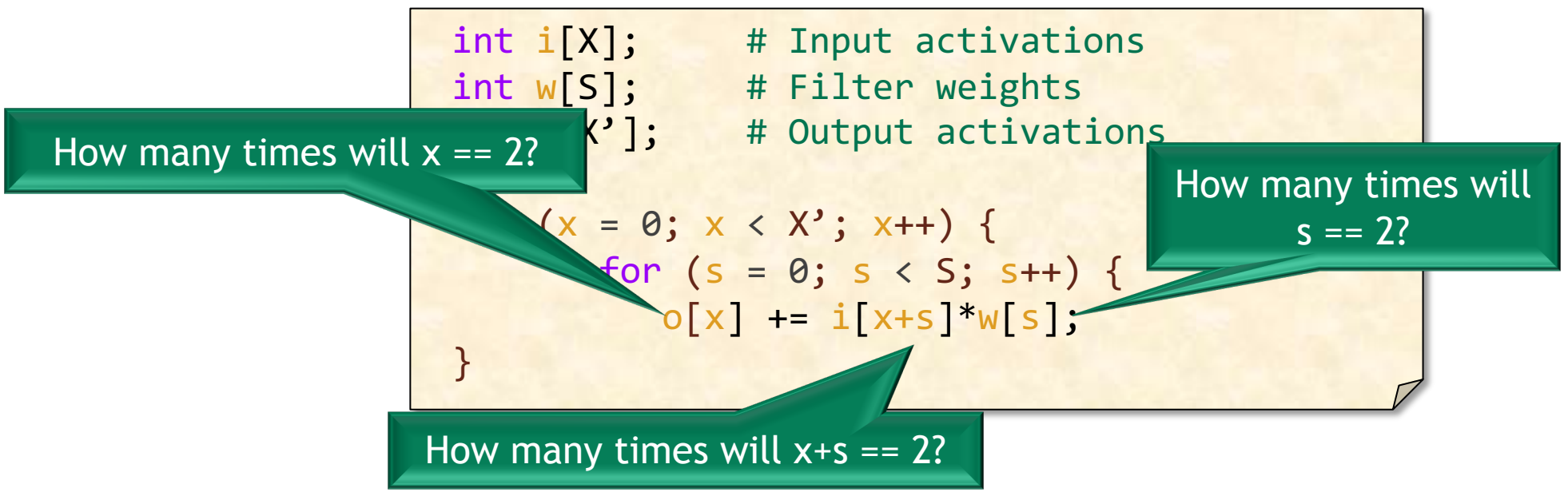


Imprecise analogy: think of data transfers as a wave with "amplitude" and "period"

- The stationary datatype has the **longest** period (locally held tile changes most slowly)
- Note: like waves, also can have harmful "constructive interference" (bursts)
- Later we will see how intermediate staging buffers reduce both bandwidth and energy

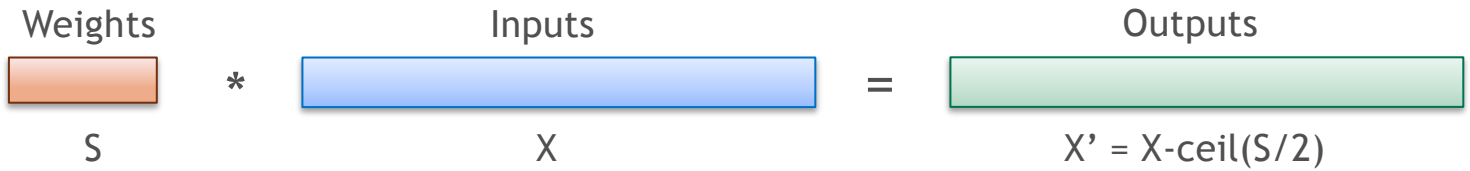
Often corresponds to datatype that is "done with" earliest without further reloads

“DONE WITH” VERSUS “NEEDS RELOAD”



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

FROM “LOOP NEST” TO DATAFLOW

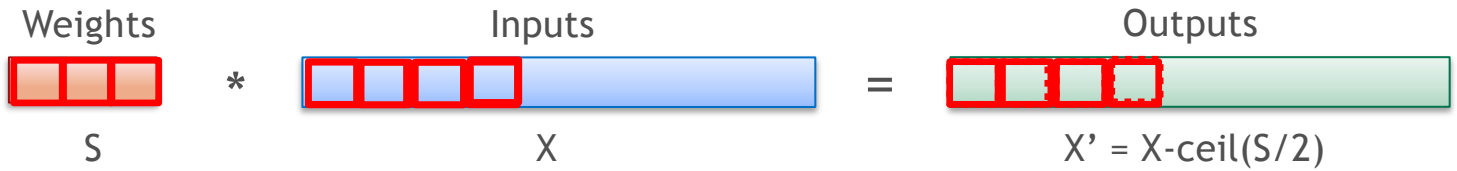



```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X’];     # Output activations

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

No constraints* on loop permutations!

* Because we don’t care about where precision/saturation issues occur – usually choose data sizes such that it never happens [See NVDLA’s 48-bit accumulators for 16-bit operands]

ANOTHER DATAFLOW

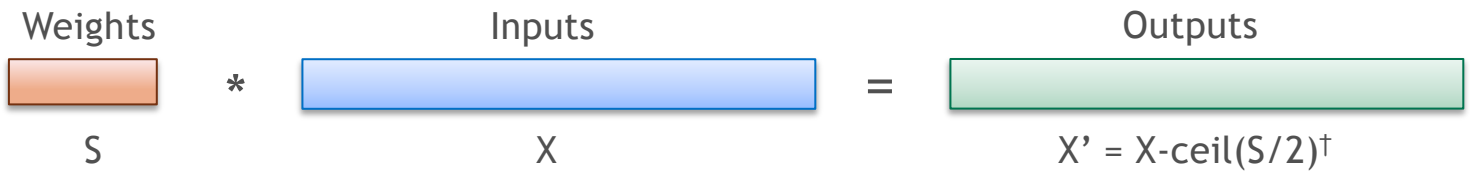



```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

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

What dataflow is this?

Weight stationary

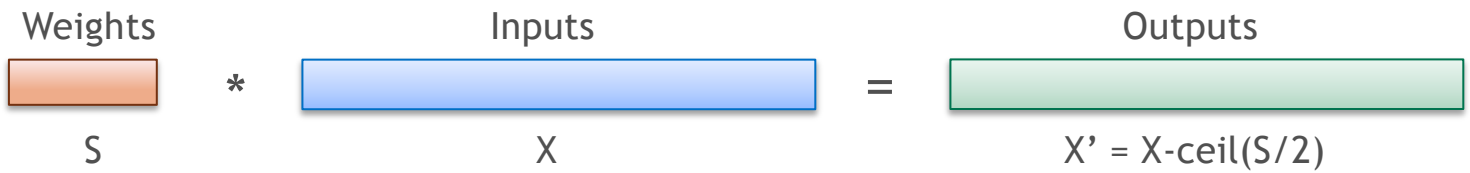
MORE DATAFLOWS




```
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 can we implement input stationary with no input index?

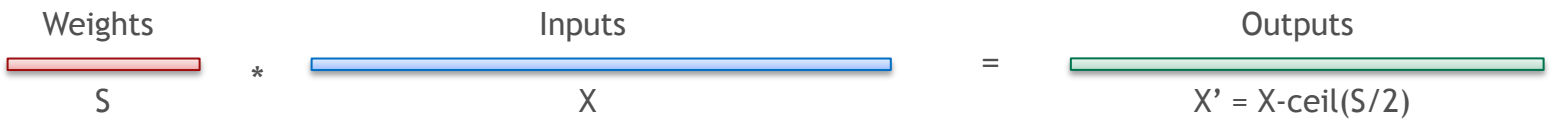
INPUT STATIONARY

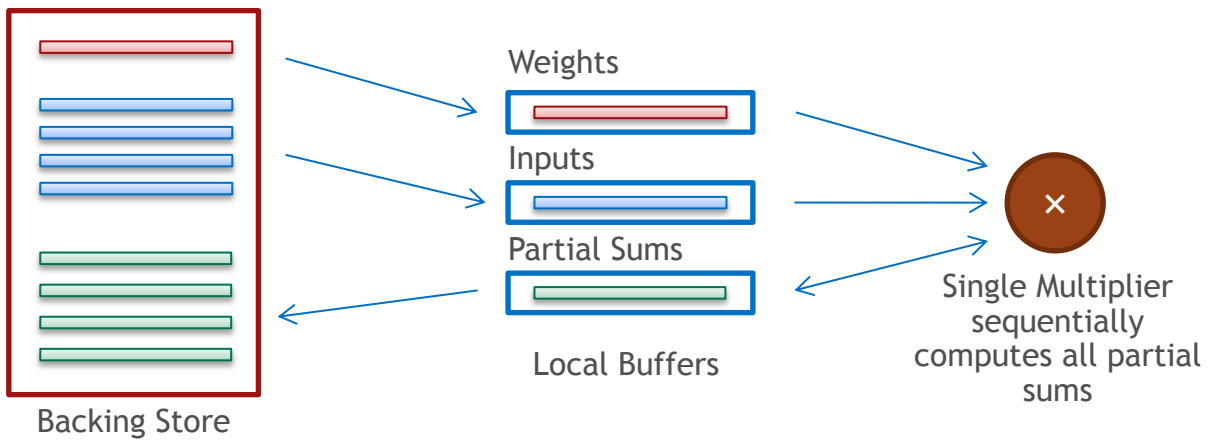



```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X']      # Output activations

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

Beware x-s must be >= 0 and <X'

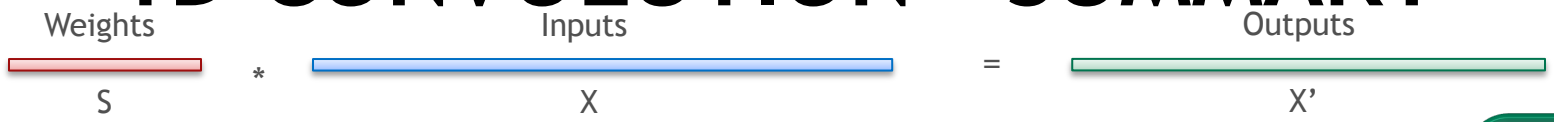
SIMPLE MODEL FOR MAPPING DATAFLOWS TO HARDWARE





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



Common metric	Weights	Inputs	Outputs / Partial Sums
Size = Alg. Min. accesses	S	X	X'
Maximum operand uses	SX'	SX'	SX'

Note: product always equals SX'

BUFSIZE-1D (Buffer size for zero re-fetch)

Dataflow	Weights	Inputs	Outputs
Weight-stationary	1	X'	X'
Input-stationary	S	1	S
Output-stationary	S	S	1

BUFMULT-1D (#times full buff[illegible]ccessed)

Dataflow	[illegible]ghts	Inputs	Outputs
Weight-st[illegible]ionary	SX'	S	S
Input-stationary	X'	SX'	X'
Output-stationary	X'	X'	SX'

Buffer access energy:

WS = SX'[f(1) + f(X') + 2f(X')]

IS = SX'[f(S) + f(1) + 2f(S)]

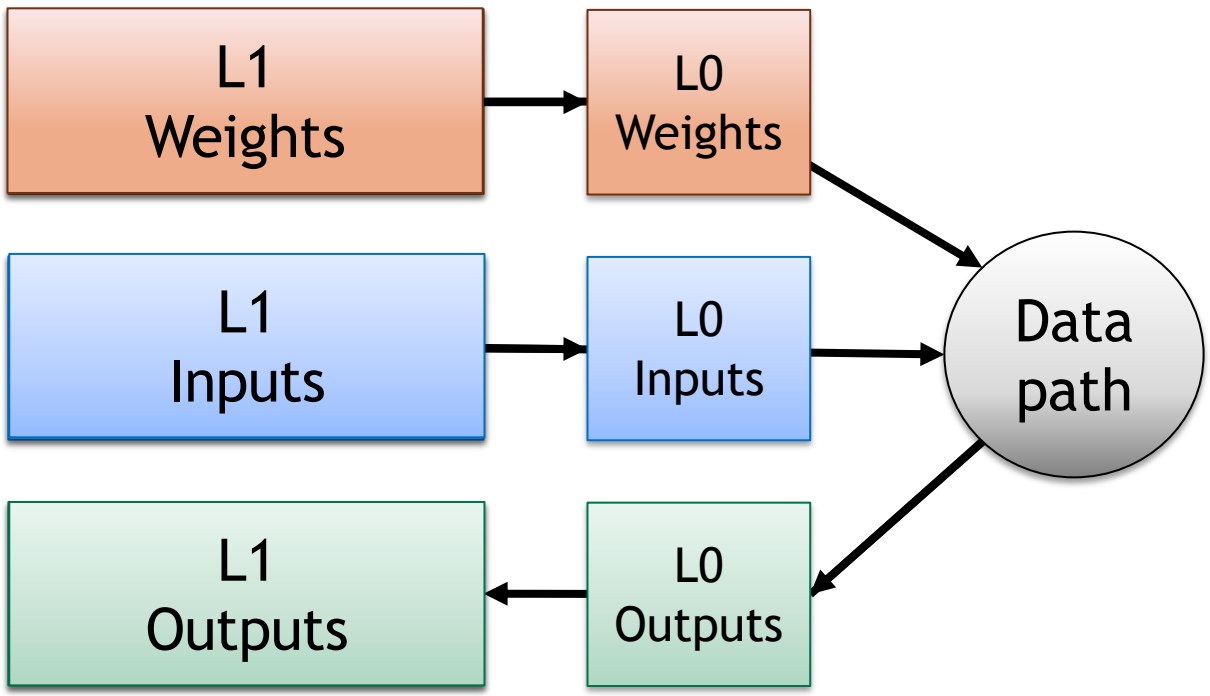
OS = SX'[f(S) + f(S) + 2f(1)]

f(x) = energy cost of accessing a RAM structure of size x

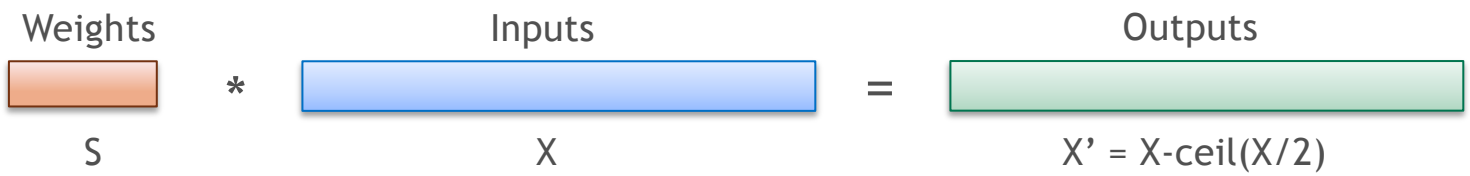
Significant difference in buffer access energy cost based on dataflow
But what if the provisioned buffering is smaller than required?

GETTING MORE REALISTIC

MULTI-LAYER BUFFERING



1-D CONVOLUTION - BUFFERED

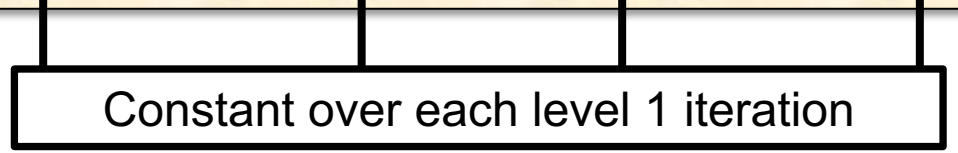



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// 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

ENERGY COSTS OF A MAPPING

```
// 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];
}
```



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

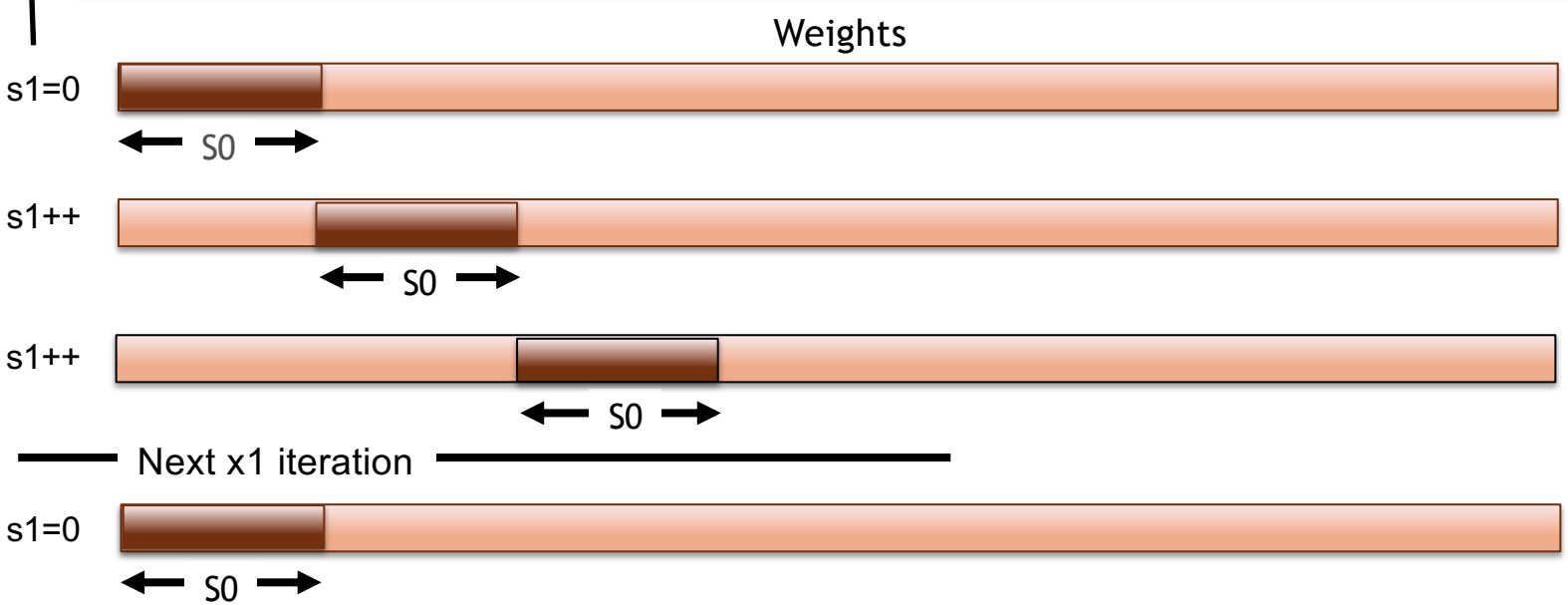
- For level 0 that is all the operands to the Datapath

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

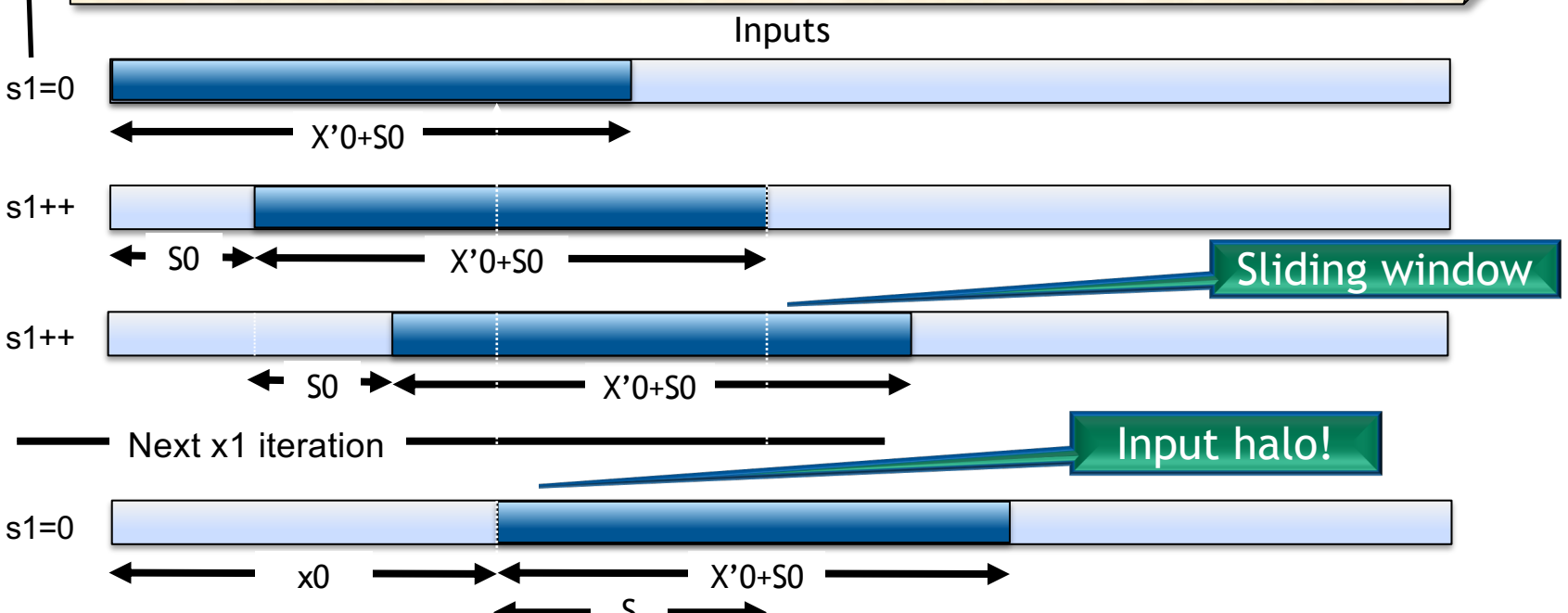
Times same number of level 1 iterations = X'1 * S1

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

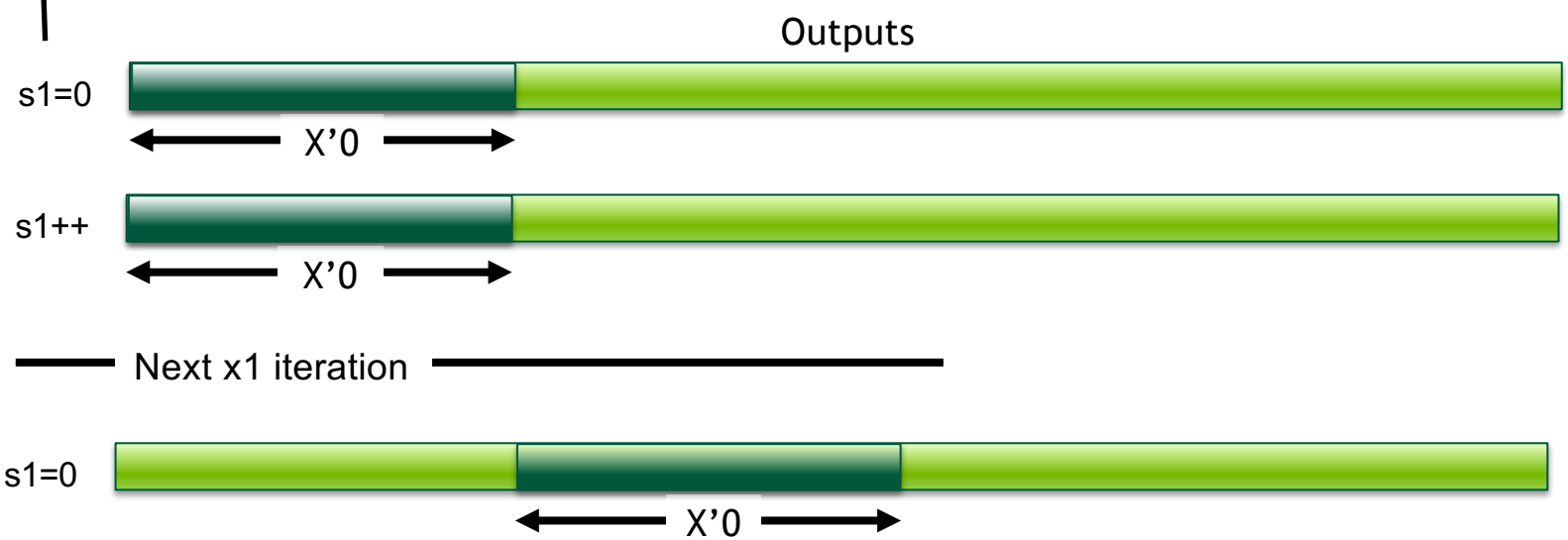
Times number of x1 iterations = X'1

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

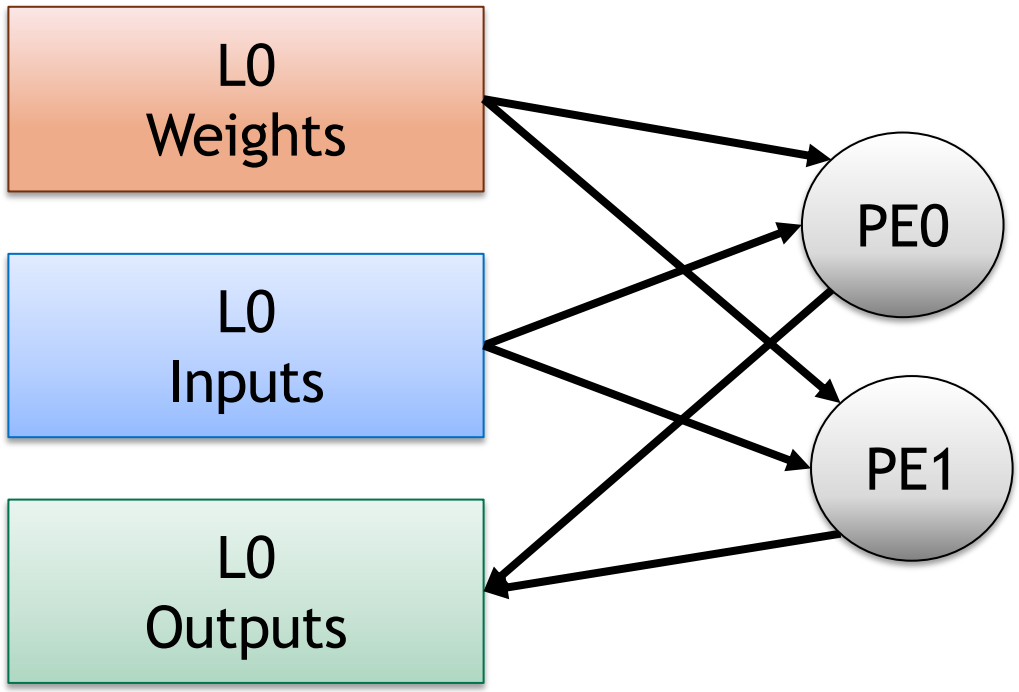
- After each S1 iterations a completed partial sum for X’0 outputs are transferred
- There are X’1 chunks of S1 iterations
- So total is X’1*X’0 = X’ transfers

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'

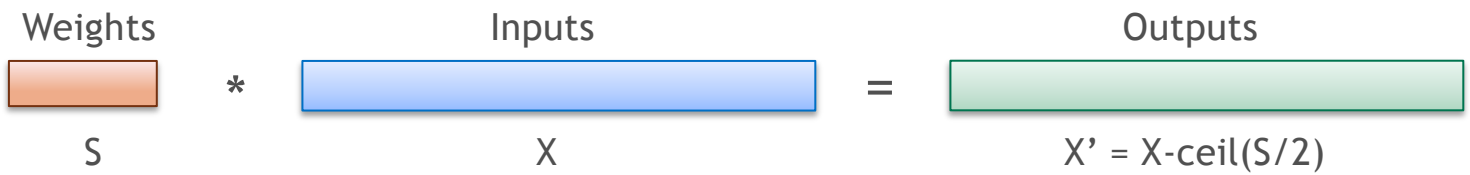
SPATIAL PES



How will this be reflected in the loop nest?

New ‘level’ of loops

1-D CONVOLUTION – SPATIAL



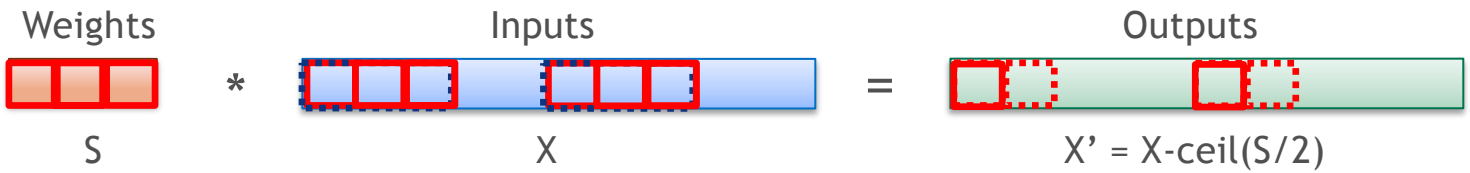

```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// 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

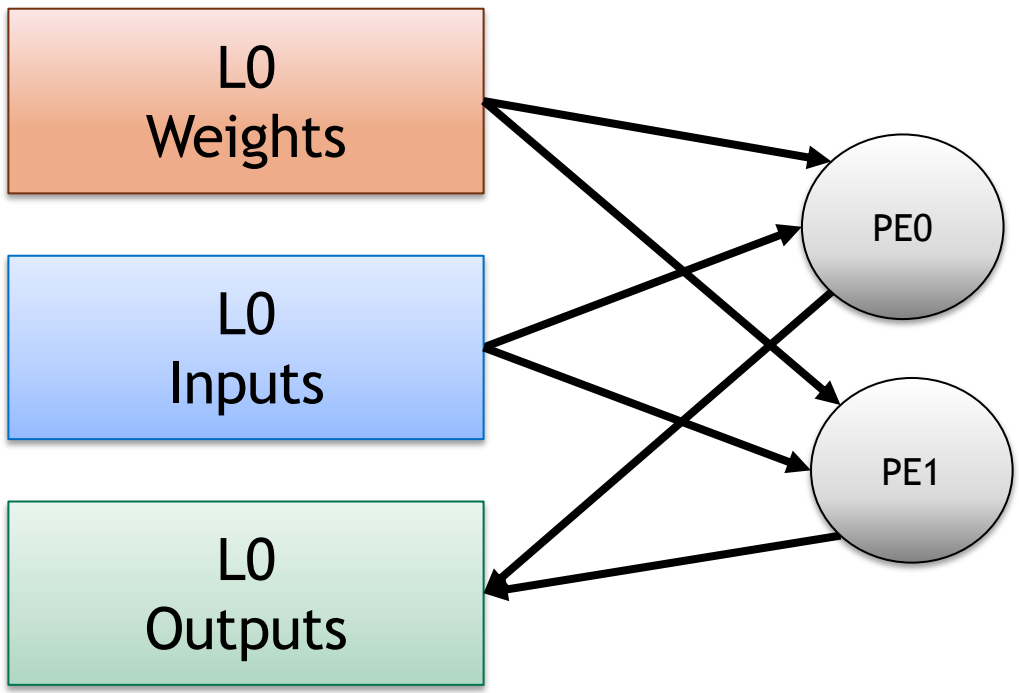
S1 = 1 => s1 = 0

1-D CONVOLUTION – SPATIAL




```
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
```
>>w[0]
```

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

Implementation opportunity? Yes, single fetch and multicast

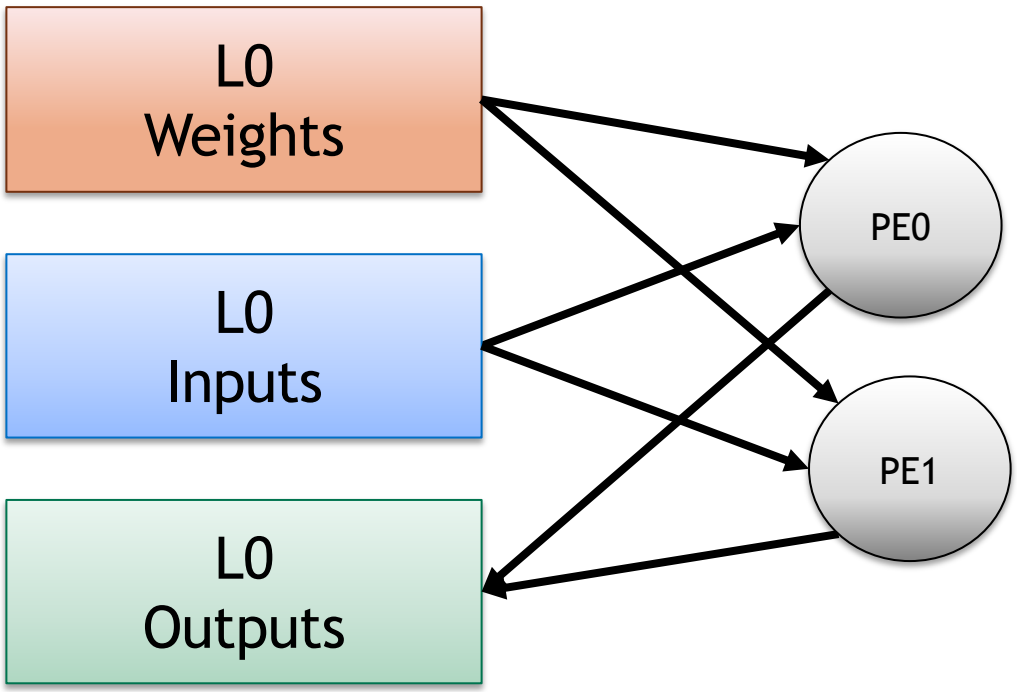
1-D CONVOLUTION – SPATIAL

```
// 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 INPUTS



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

SPATIAL PARTITIONING WEIGHTS


```
int i[W];      # Input activations
int w[R];      # Filter Weights
int o[E];      # 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 = 1 => x1 = 0

S0 = 1, S1 = 2

SPATIAL PARTITIONING 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

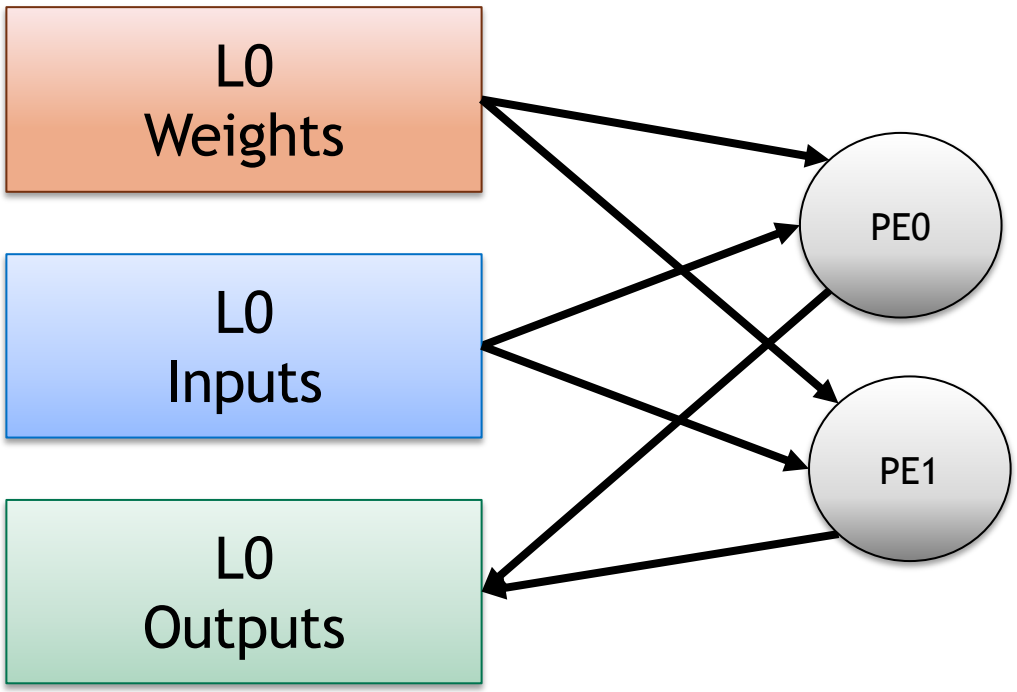
1-D CONVOLUTION – SPATIAL

```
// 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 summation…

Other multicast opportunities? — No

SPATIAL PES: PARTITIONED WEIGHTS

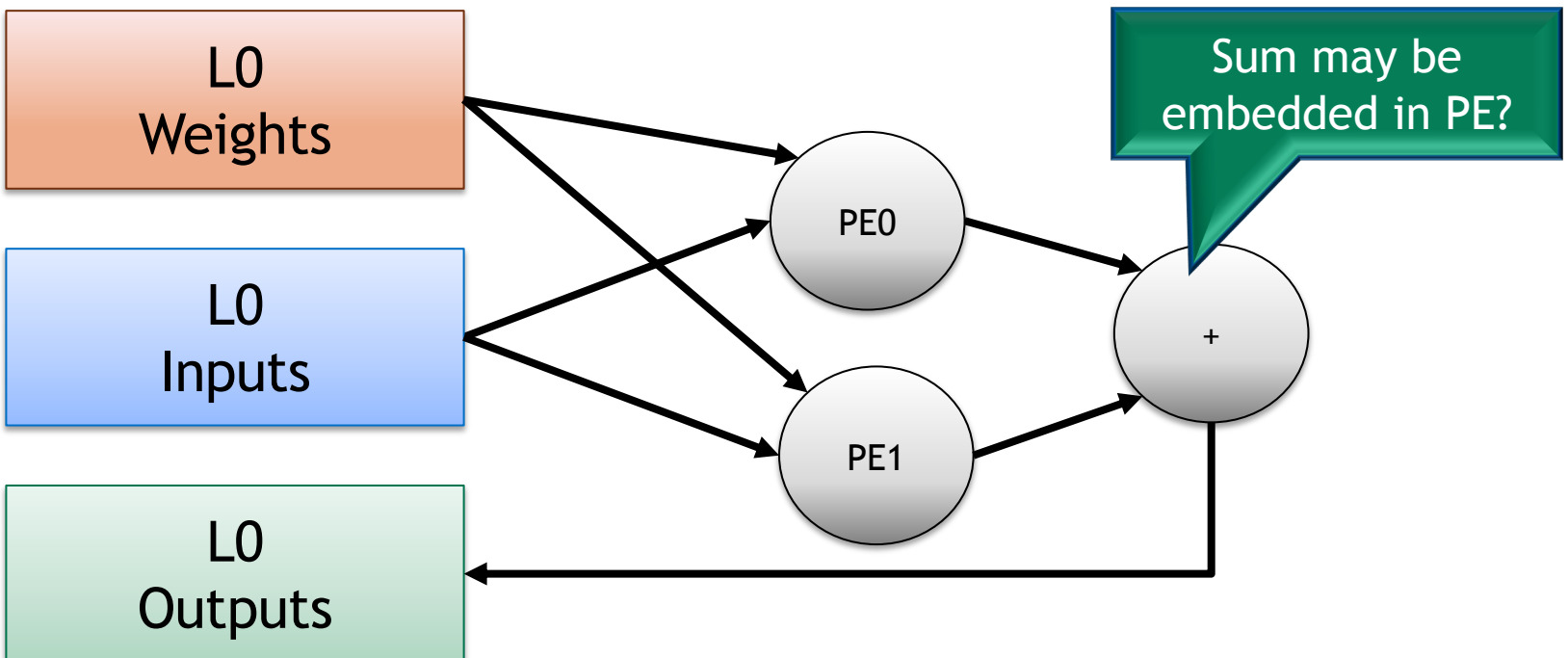



```
PE0                PE1
>>w[0]             >>w[S0+0]
>>i[0]             >>i[S0+0]
>>w[1]             >>w[S0+1]
>>i[1]             >>i[S0+1]
>>w[2]             >>w[S0+2]
>>i[2]             >>i[S0+2]
<<o[0]             <<o[0]

>>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?

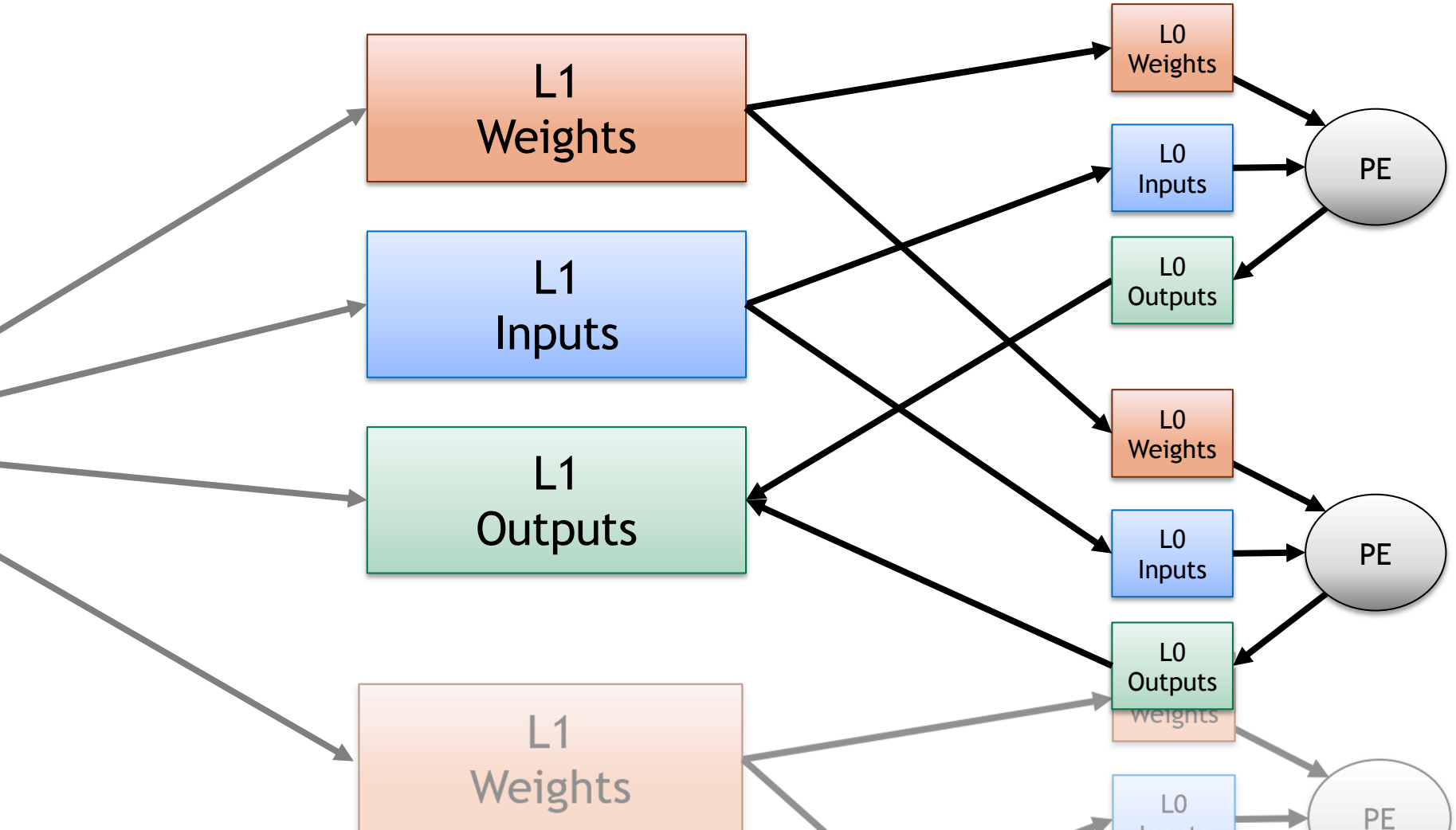
Illegal mapping!

MORE REALISTIC LOOP NEST

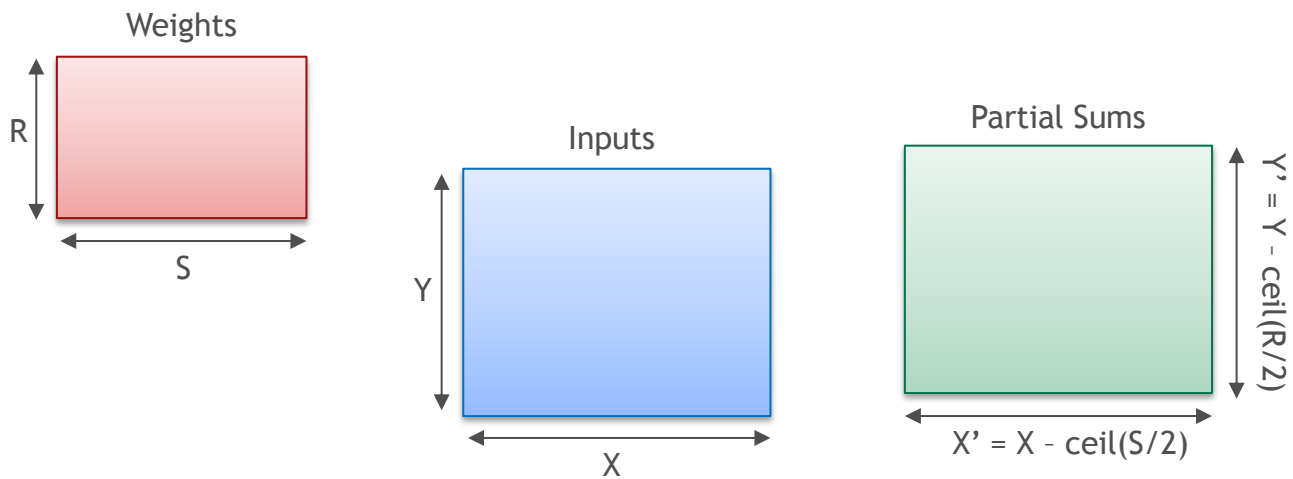
```
int i[W];      # Input activations
int w[R];      # Filter Weights
int o[E];      # Output activations

// 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++) {
              ...
```

General approach: alternate temporal/spatial levels and choose values (including 1) carefully

“FRACTAL” ACCELERATOR DESIGN



MORE REALISTIC PROBLEM: 2D CONVOLUTION



2-D CONVOLUTION LOOP NEST

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

for (y = 0; y < Y'; y++) {
  for (x = 0; x < X'; x++) {
     for (r = 0; r < R; r++) {
       for (s = 0; s < S; s++) {
          o[y][x] += i[y+r][x+s]*w[r][s];
}
```

What dataflow is this?

Output stationary + row major

What new opportunities can we exploit?

2D allows exploration of rectangular data tile aspect ratios

TILED 2-D CONVOLUTION

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

for (y2 = 0; y2 < Y2'; y2++) {
  for (x2 = 0; x2 < X2'; x2++) {
     for (r2 = 0; r2 < R2; r2++) {
       for (s2 = 0; s2 < S2; s2++) {
         parallel-for (y1 = 0; y1 < Y1; y1++) {
           ...
```

Note:
X'0*X'1*X'2 = X'
and so on...

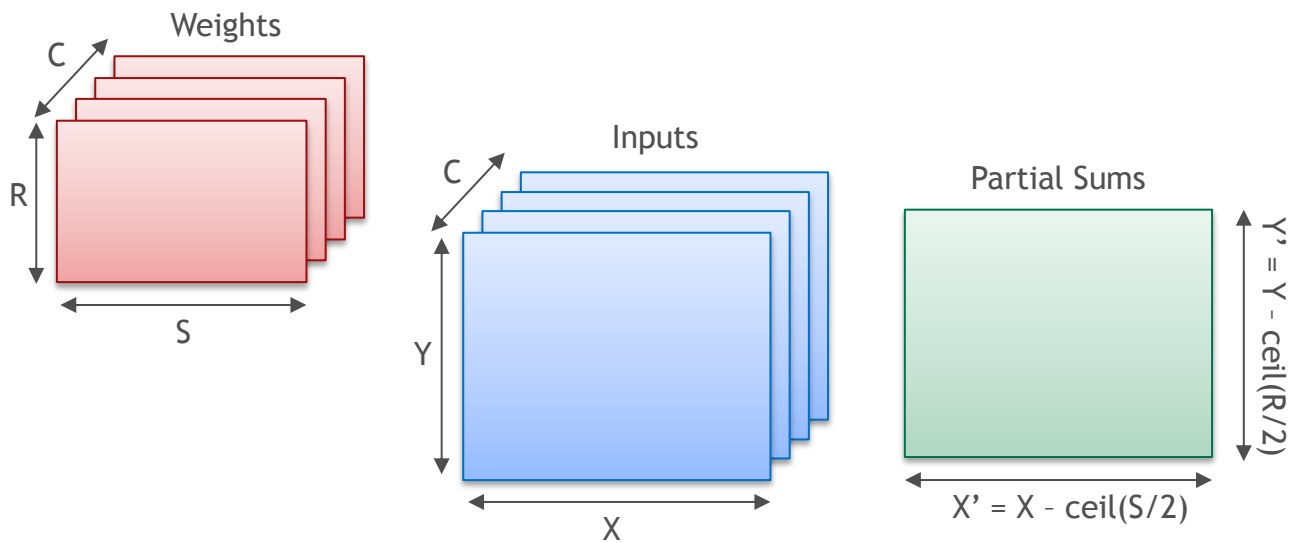
How do you make a square tile?

Set X0=Y0, and/or R0=S0

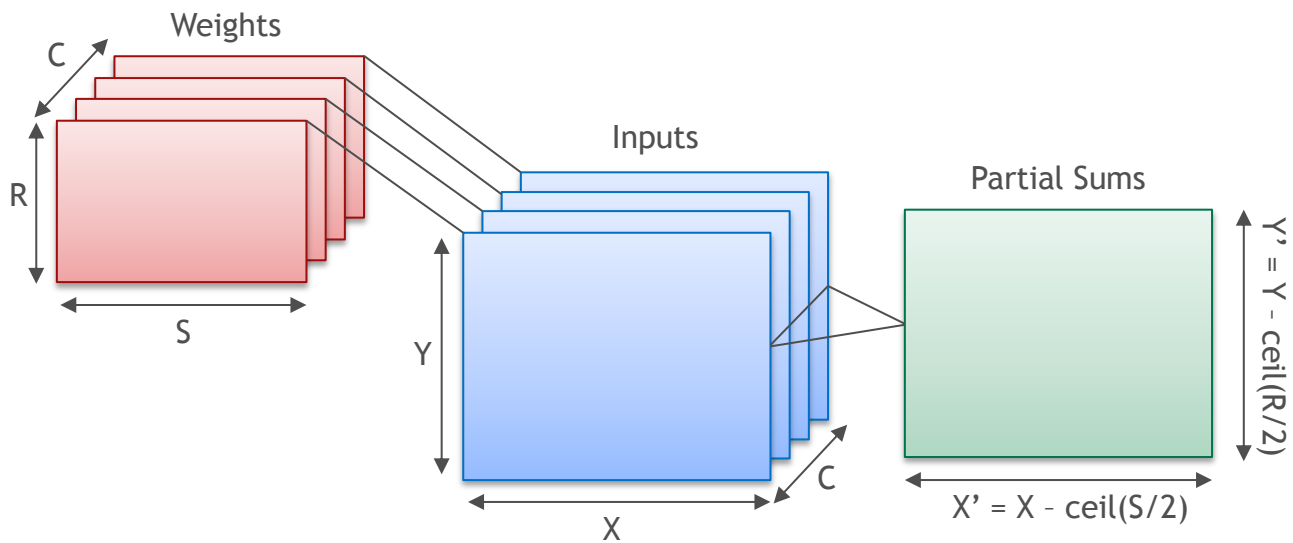
How do you make a 1D tile? (e.g., strip mining)

Set X0=width, Y0=1
or X0=1, Y0=height...

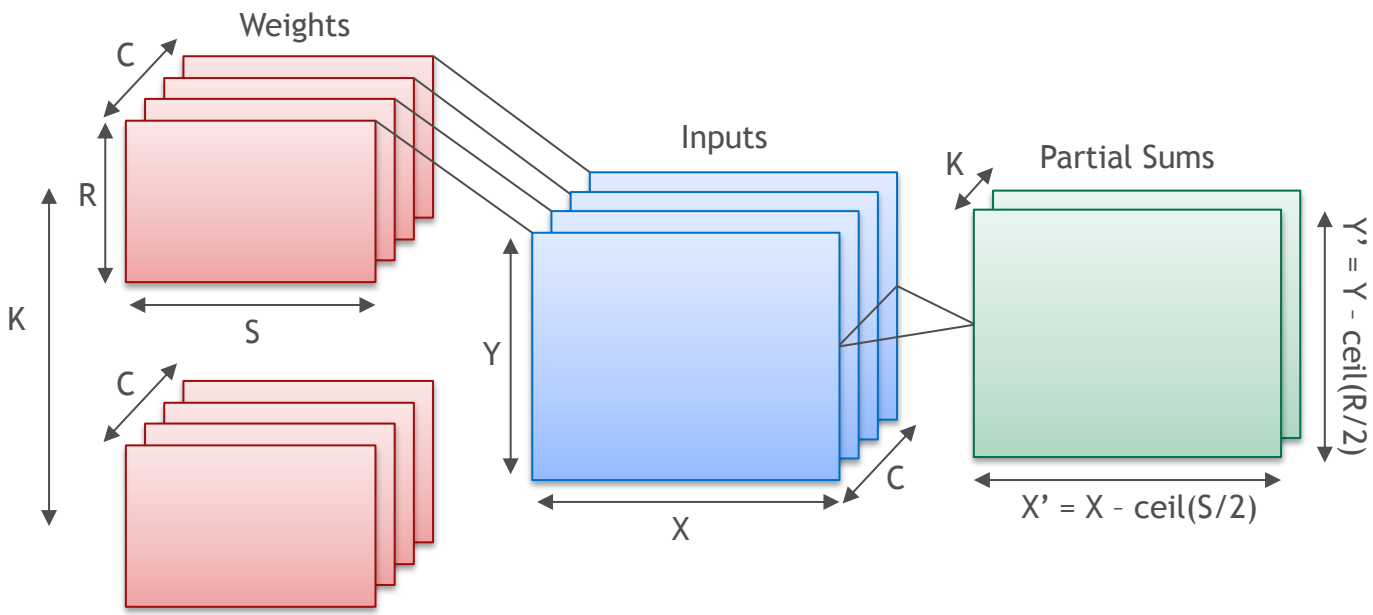
CNN PROBLEM FORMULATION - INPUT CHANNELS



CNN PROBLEM FORMULATION - INPUT CHANNELS



CNN PROBLEM FORMULATION - OUTPUT CHANNELS



REFERENCE CONVOLUTIONAL 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];
```

What dataflow is this?

Output-Channel Output Stationary, row major (input channel most minor)

What new opportunities can we exploit?

Each input contributes to many planes of outputs

TILING A CONVOLUTIONAL 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 (k1 = 0; k1 < K1; k1++) {
  for (y1 = 0; y1 < Y1'; y1++) {
    for (x1 = 0; x1 < X1'; x1++) {
      for (c1 = 0; c1 < C1; c1++) {
        for (r1 = 0; r1 < R1; r1++) {
          for (s1 = 0; s1 < S1; s1++) {
            parallel for (k0 = 0; k0 < K0; k0++) {
              ...
```

Gigantic space of potential loop orders and factorizations – how to explore?

- Cycle-accurate modeling of realistic dimensions and fabric sizes too slow
- Solution: use an analytic modeling

CONCLUSIONS

Understanding CNN dataflow and tiling concepts is critical for computer architecture researchers focusing on domain-specific accelerators

Caches are generally considered too expensive (area+power) but we can easily make up the difference using workload knowledge (more about this later today)

For an example of exploiting advanced dataflow knowledge, see:

UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition - Kartik Hegde (UIUC), Jiyong Yu (UIUC), Rohit Agrawal (UIUC), Mengjia Yan (UIUC), Michael Pellauer (NVIDIA), Christopher Fletcher (UIUC) [ISCA 2018]