Harvard John A. Paulson
School of Engineering
and Applied Sciences

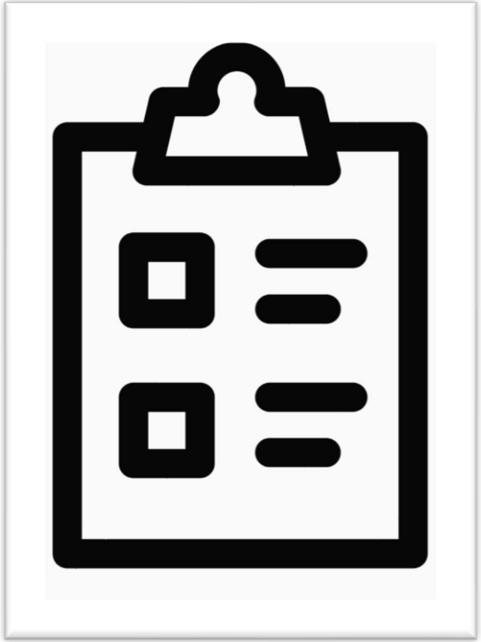
A New Method for Analyzing DNN Accelerator Fault Resilience

Jaylen Wang

(w/ the help of Abdulrahman)

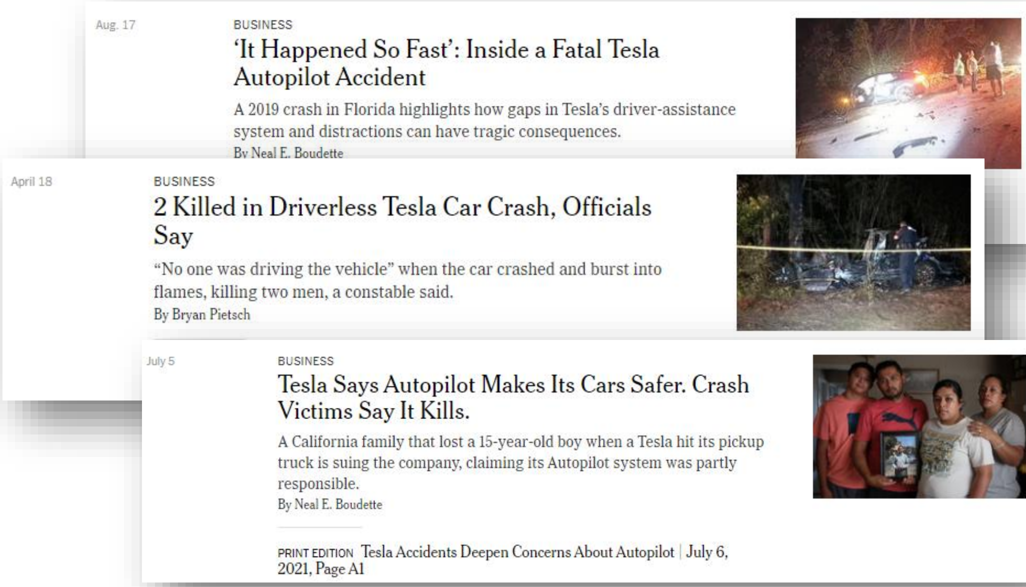
Agenda

- ❑ Motivation
- ❑ Previous Work
- ❑ Project Goal
- ❑ Background
- ❑ Framework
- ❑ Early Results
- ❑ Future Work



Motivation

- DNNs are ubiquitous and used in safety critical settings



Aug. 17

BUSINESS

‘It Happened So Fast’: Inside a Fatal Tesla Autopilot Accident

A 2019 crash in Florida highlights how gaps in Tesla’s driver-assistance system and distractions can have tragic consequences.

By Neal E. Boudette

April 18

BUSINESS

2 Killed in Driverless Tesla Car Crash, Officials Say

“No one was driving the vehicle” when the car crashed and burst into flames, killing two men, a constable said.

By Bryan Pietsch

July 5

BUSINESS

Tesla Says Autopilot Makes Its Cars Safer. Crash Victims Say It Kills.

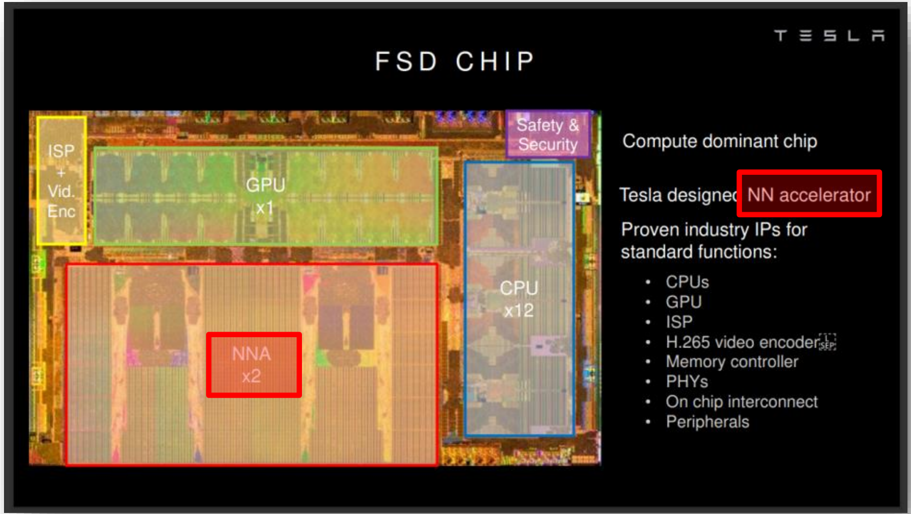
A California family that lost a 15-year-old boy when a Tesla hit its pickup truck is suing the company, claiming its Autopilot system was partly responsible.

By Neal E. Boudette

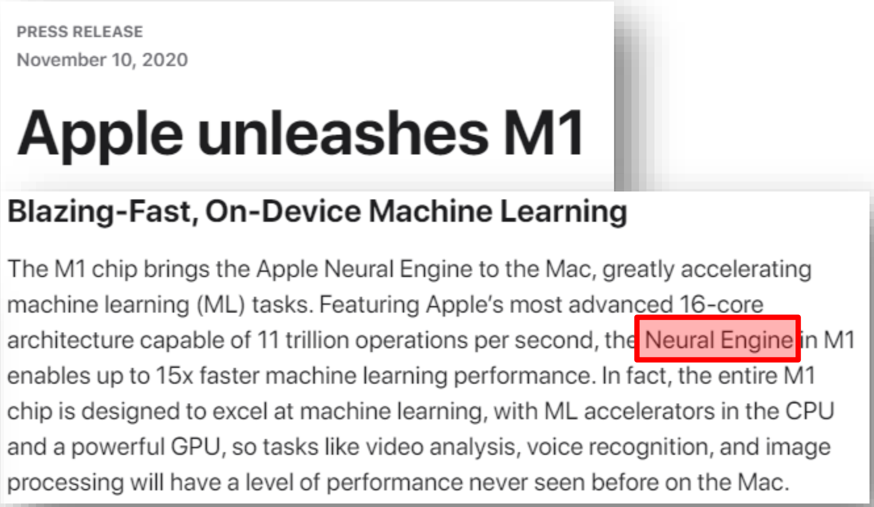
PRINT EDITION Tesla Accidents Deepen Concerns About Autopilot | July 6, 2021, Page A1

Motivation

- DNNs are ubiquitous and used in safety critical settings
- They are also increasingly being run on specialized accelerators



Source



PRESS RELEASE
November 10, 2020

Apple unleashes M1

Blazing-Fast, On-Device Machine Learning

The M1 chip brings the Apple Neural Engine to the Mac, greatly accelerating machine learning (ML) tasks. Featuring Apple's most advanced 16-core architecture capable of 11 trillion operations per second, the Neural Engine in M1 enables up to 15x faster machine learning performance. In fact, the entire M1 chip is designed to excel at machine learning, with ML accelerators in the CPU and a powerful GPU, so tasks like video analysis, voice recognition, and image processing will have a level of performance never seen before on the Mac.

Source

Harvard John A. Paulson School of Engineering and Applied Sciences

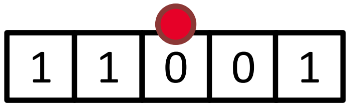
Motivation

- DNNs are ubiquitous and used in safety critical settings
- They are also increasingly being run on specialized accelerators
- Soft errors pose a problem for DNN systems running on such architectures

1	1	0	0	1

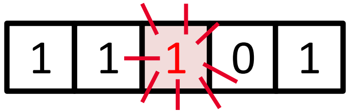
Motivation

- DNNs are ubiquitous and used in safety critical settings
- They are also increasingly being run on specialized accelerators
- Soft errors pose a problem for DNN systems running on such architectures



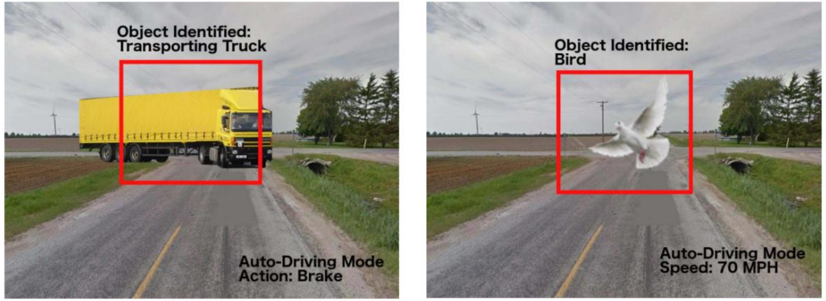
Motivation

- DNNs are ubiquitous and used in safety critical settings
- They are also increasingly being run on specialized accelerators
- Soft errors pose a problem for DNN systems running on such architectures



Motivation

- DNNs are ubiquitous and used in safety critical settings
- They are also increasingly being run on specialized accelerators
- Soft errors pose a problem for DNN systems running on such architectures



(a) Fault-free execution: A truck is identified by the DNN and brakes are applied

(b) SDC: Truck is incorrectly identified as bird and brakes may be not applied

Figure 2: Example of SDC that could lead to collision in self-driving cars due to soft errors

G. Li *et al. (2017)*

Motivation

- DNNs are ubiquitous and used in safety critical settings
- They are also increasingly being run on specialized accelerators
- Soft errors pose a problem for DNN systems running on such architectures

Important to understand resilience in such systems!

Previous Work

- FIdelity, a recent MICRO paper, looked at how to model datapath FF errors at the software level
    - Since hardware errors can only come up as errors at software-level outputs
- 2017 paper from Li is really the only one to look at buffer faults for accelerators, but methodology is quite opaque

	Targets DNNs	HW Aware	Generalizable	Fast
FIdelity	✓	✓	✗	✗
PyTorchFI	✓	✗	✓	✓
TensorFI	✓	✗	✓	✓
Ares	✓	✗	✓	✓

Let’s Fill in the Gaps: Project Goals

1. Create a more automated and generalizable method for modeling both memory and datapath errors in DNN accelerators.
2. More thorough analysis and case studies of accelerator dataflows and mappings as well as model architectures/layer shapes.
3. Use some SotA software error mitigation techniques along with injecting errors.

CNN Loop Nest

```
for r=[0:R):
 for s=[0:S):
  for p=[0:P):
   for q=[0:Q):
    for c=[0:C):
     for k=[0:K):
      for n=[0:N):
       Output[p][q][k][n] +=
         Weight[r][s][k][c] * Input[p+r][q+s][c][n];
```

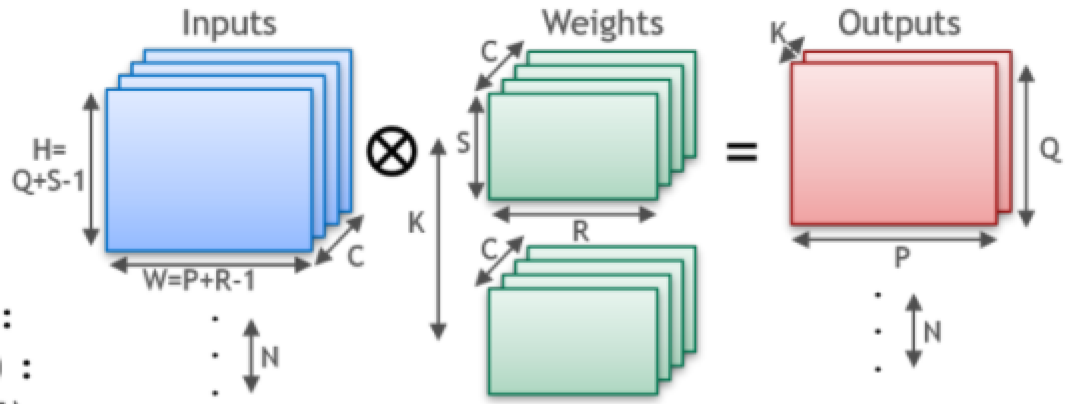


Fig. 3. Convolutional layer 7D loop nest.

Parashar *et al.* (2019)

Dataflow Loop Nest

- The 7 loop levels can be:
  - Reordered
  - Tiled
  - Spatially partitioned
- Some combination of these generates a dataflow

```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            … x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

(a) NVDLA Style Dataflow

Kwon *et al*. (2021)

DNN Error Propagation

Input

i1	i2	i3	i4
i5	i6	i7	i8
i9	i10	i11	i12
i13	i14	i15	i16

Weights

w1	w2
w3	w4

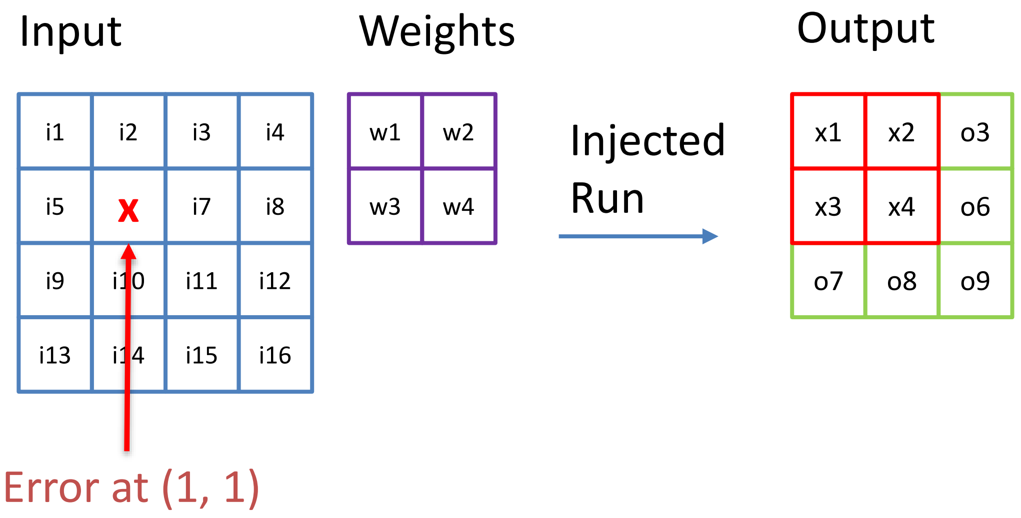
Injected Run →

Output

o1	o2	o3
o3	o4	o6
o7	o8	o9

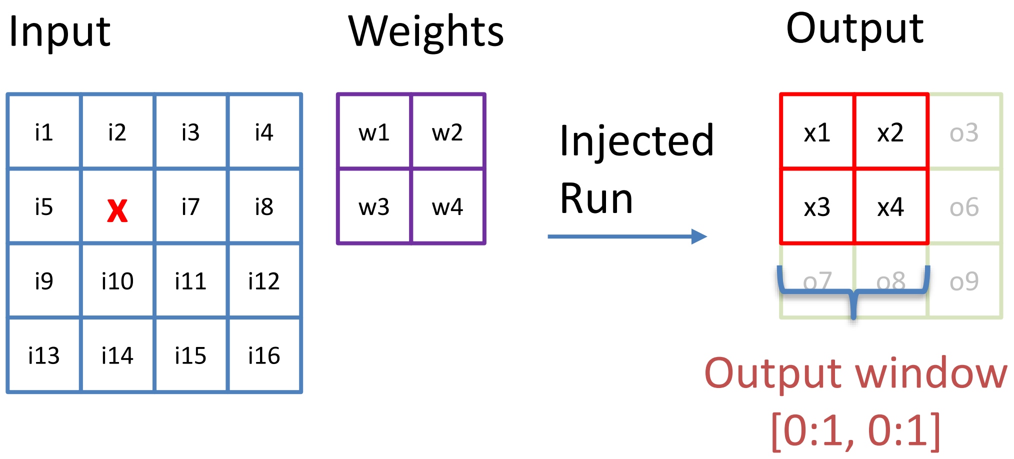
A normal software injection will produce errors at all locations at the output that use the value during MACs.

DNN Error Propagation



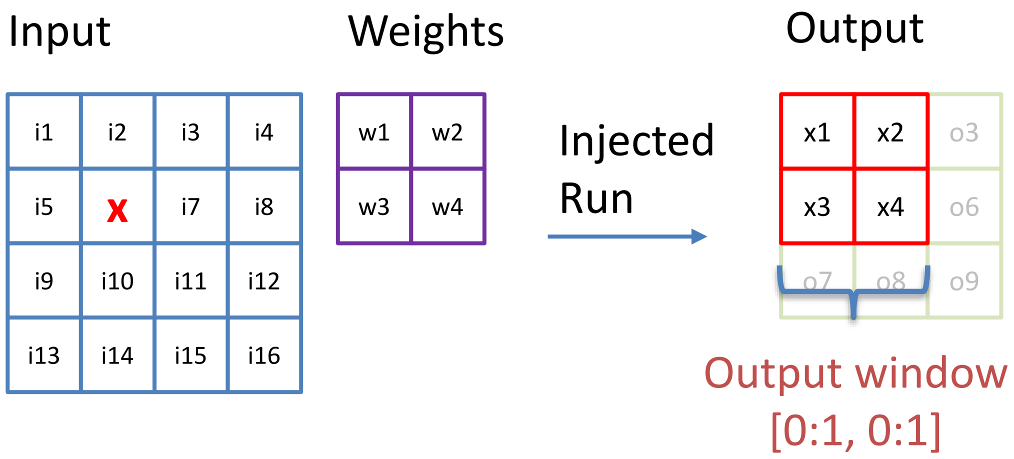
A normal software injection will produce errors at all locations at the output that use the value during MACs.

DNN Error Propagation



A normal software injection will produce errors at all locations at the output that use the value during MACs.

DNN Error Propagation



Adjusted conclusions from FIdelity:

- For errors in hardware, the set of output error locations must be a subset (with the same values) of the output window
- The values of the hardware error subset are the same as those produced by software-level injection

DNN Error Propagation

DRAM

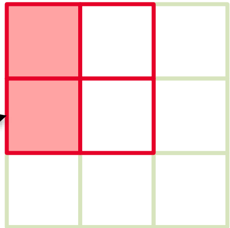
```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
```

CBUF

```
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            … x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

(a) NVDLA Style Dataflow

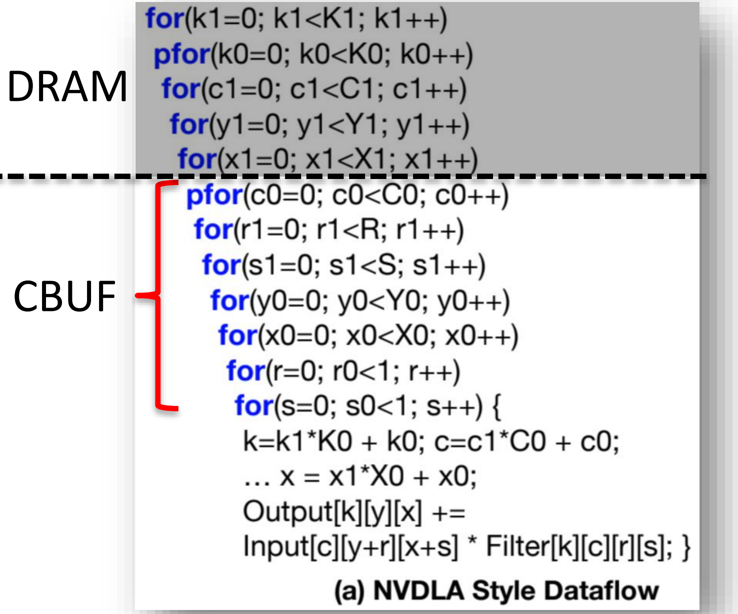
Dataflow reuse sites
(subset of the window)



Output window values

x1	x2	o3
x3	x4	o6
o7	o8	o9

DNN Error Propagation

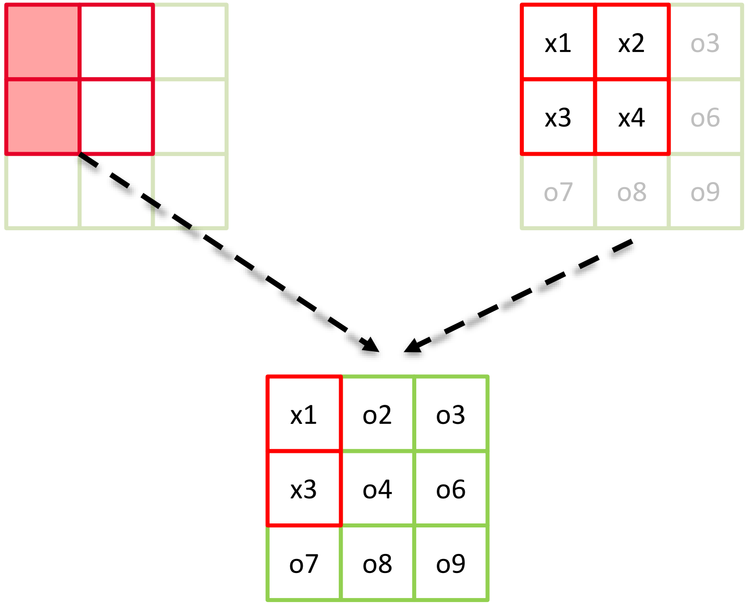



```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            … x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

(a) NVDLA Style Dataflow

Dataflow reuse sites (subset of the window)

Output window values



Final faulty output for a CBUF error

DNN Error Propagation

DRAM

```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
```

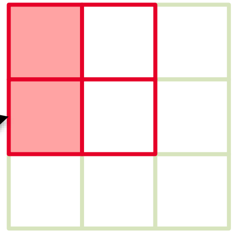
CBUF

```
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            … x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

(a) NVDLA Style Dataflow

?

Dataflow reuse sites
(subset of the window)



Error Propagation Modeling

- Literally ran a huge nested for loop taken directly from paper for NVDLA and tracked the propagation of an error and confirmed it produced the same error sites as reported by FIdelity

```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            ... x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

(a) NVDLA Style Dataflow

Error Propagation Modeling

- Literally ran a huge nested for loop taken directly from paper for NVDLA and tracked the propagation of an error and confirmed it produced the same error sites as reported by FIdelity

```
#----DRAM----#
for k1 in range(K1):
    for k0 in range(K0): # parallel
        for c1 in range(C1):
            for y1 in range(Y1):
                for x1 in range(X1):
                    #----- weight/input buffer -----#
                    for c0 in range(C0): # parallel
                        for s1 in range(S):
                            for r1 in range(R):
                                for y0 in range(Y0):
                                    for x0 in range(X0):
                                        #----- i_reg, o_reg, w_reg -----#
```

```
Output shape: (1, 32, 64, 64)

Affected output locations: (N, K, Y, X)
(0, 0, 3, 3)
(0, 1, 3, 3)
(0, 2, 3, 3)
(0, 3, 3, 3)
(0, 4, 3, 3)
(0, 5, 3, 3)
(0, 6, 3, 3)
(0, 7, 3, 3)
(0, 8, 3, 3)
(0, 9, 3, 3)
(0, 10, 3, 3)
(0, 11, 3, 3)
(0, 12, 3, 3)
(0, 13, 3, 3)
(0, 14, 3, 3)
(0, 15, 3, 3)
```

Generalized Loop Construction

- Given a mapping from Timeloop (or provided by the user):
  - Recursively construct the loop nest that describes the mapping of the workload
  - Simulate the error propagation at a specified location and memory level
- Verified with both NVDLA and Eyeriss

```
DRAM [ Weights:34848 Inputs:154587 Outputs:290400 ]
---------------------------------------------------
| for Q in [0:5)

shared_glb [ Inputs:34731 ]
---------------------------
|   for M in [0:6)
|     for Q in [0:11)
|       for P in [0:55)
|         for M in [0:16) (Spatial-Y)
|           for C in [0:3) (Spatial-X)

pe_spad [ Weights:121 ]
-----------------------
|             for S in [0:11)
|               for R in [0:11)

weight_reg [ Weights:1 ]
------------------------
|                 for Q in [0:1)

input_activation_reg [ Inputs:1 ]
---------------------------------
|                   for Q in [0:1)

output_activation_reg [ Outputs:1 ]
-----------------------------------
|                     for Q in [0:1)
```

DNN Error Propagation

DRAM

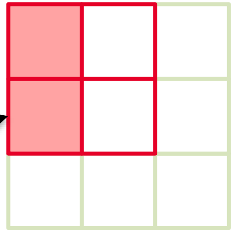
```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            … x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

CBUF

(a) NVDLA Style Dataflow

?

Dataflow reuse sites
(subset of the window)



DNN Error Propagation

DRAM

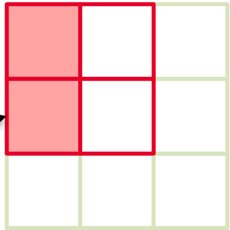
```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
```

CBUF

```
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            … x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

(a) NVDLA Style Dataflow

Dataflow reuse sites
(subset of the window)



DNN Error Propagation

DRAM

```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
```

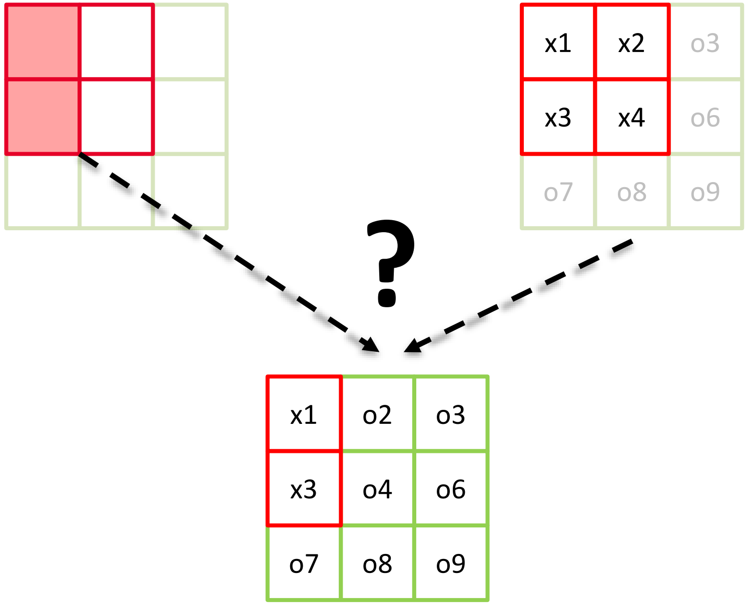
CBUF

```
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            … x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

(a) NVDLA Style Dataflow

Dataflow reuse sites (subset of the window)

Output window values

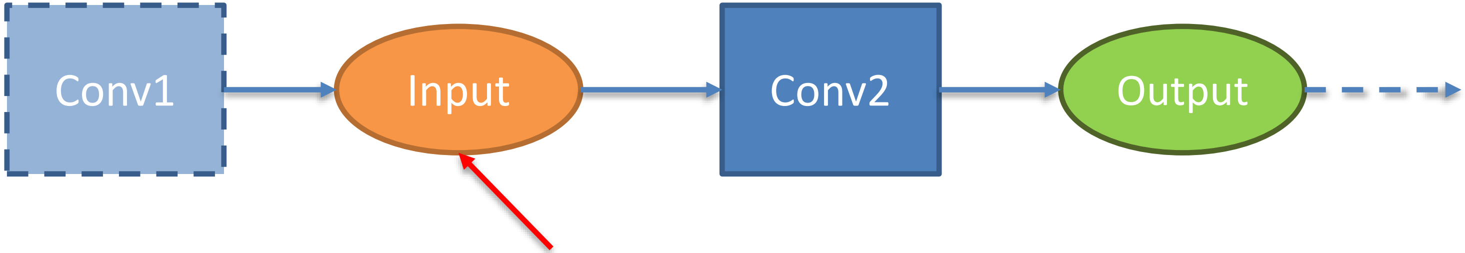


Final faulty output for a CBUF error

PyTorch Injection

PyTorch **hooks** make it *easy* and *fast* to perform various injections and operations:

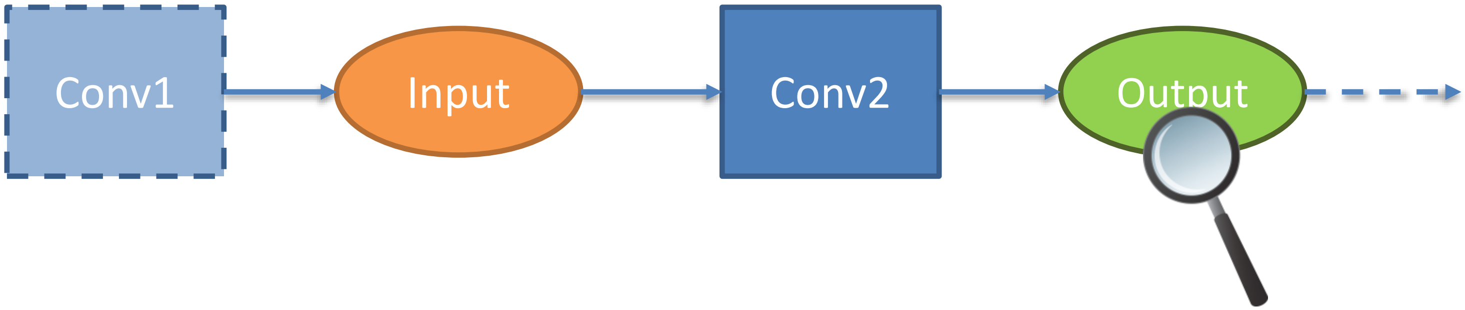
- **Can perform injections by changing inputs**



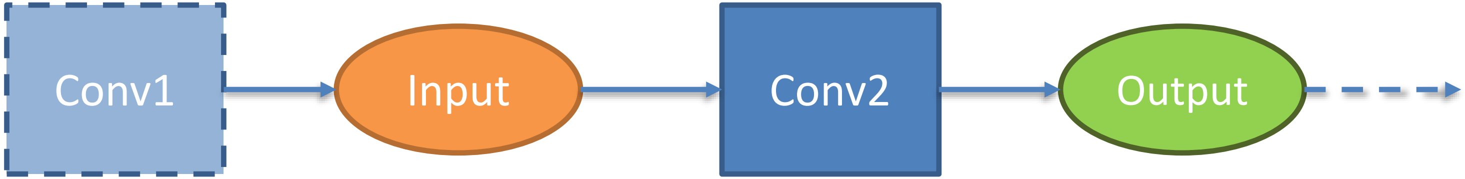
PyTorch Injection

PyTorch **hooks** make it *easy* and *fast* to perform various injections and operations:

- **Can perform injections by changing inputs**
- **Can selectively change error sites at the output**



```
net = alexnet(pretrained=True)
net(image)
```



Input matrix

i1	i2	i3	i4
i5	i6	i7	i8
i9	i10	i11	i12
i13	i14	i15	i16

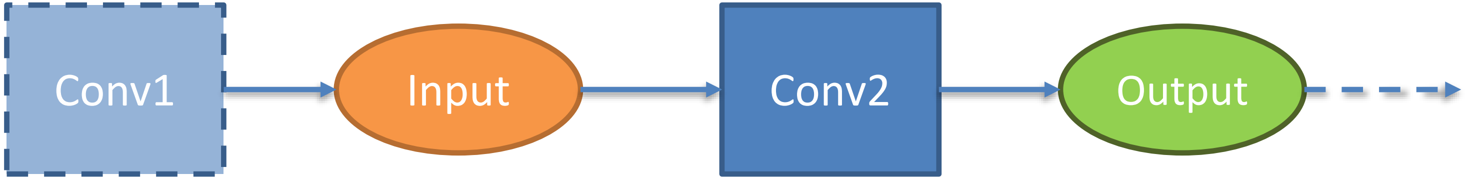
Weight matrix

w1	w2
w3	w4

Output matrix

o1	o2	o3
o3	o4	o6
o7	o8	o9

Normal Run



Input matrix

i1	i2	i3	i4
i5	i6	i7	i8
i9	i10	i11	i12
i13	i14	i15	i16

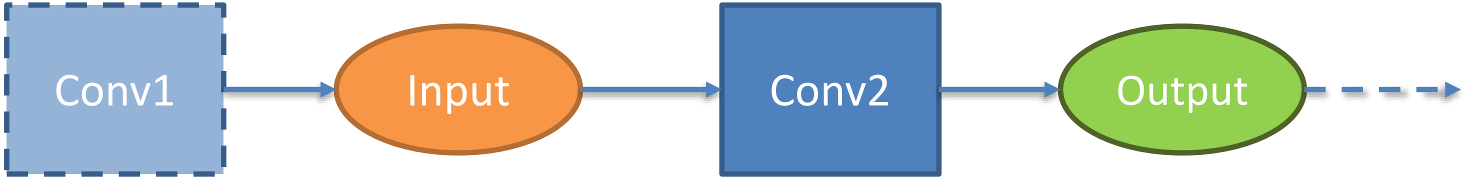
Weight matrix

w1	w2
w3	w4

Output matrix

o1	o2	o3
o3	o4	o6
o7	o8	o9

```
net.register_hook(Conv2, inject)
```



Input matrix

i1	i2	i3	i4
i5	i6	i7	i8
i9	i10	i11	i12
i13	i14	i15	i16

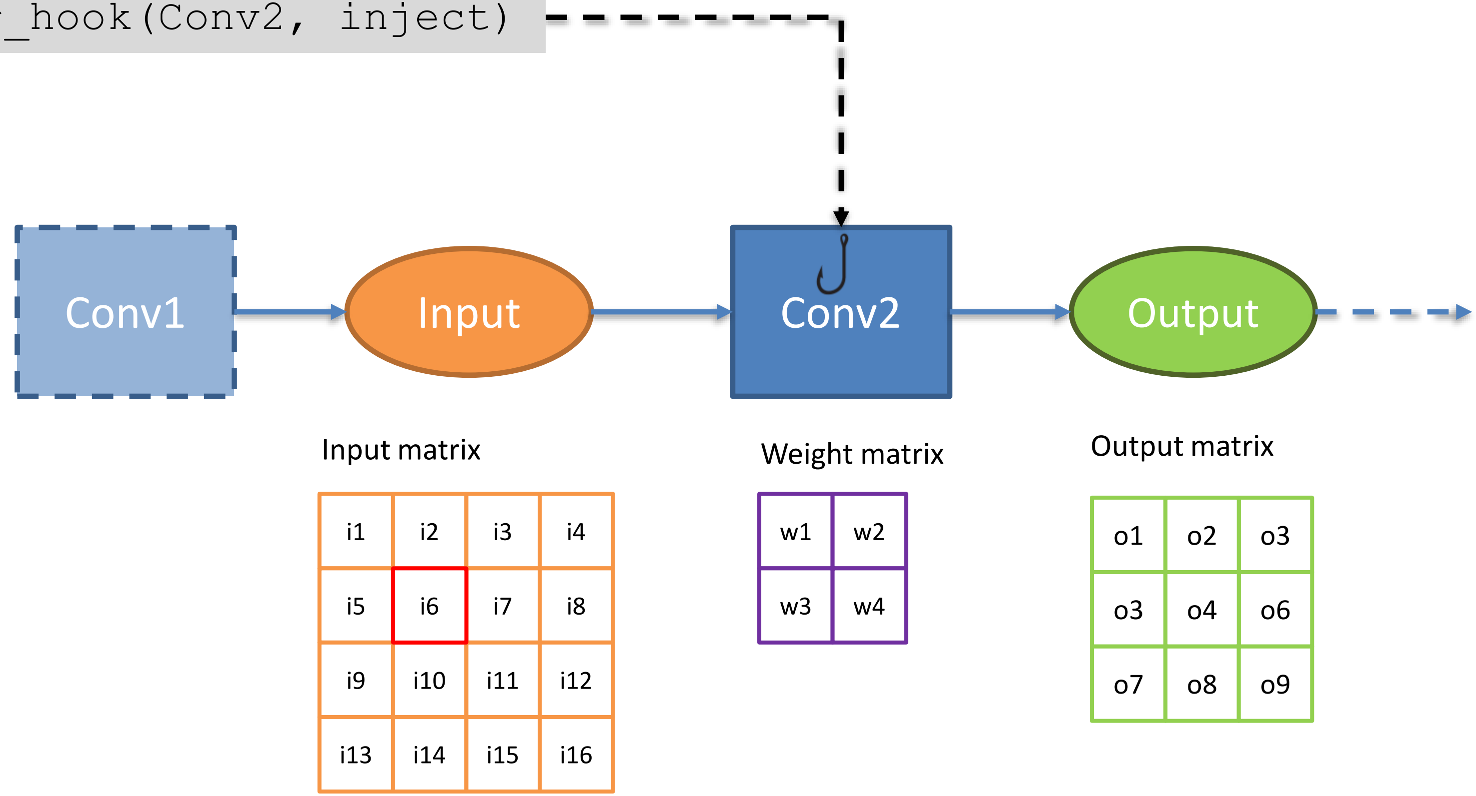
Weight matrix

w1	w2
w3	w4

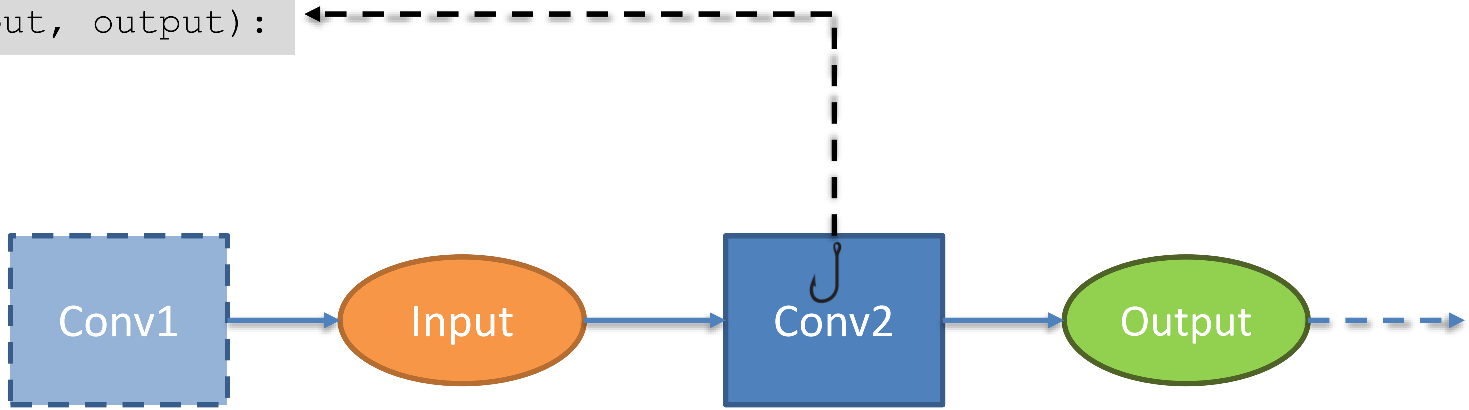
Output matrix

o1	o2	o3
o3	o4	o6
o7	o8	o9

```
net.register_hook(Conv2, inject)
```



```
def inject(input, output):
```



Input matrix

i1	i2	i3	i4
i5	i6	i7	i8
i9	i10	i11	i12
i13	i14	i15	i16

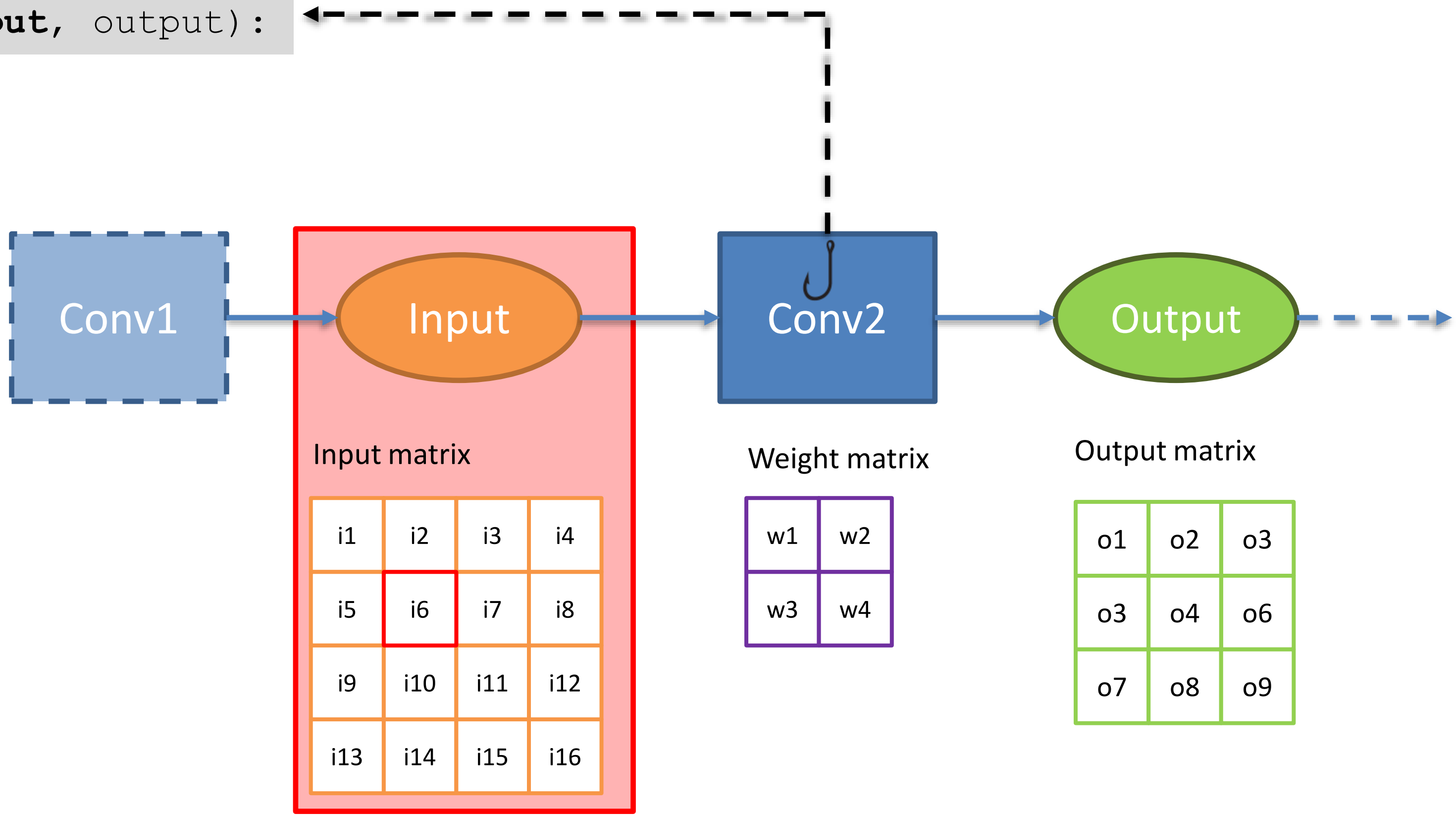
Weight matrix

w1	w2
w3	w4

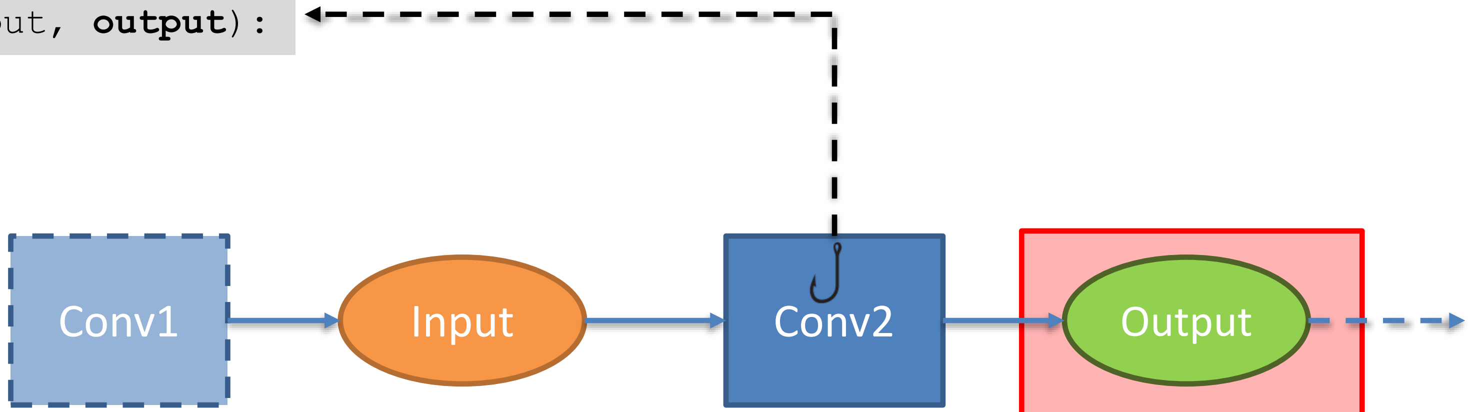
Output matrix

o1	o2	o3
o3	o4	o6
o7	o8	o9

```
def inject(input, output):
```



```
def inject(input, output):
```



Input matrix

i1	i2	i3	i4
i5	i6	i7	i8
i9	i10	i11	i12
i13	i14	i15	i16

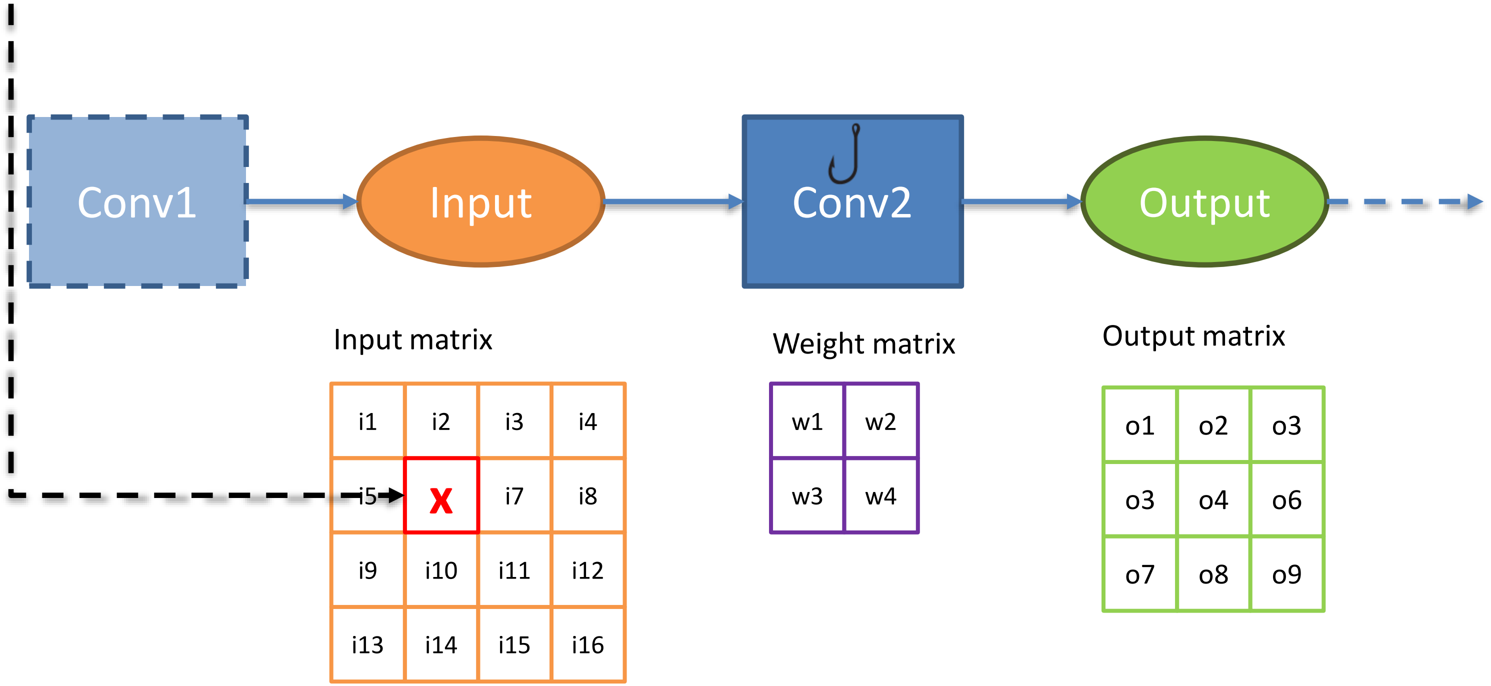
Weight matrix

w1	w2
w3	w4

Output matrix

o1	o2	o3
o3	o4	o6
o7	o8	o9

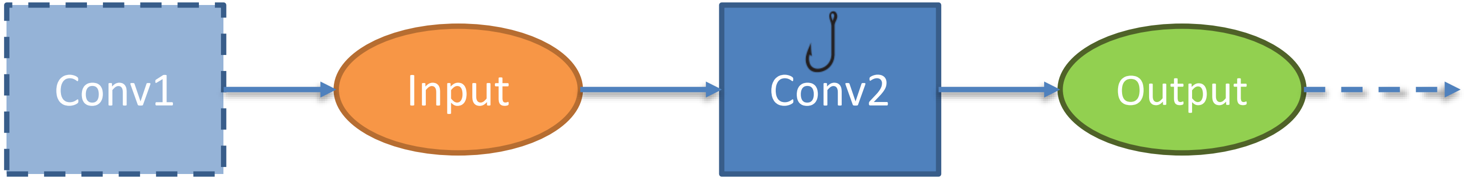
```
def inject(input, output):
  input[i6] = x
```



```
def inject(input, output):
  input[i6] = x
  faulty_output = Conv2(input)
```

Faulty output

x1	x2	o3
x3	x4	o6
o7	o8	o9



Input matrix

i1	i2	i3	i4
i5	**x**	i7	i8
i9	i10	i11	i12
i13	i14	i15	i16

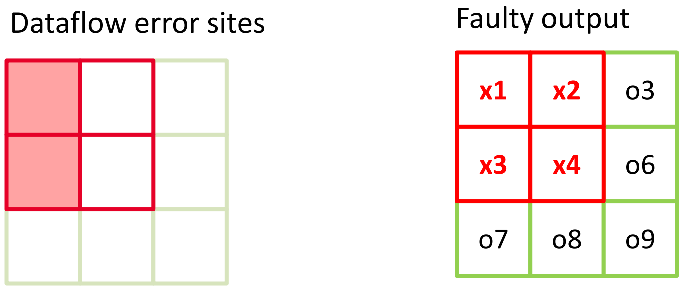
Weight matrix

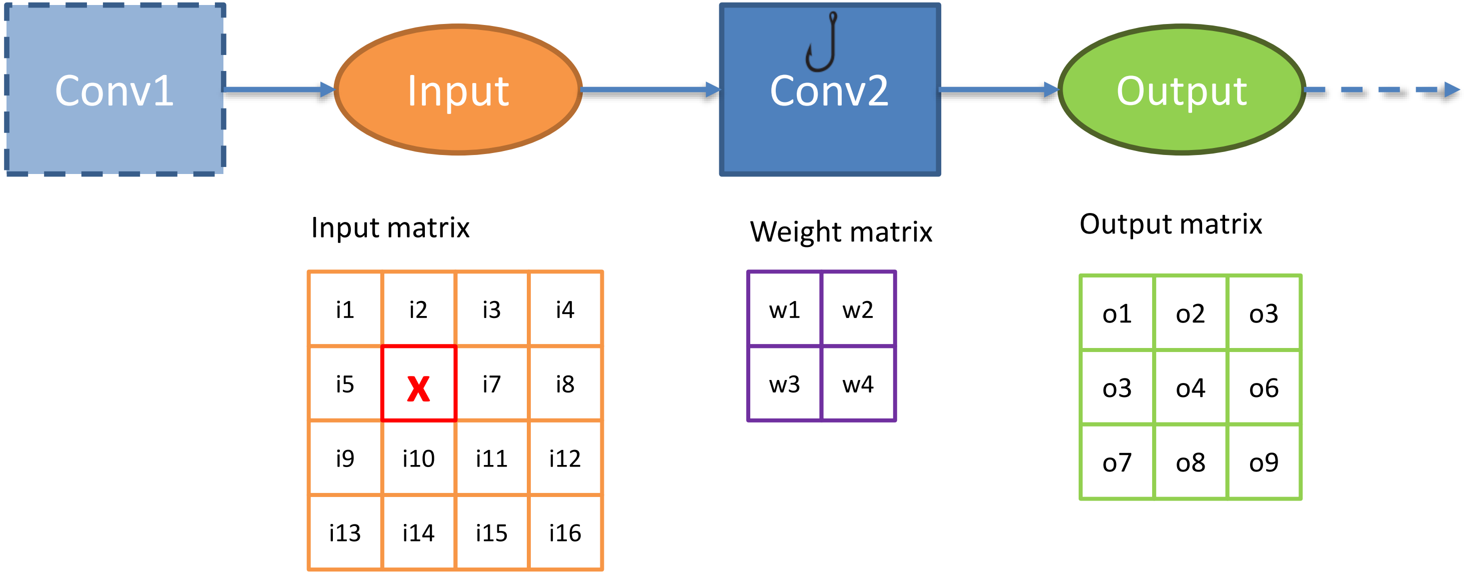
w1	w2
w3	w4

Output matrix

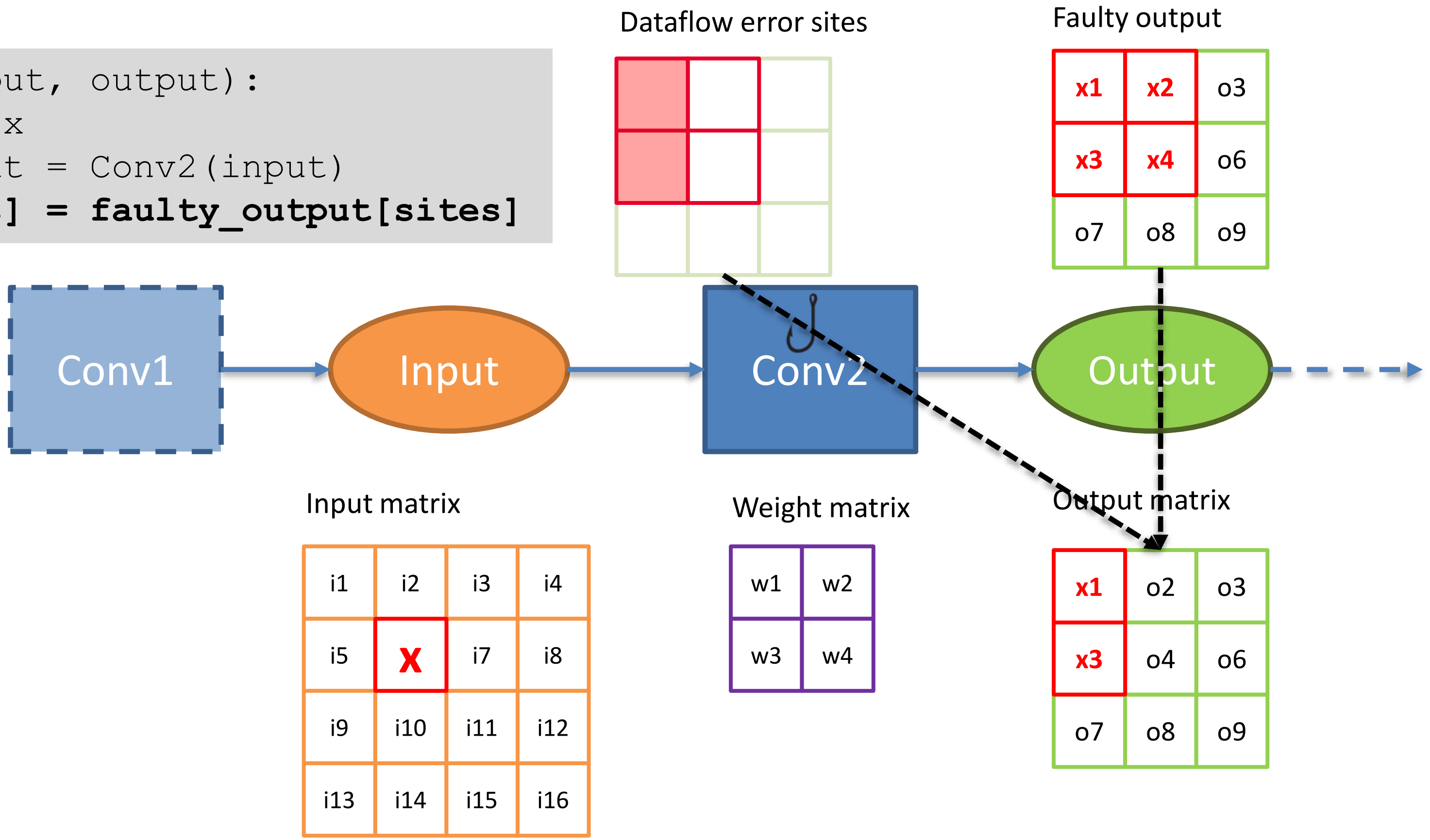
o1	o2	o3
o3	o4	o6
o7	o8	o9

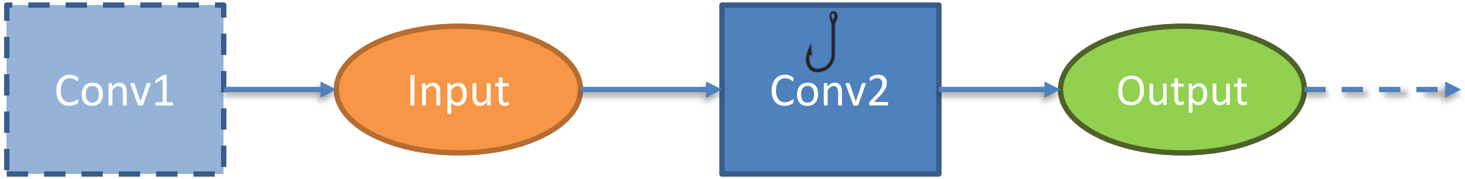
```
def inject(input, output):
  input[i6] = x
  faulty_output = Conv2(input)
  output[sites] = faulty_output[sites]
```





```
def inject(input, output):
  input[i6] = x
  faulty_output = Conv2(input)
  output[sites] = faulty_output[sites]
```





Input matrix

i1	i2	i3	i4
i5	x	i7	i8
i9	i10	i11	i12
i13	i14	i15	i16

Weight matrix

w1	w2
w3	w4

Output matrix

x1	o2	o3
x3	o4	o6
o7	o8	o9

DNN Error Propagation

DRAM

```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
```

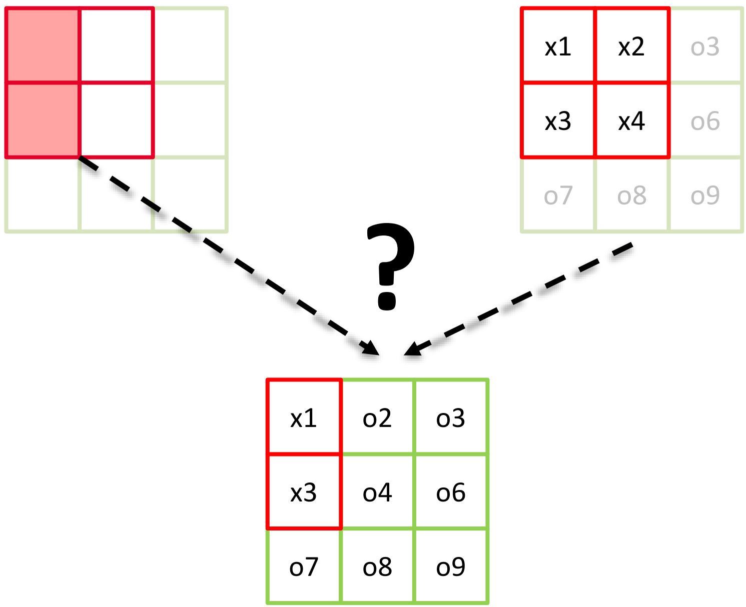
CBUF

```
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            … x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

(a) NVDLA Style Dataflow

Dataflow reuse sites (subset of the window)

Output window values



Final faulty output for a CBUF error

DNN Error Propagation

DRAM

```
for(k1=0; k1<K1; k1++)
 pfor(k0=0; k0<K0; k0++)
  for(c1=0; c1<C1; c1++)
   for(y1=0; y1<Y1; y1++)
    for(x1=0; x1<X1; x1++)
```

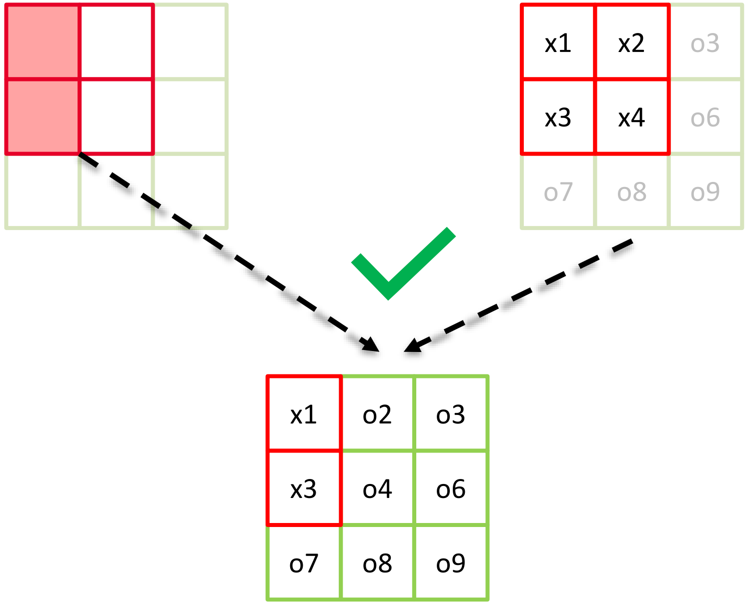
CBUF

```
     pfor(c0=0; c0<C0; c0++)
      for(r1=0; r1<R; r1++)
       for(s1=0; s1<S; s1++)
        for(y0=0; y0<Y0; y0++)
         for(x0=0; x0<X0; x0++)
          for(r=0; r0<1; r++)
           for(s=0; s0<1; s++) {
            k=k1*K0 + k0; c=c1*C0 + c0;
            … x = x1*X0 + x0;
            Output[k][y][x] +=
            Input[c][y+r][x+s] * Filter[k][c][r][s]; }
```

(a) NVDLA Style Dataflow

Dataflow reuse sites (subset of the window)

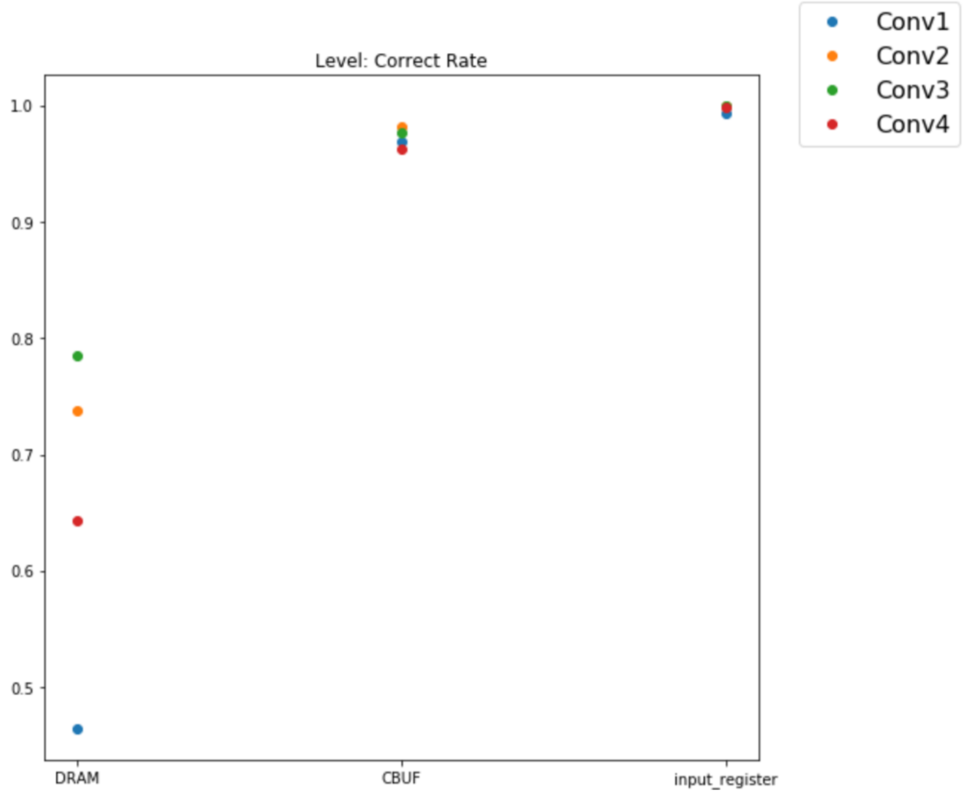
Output window values



Final faulty output for a CBUF error

NVDLA Injection Results

- Started with injecting a large constant value into NVDLA
- Takeaways:
  - More reuse (the more dataflow error sites that there are for a given injection) there is a greater chance for error
  - Register errors are relatively unlikely to produce errors



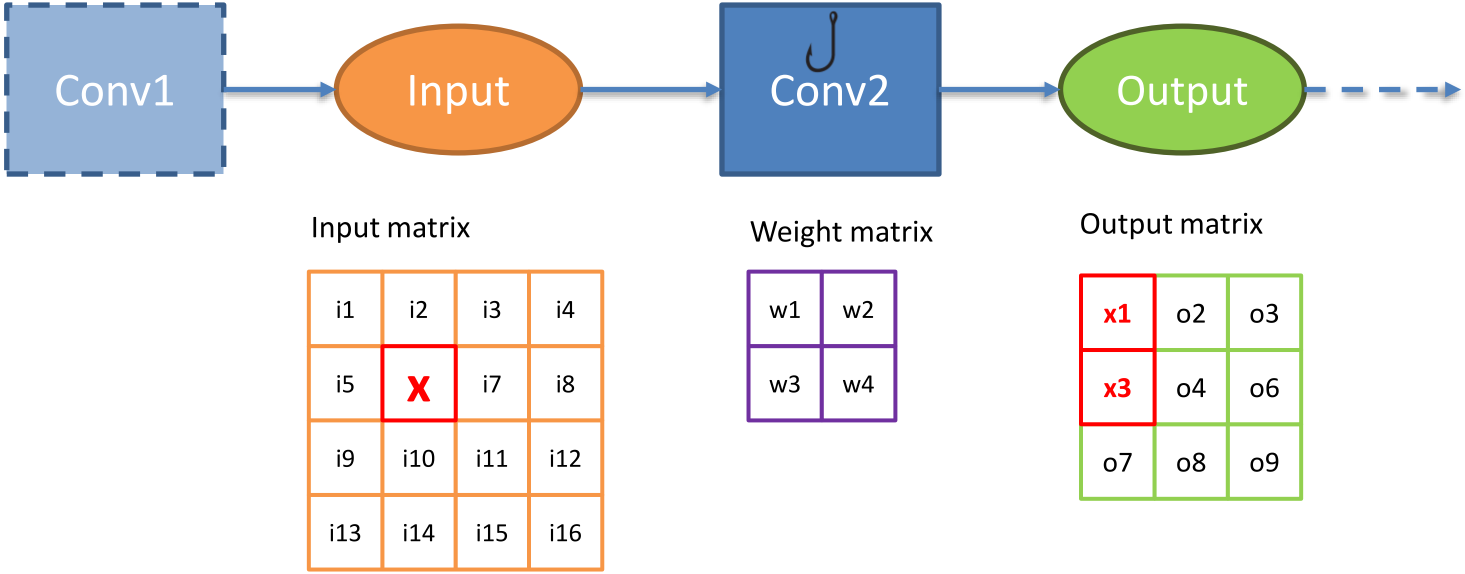
More reuse → more chance for error

Harvard John A. Paulson School of Engineering and Applied Sciences

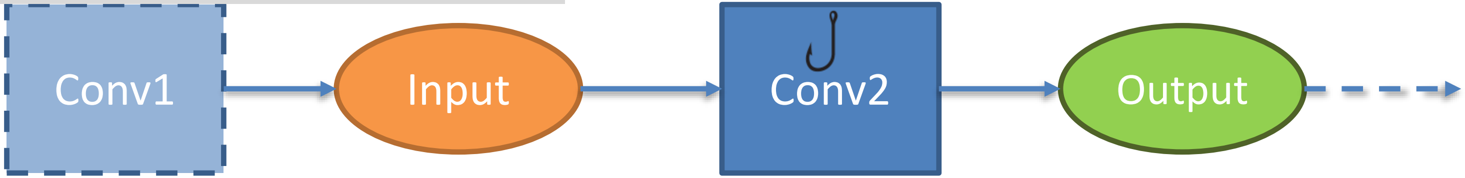
Error Mitigation: Ranger

- Ranger, *Chen et al. (2021)*, proposed a software-level fault mitigation technique that restricts output values to be between some maximum value (they propose the maximum value found through some training set).
- Since it's relatively lightweight, incorporated it into the injection to get more fair results.

```
def inject(input, output):
  input[i6] = x
  faulty_output = Conv2(input)
  output[sites] = faulty_output[sites]
```



```
def inject(input, output):
  input[i6] = x
  faulty_output = Conv2(input)
  output[sites] = faulty_output[sites]
  output = clamp(output, [-MAX, MAX])
```



Input matrix

i1	i2	i3	i4
i5	x	i7	i8
i9	i10	i11	i12
i13	i14	i15	i16

Weight matrix

w1	w2
w3	w4

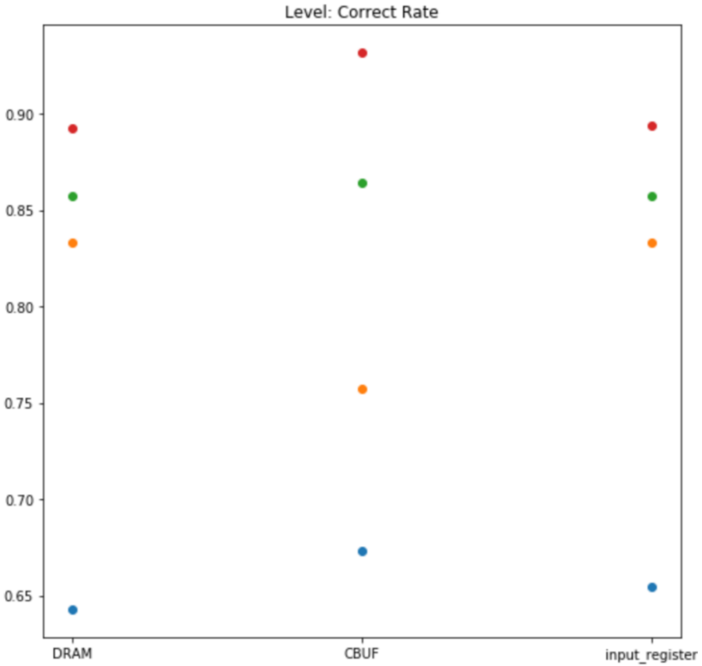
Output matrix

MAX	o2	o3
x3	o4	o6
o7	o8	o9

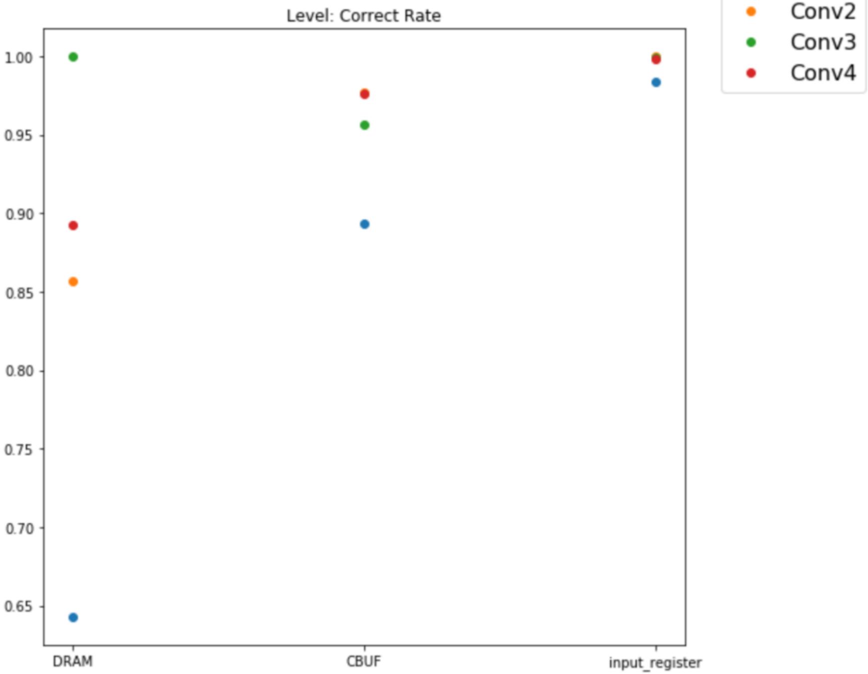
Assume `x1 > MAX`

Flipping Bit 3 w/ and w/o Ranger

No range clamping:



With range clamping:



Large errors see no difference → with ranging the same pattern emerges

Future Work/Directions

- Need to account for different FIT rates for different memory types
- Need to account for how long a value remains in memory
  - Not sure how much this will change things (might need sensitivity analysis)
- More analysis! (i.e. different networks, different dataflows/mappings)

Thank you all for listening!

Questions? Feedback?

Also – big thanks to Abdulrahman for helping at each step of the way :)

Please reach out with any questions (or to chat about whatever!)
Can find me over Slack or email me at jaylenwang@college.harvard.edu