Convolution Lowering & Tiling

Hardware System Design

Spring, 2023

Outline

- Week 9 – Pytorch Intro
- Week 10 – Quantization
- Week 11 – Convolution Lowering & Tiling

train_py.ipynb

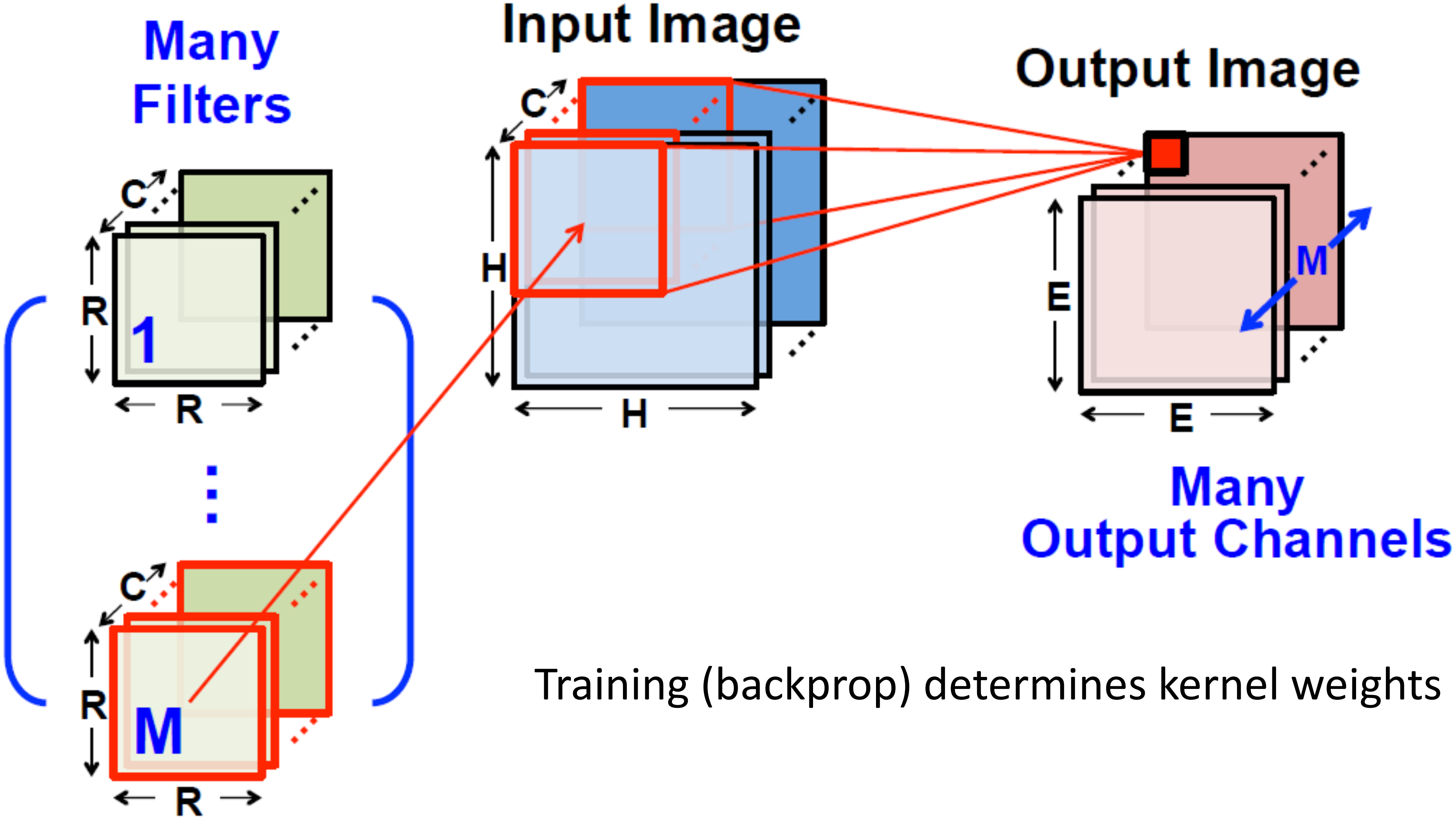
```python
class CNN(torch.nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 6,
                            kernel_size=3, stride=1, padding=0, bias=False),
                            # output shape 26 x 26 x 6 x 9 = 36504
            torch.nn.BatchNorm2d(6),
            torch.nn.ReLU(),
            )
        self.fc1 = torch.nn.Linear(4056, 30, bias=False) # 4056 * 30 = 121680
        self.fc2 = torch.nn.Linear(30, 10, bias=False) # 30 * 10 = 300
        self.layer2 = torch.nn.Sequential(
            self.fc1,
            torch.nn.ReLU(),
            self.fc2
            )

    def forward(self, x):
        out = self.layer1(x)
        out = out.view(out.size(0), -1)   # Flatten them for FC
        out = self.layer2(out)
        return out
```

[Chen, 2016]

Convolution: 3D Input / 3D Output

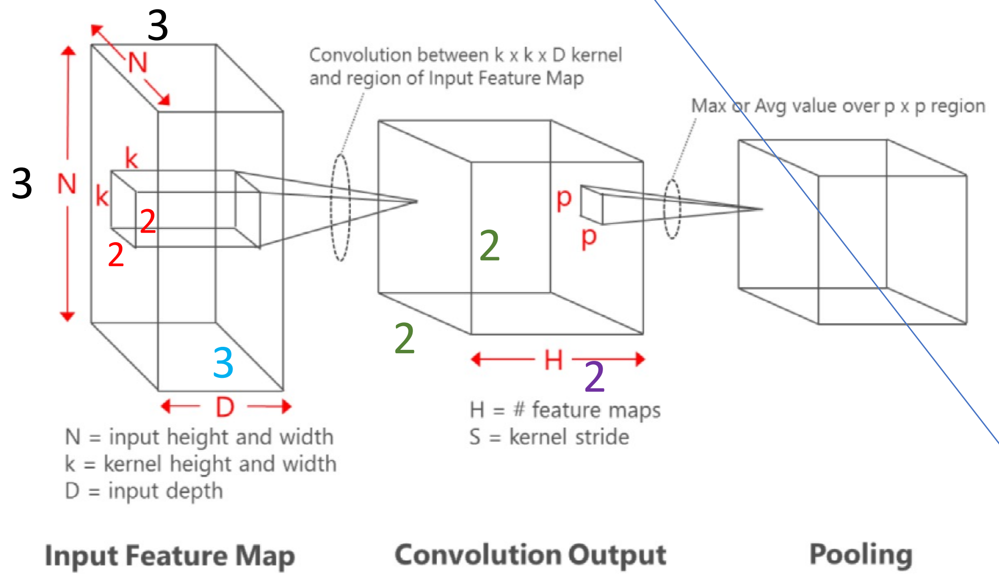


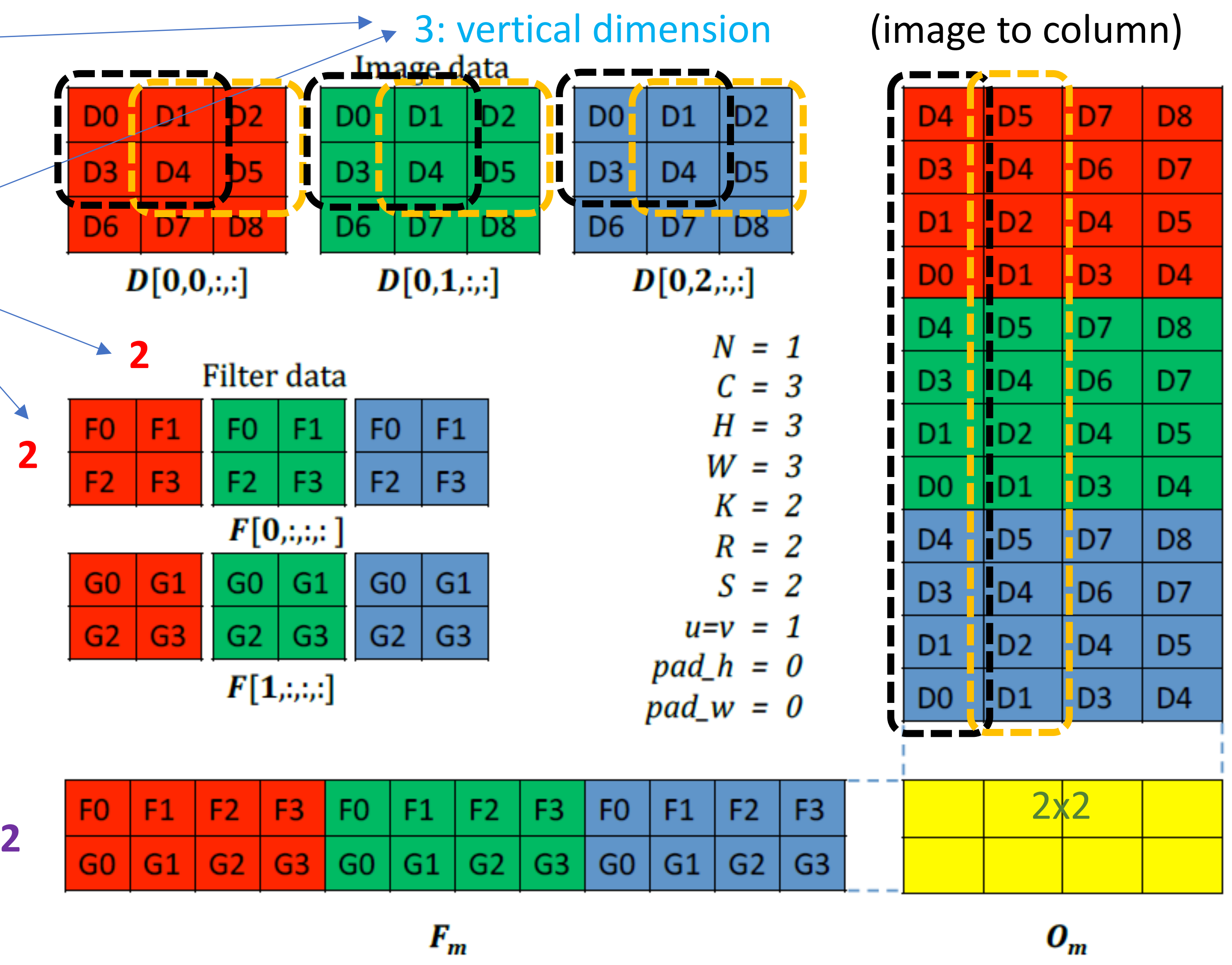
Training (backprop) determines kernel weights

[Chetlur 2014]

Convolution with Matrix Multiplication (called Convolution Lowering)

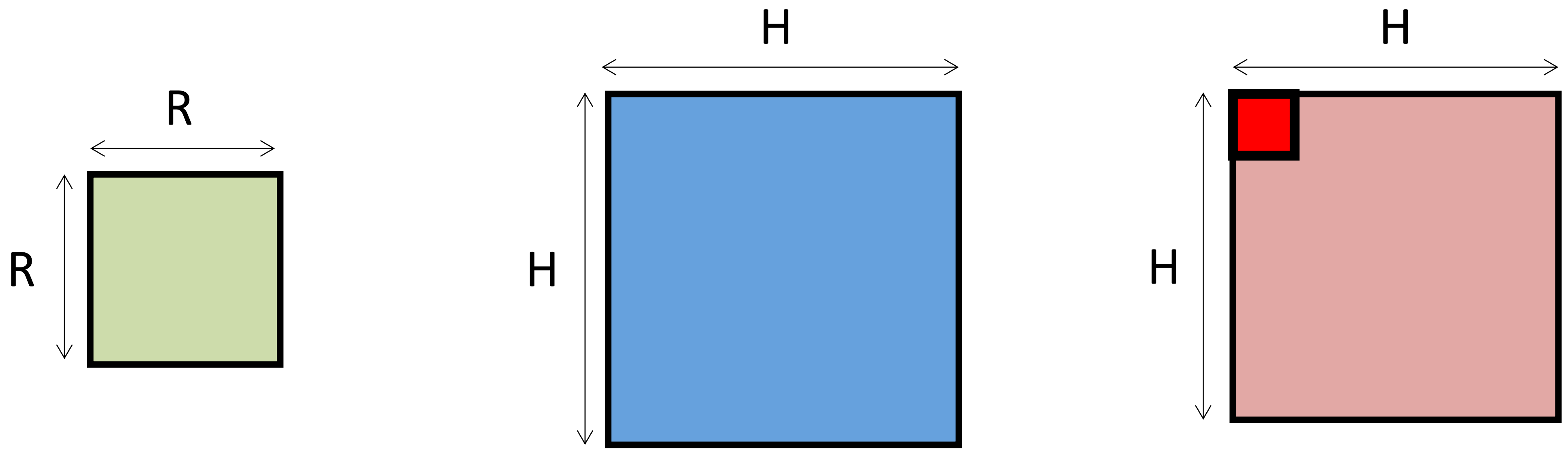
- Input: 3x3x3
- Output: 2x2x2
- Convolutional kernel: 3x2x2



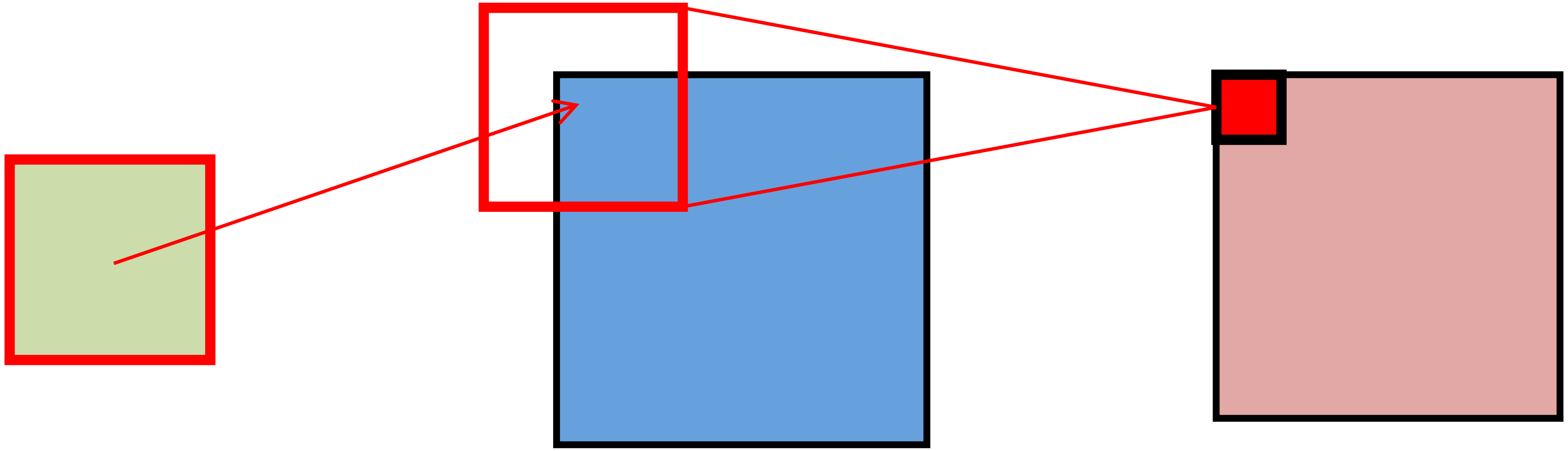


Case 1. The simplest case

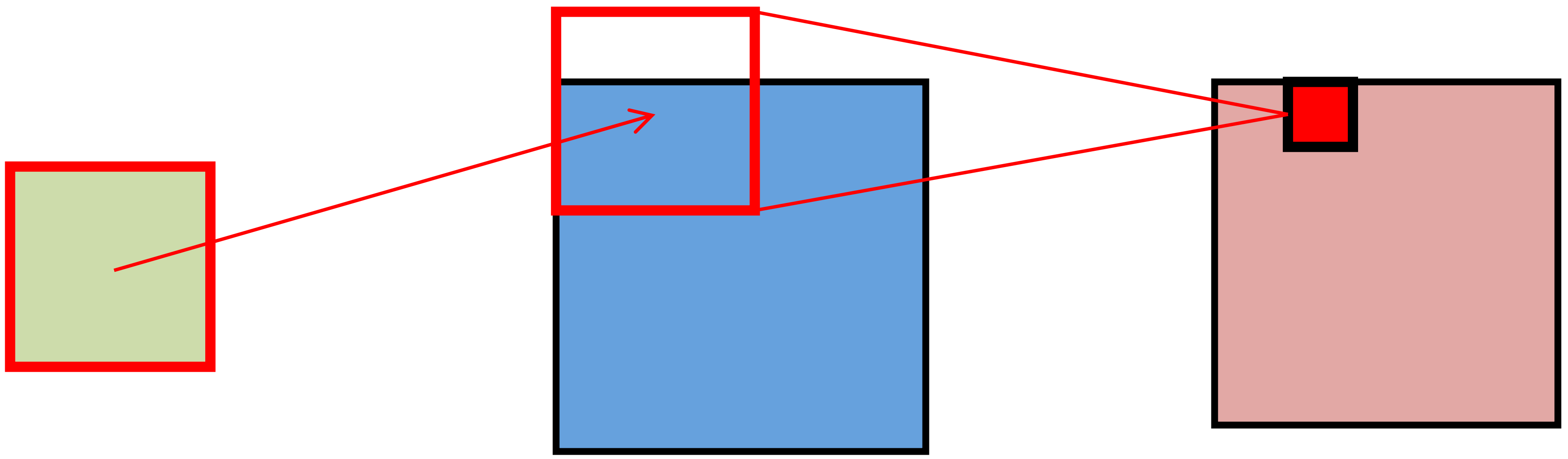
- We execute convolution between a filter and an input image, to produce an output image.
- Assume that (stride) = (padding * 2 + 1), so that output has the same resolution with input.

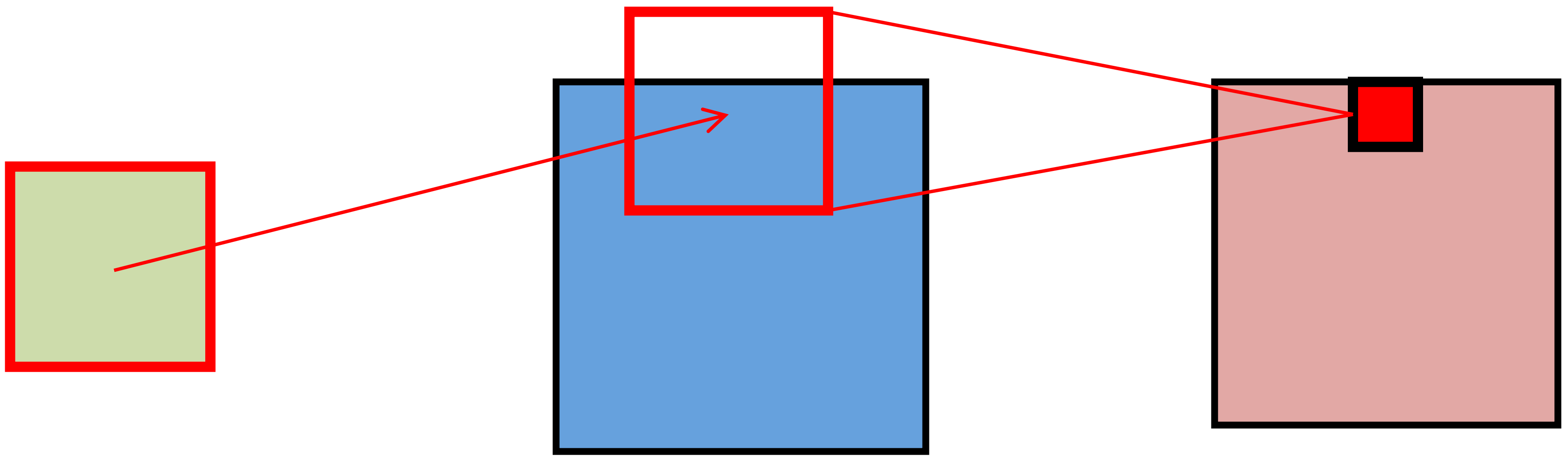


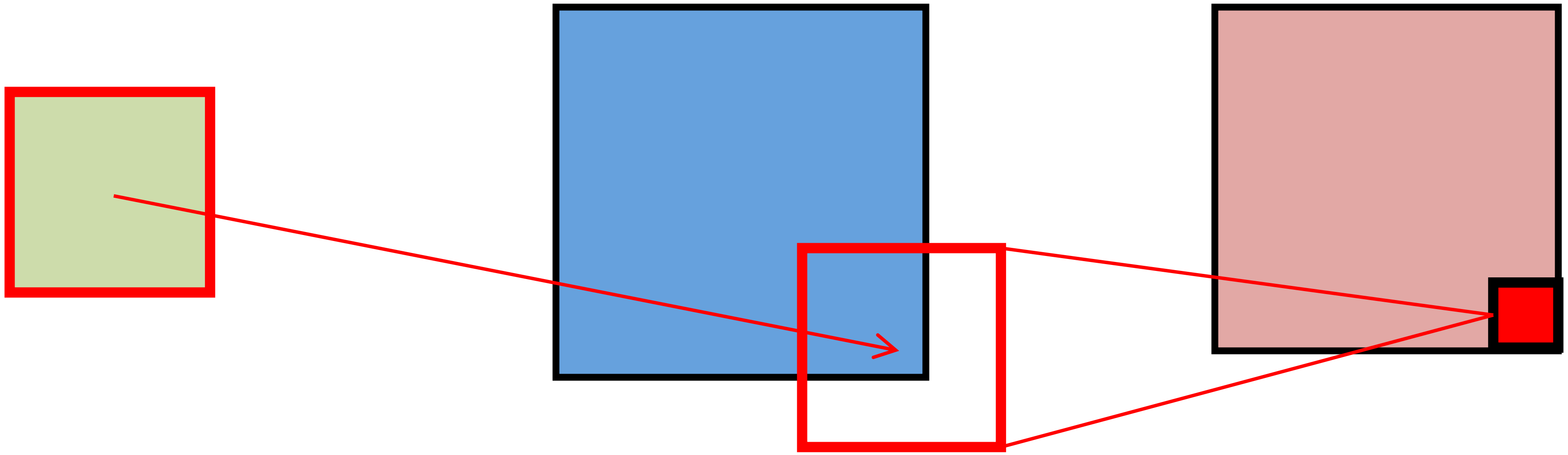
- Inner product is performed between the two red boxes



- We perform convolution while sliding the filter across the input image.

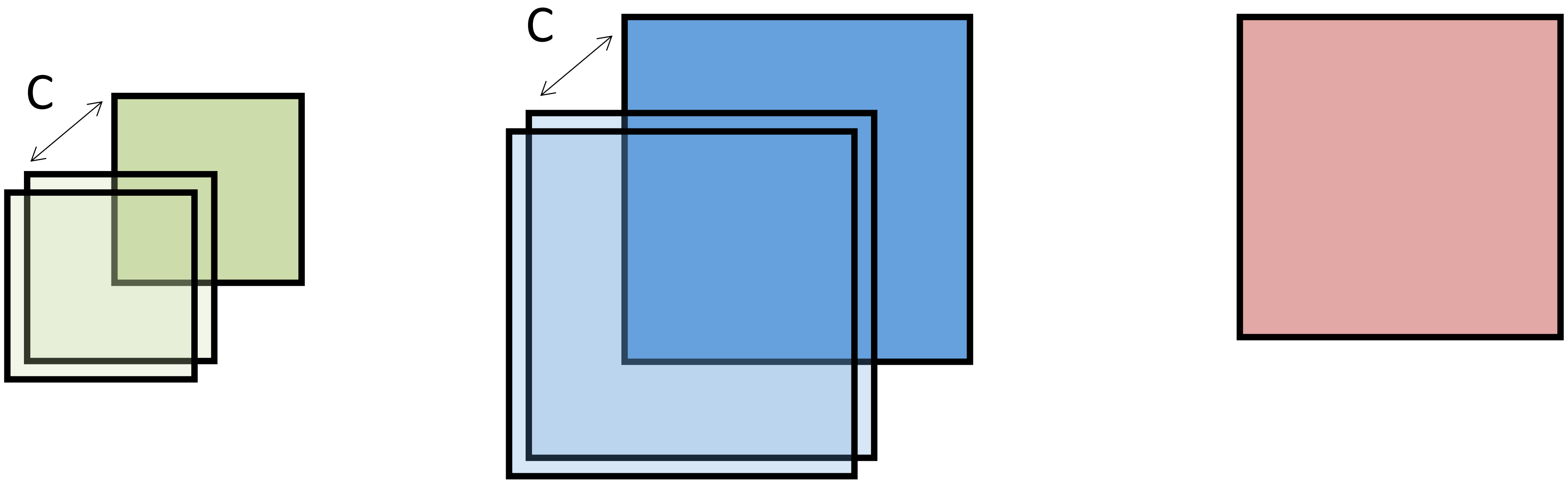




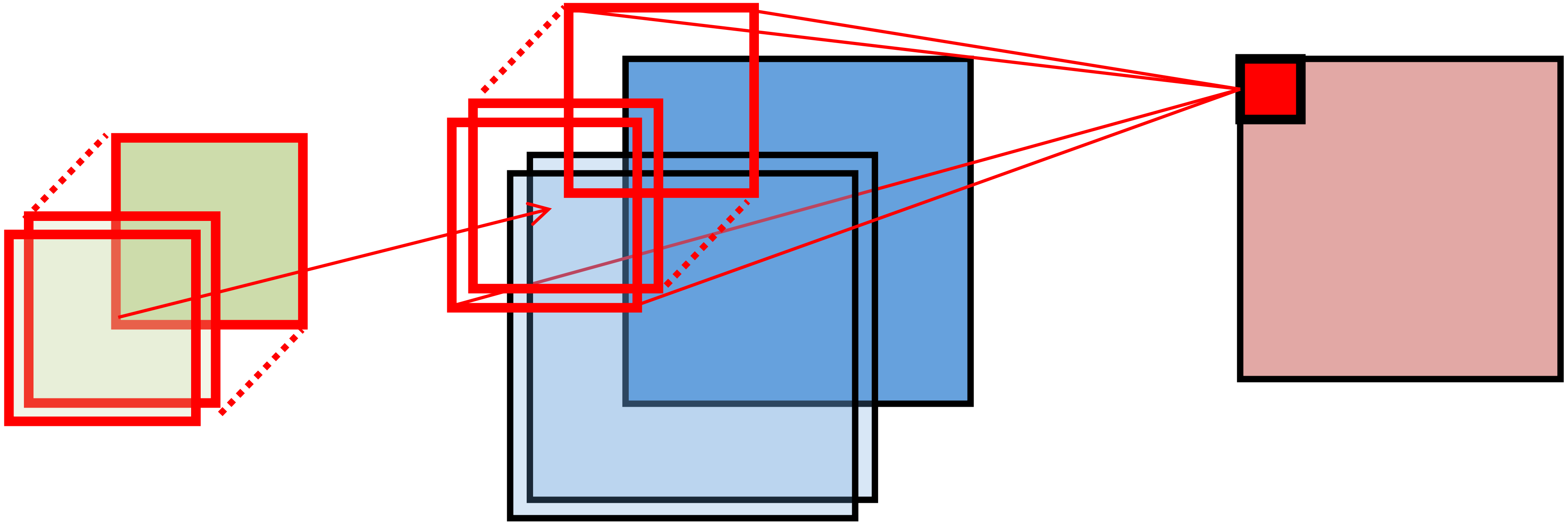


Case 2. Multiple Channels

- Indeed, a filter and an image usually have multiple channels.

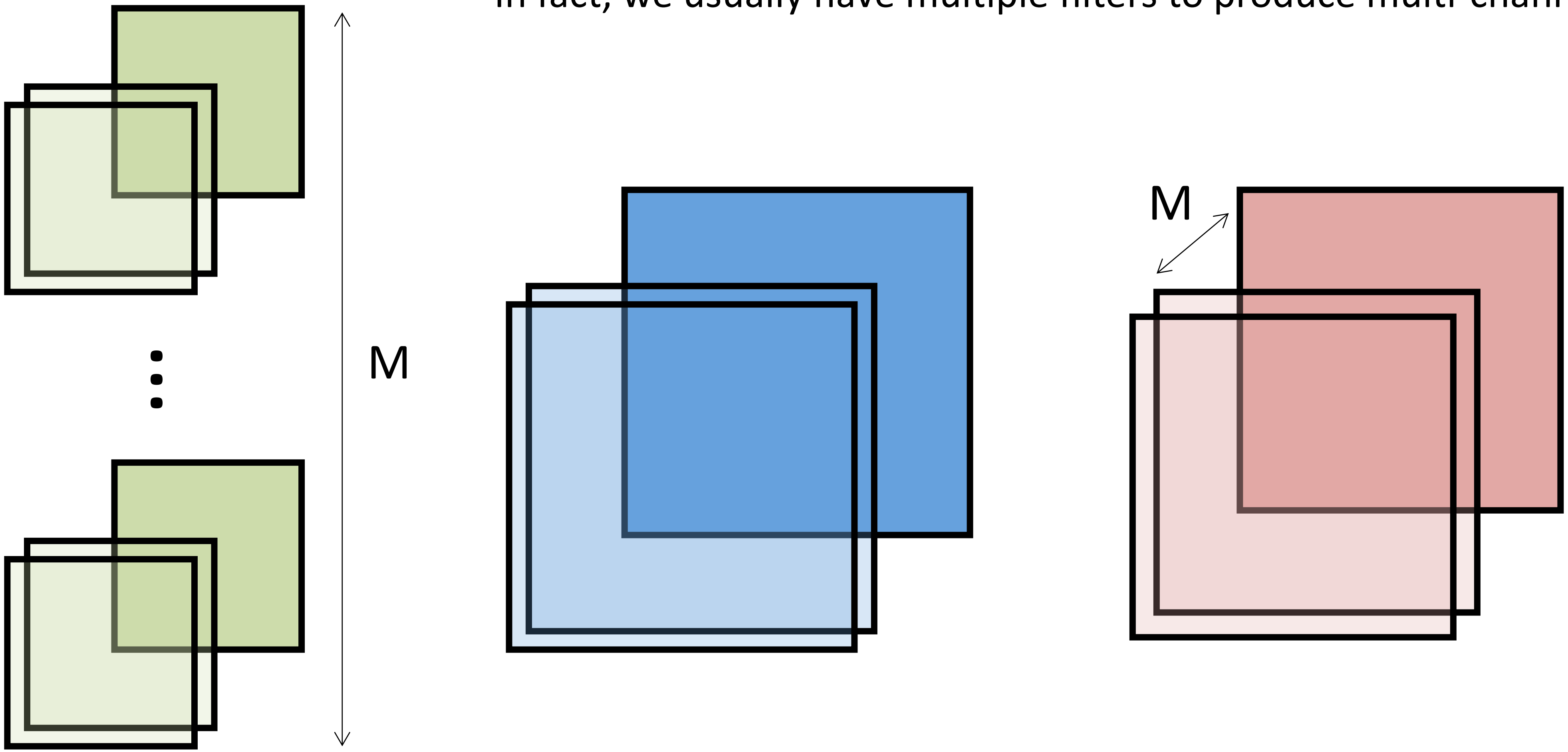


- Again, inner product is performed between the two red boxes.

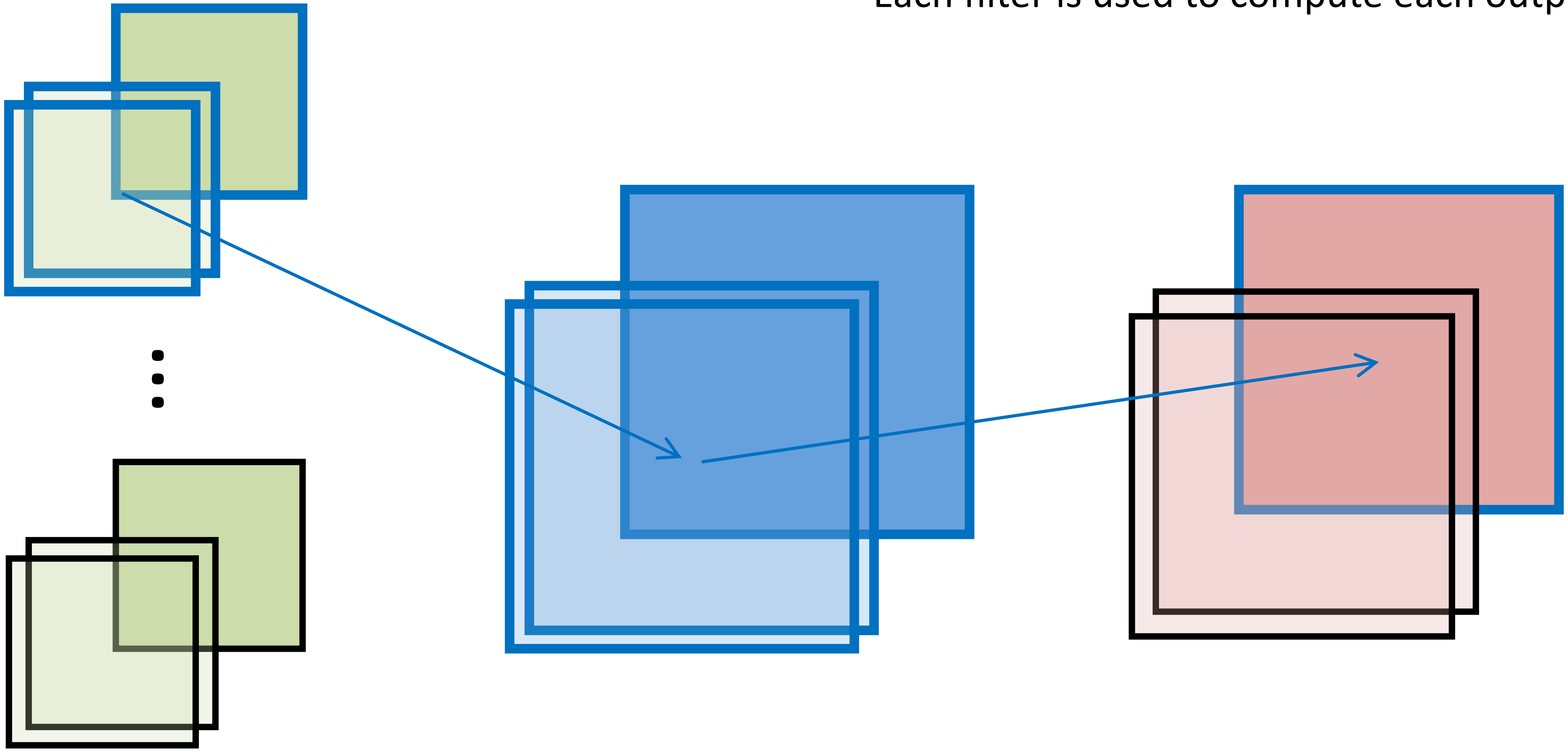


Case 3. Multiple Filters

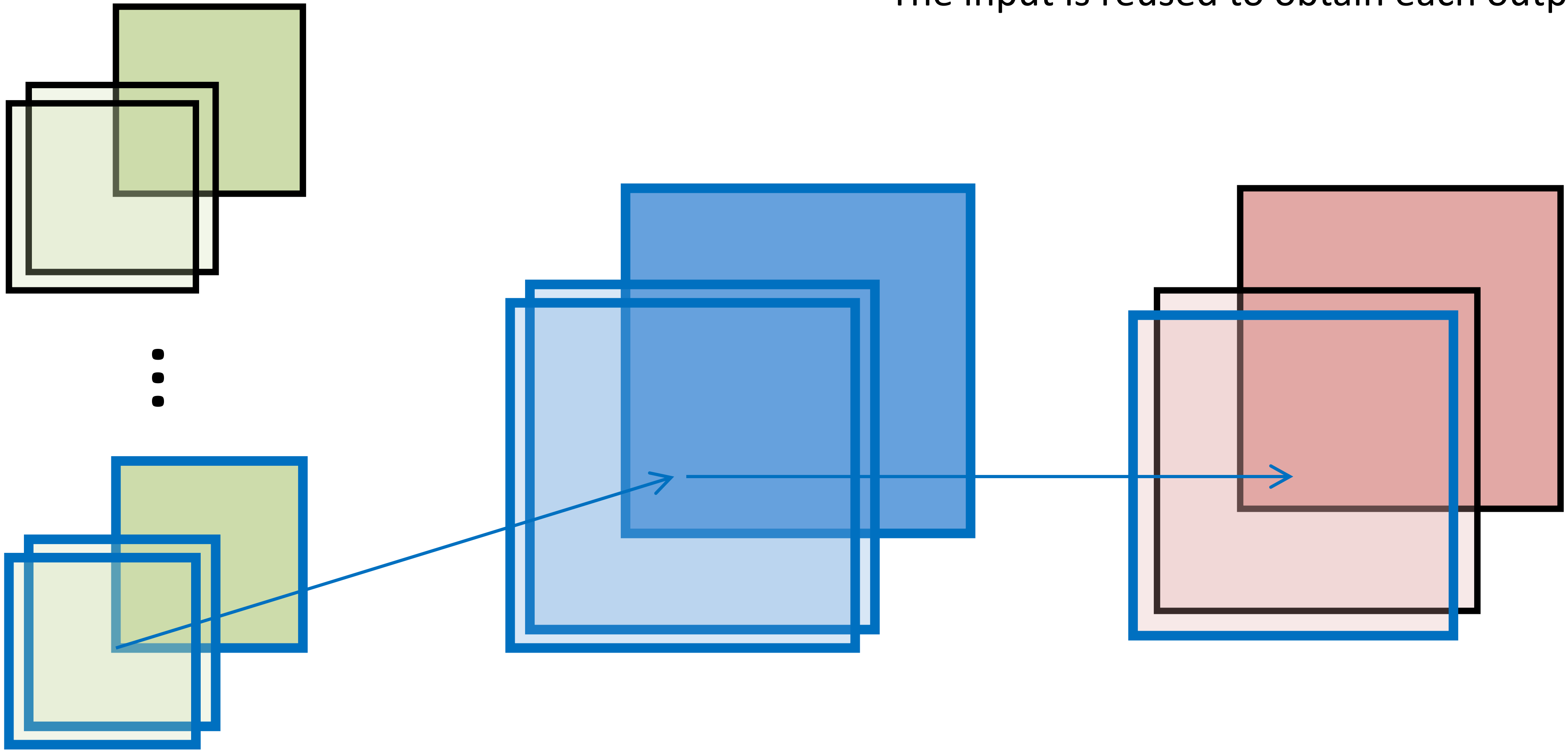
- In fact, we usually have multiple filters to produce multi-channel outputs.



- Each filter is used to compute each output channel.

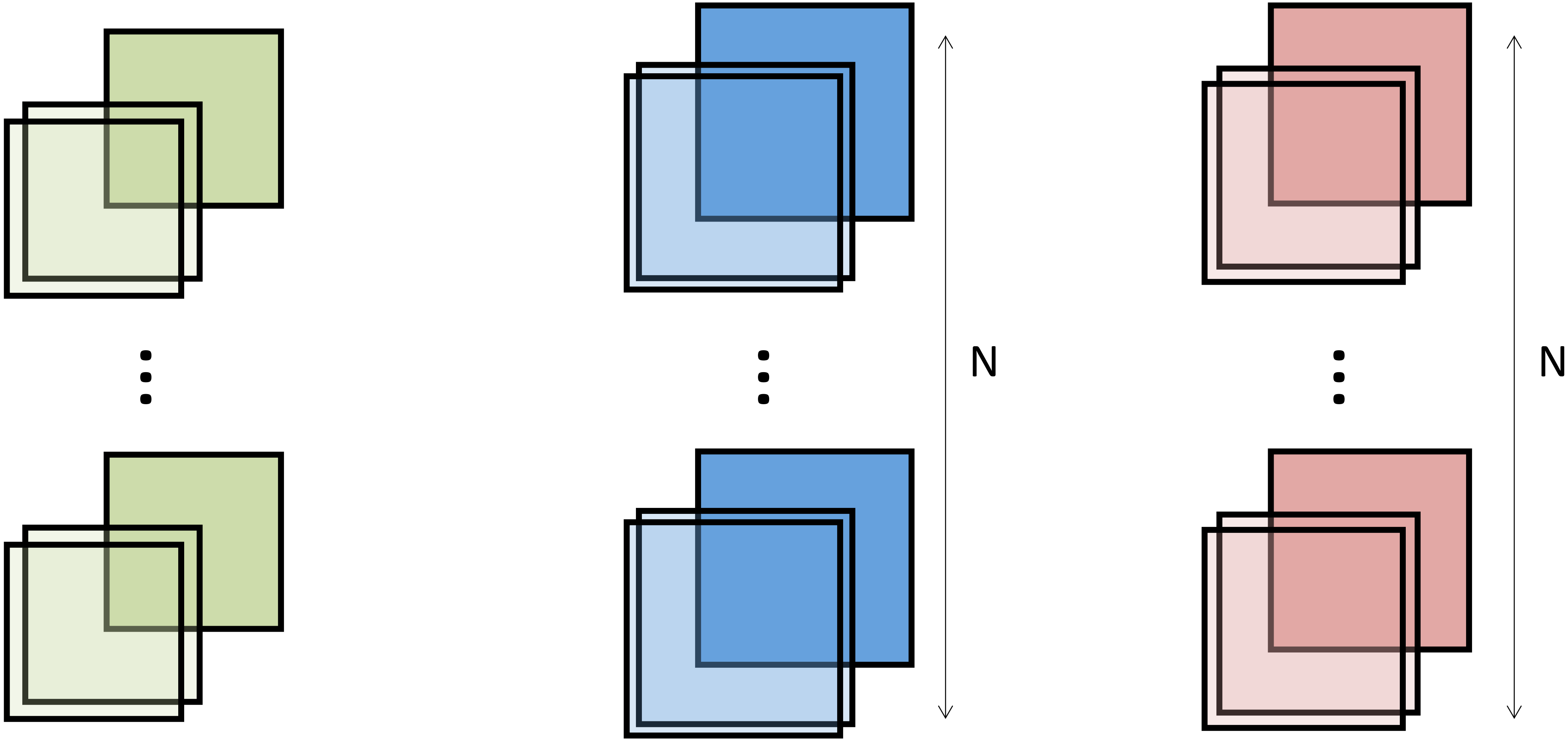


- The input is reused to obtain each output channel.

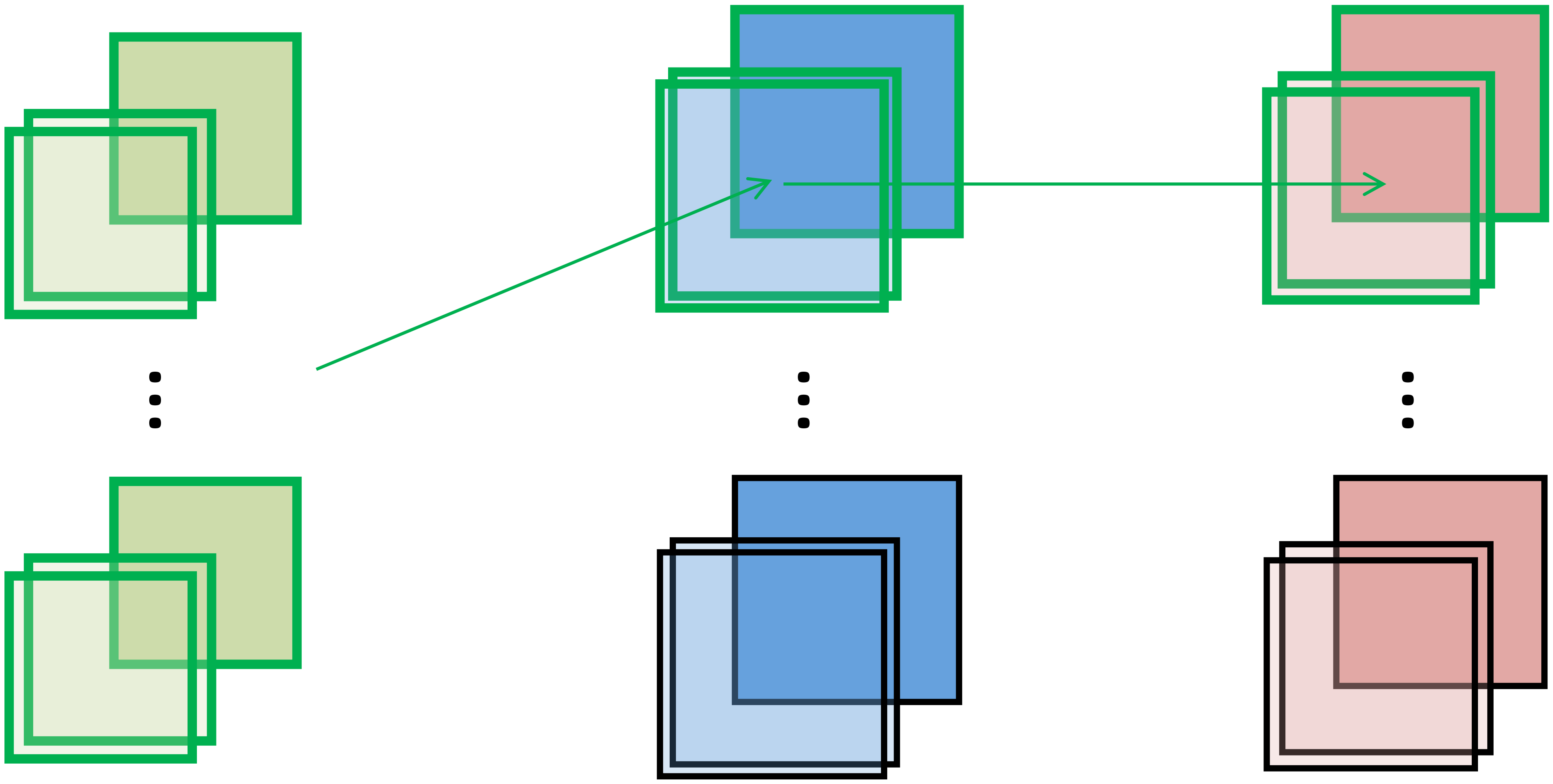


Case 4. Multiple Data

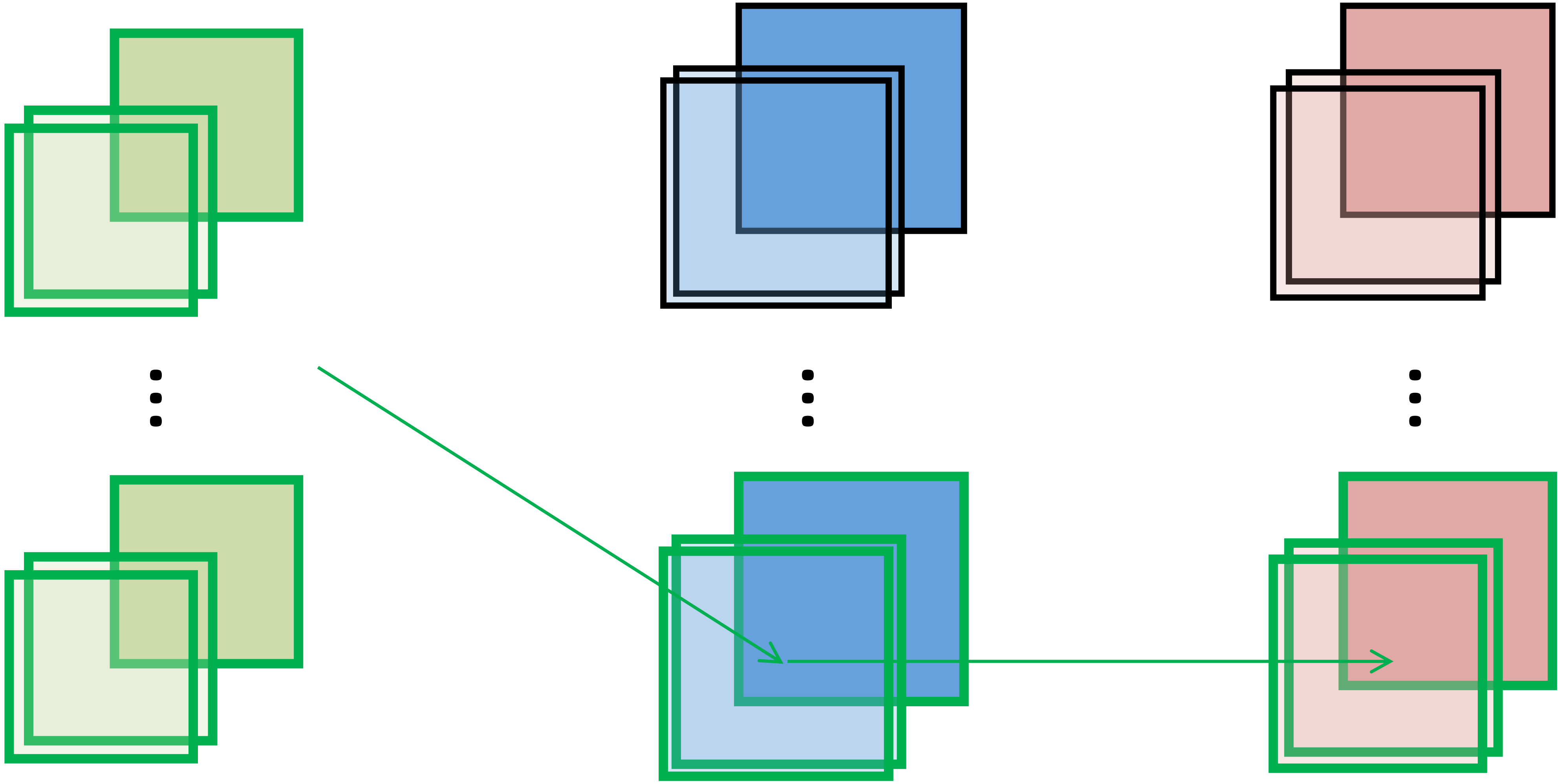
- We can process multiple data at the same time.



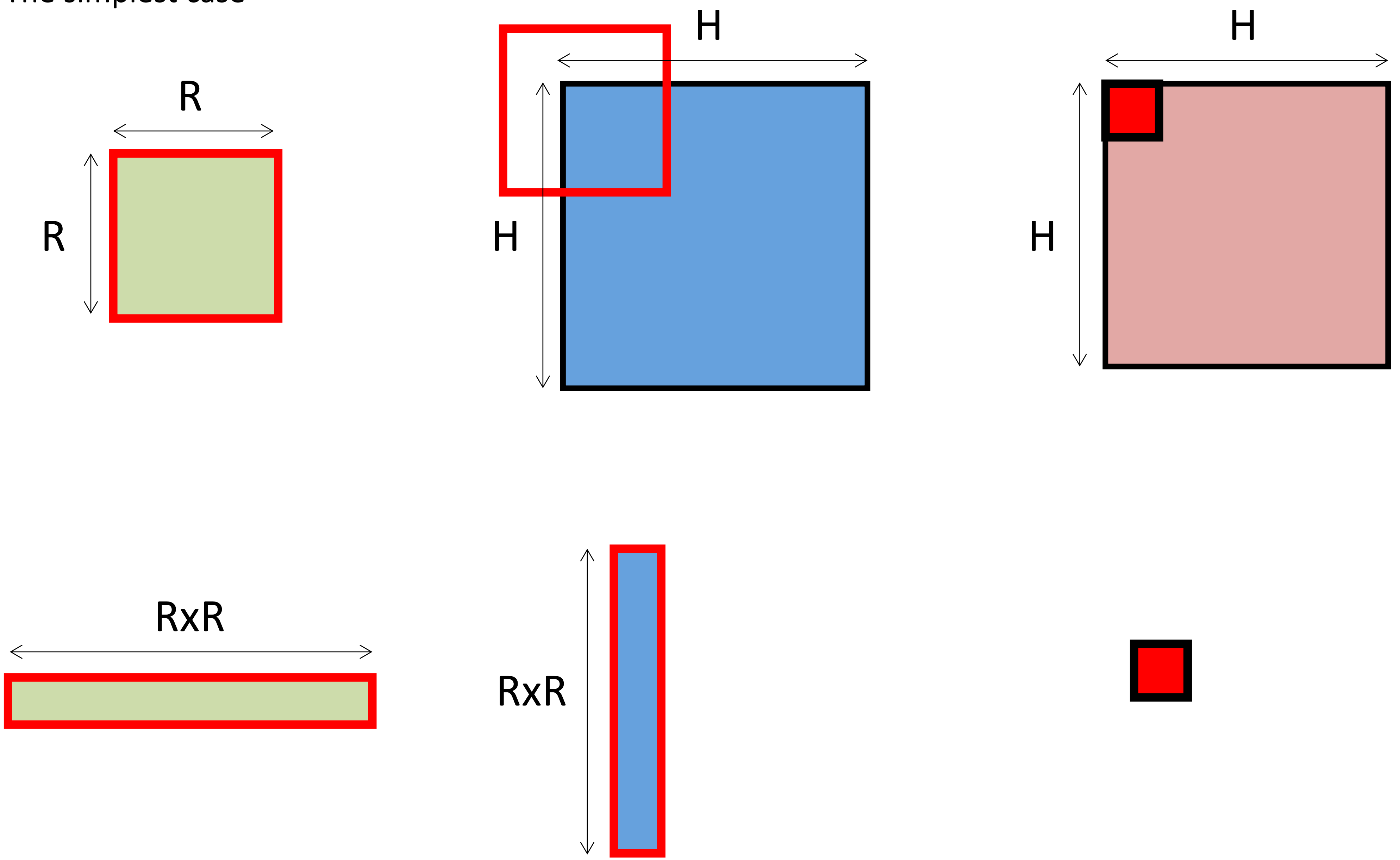
- Each input is used to compute each output

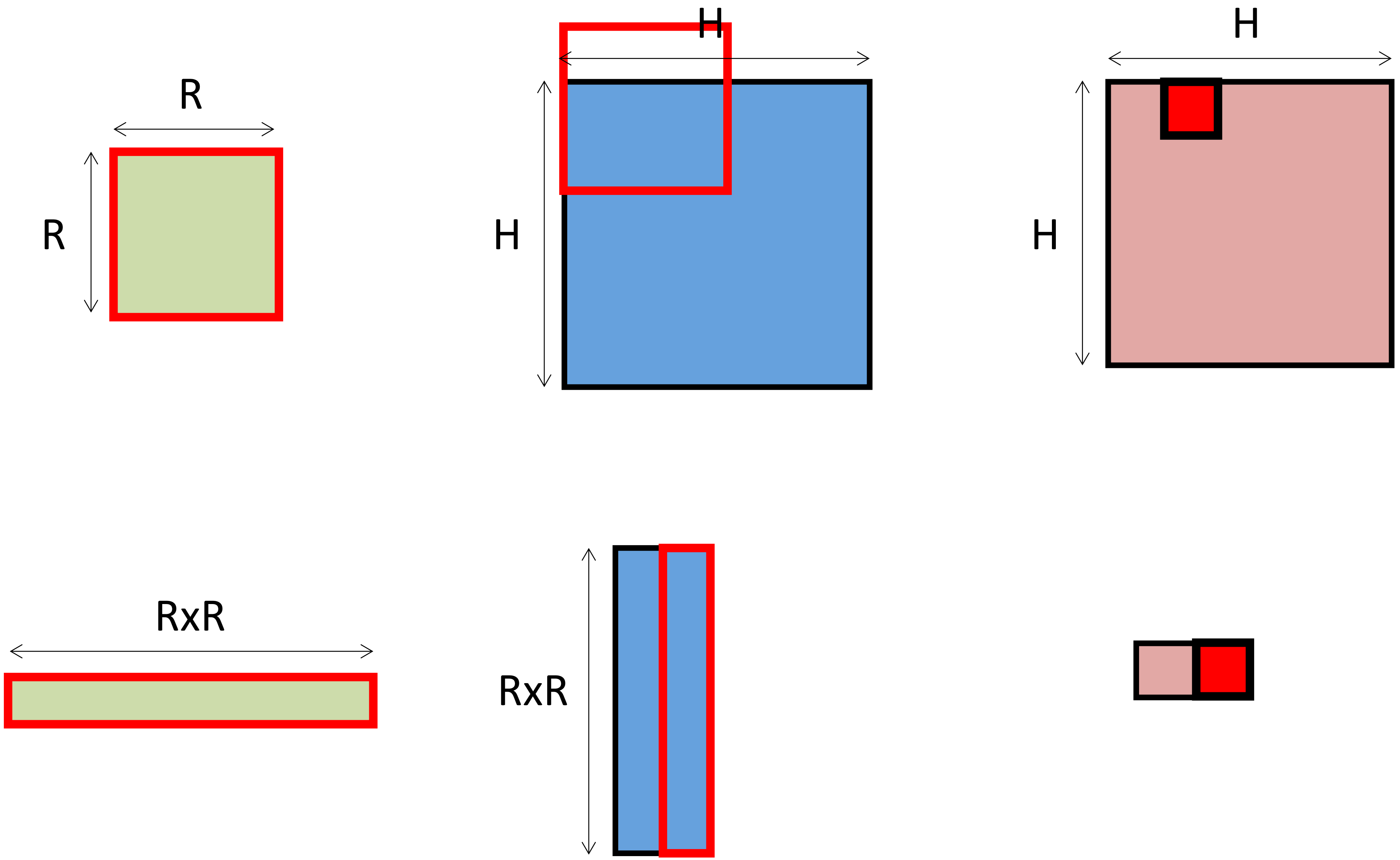


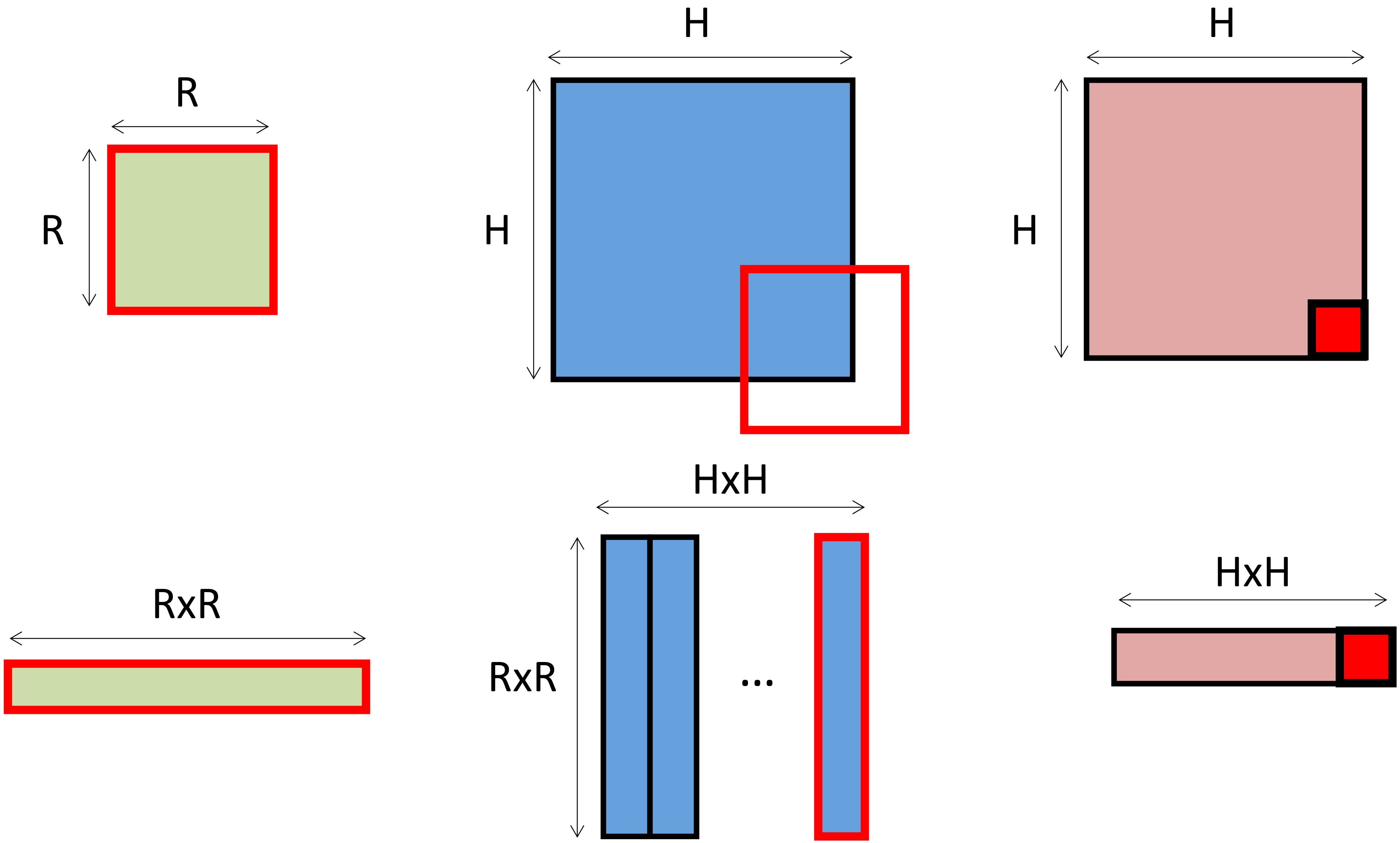
- The filters are reused to obtain each output.



Case 1. The simplest case

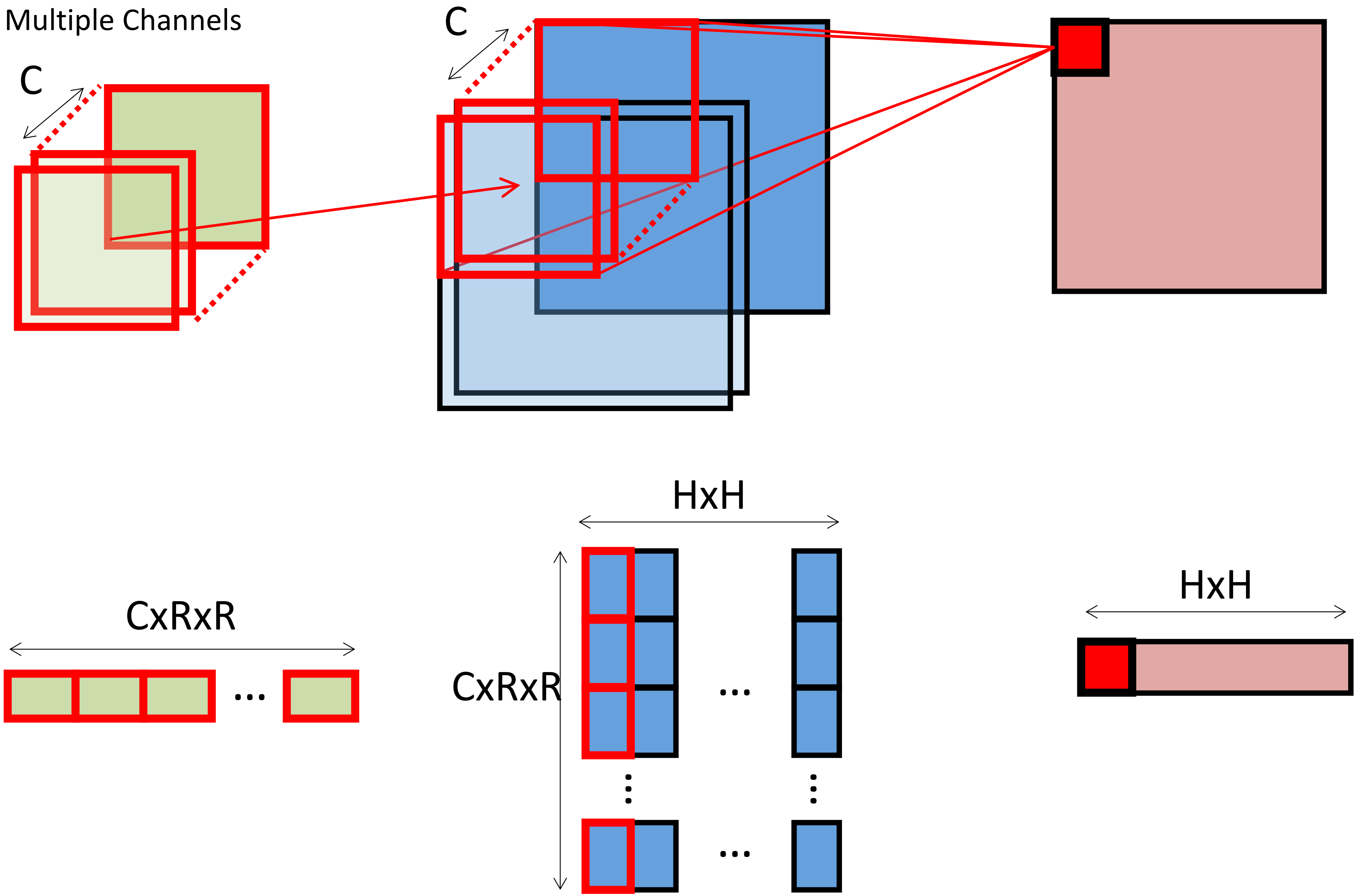



R
R
H
H
H
H
RxR
RxR

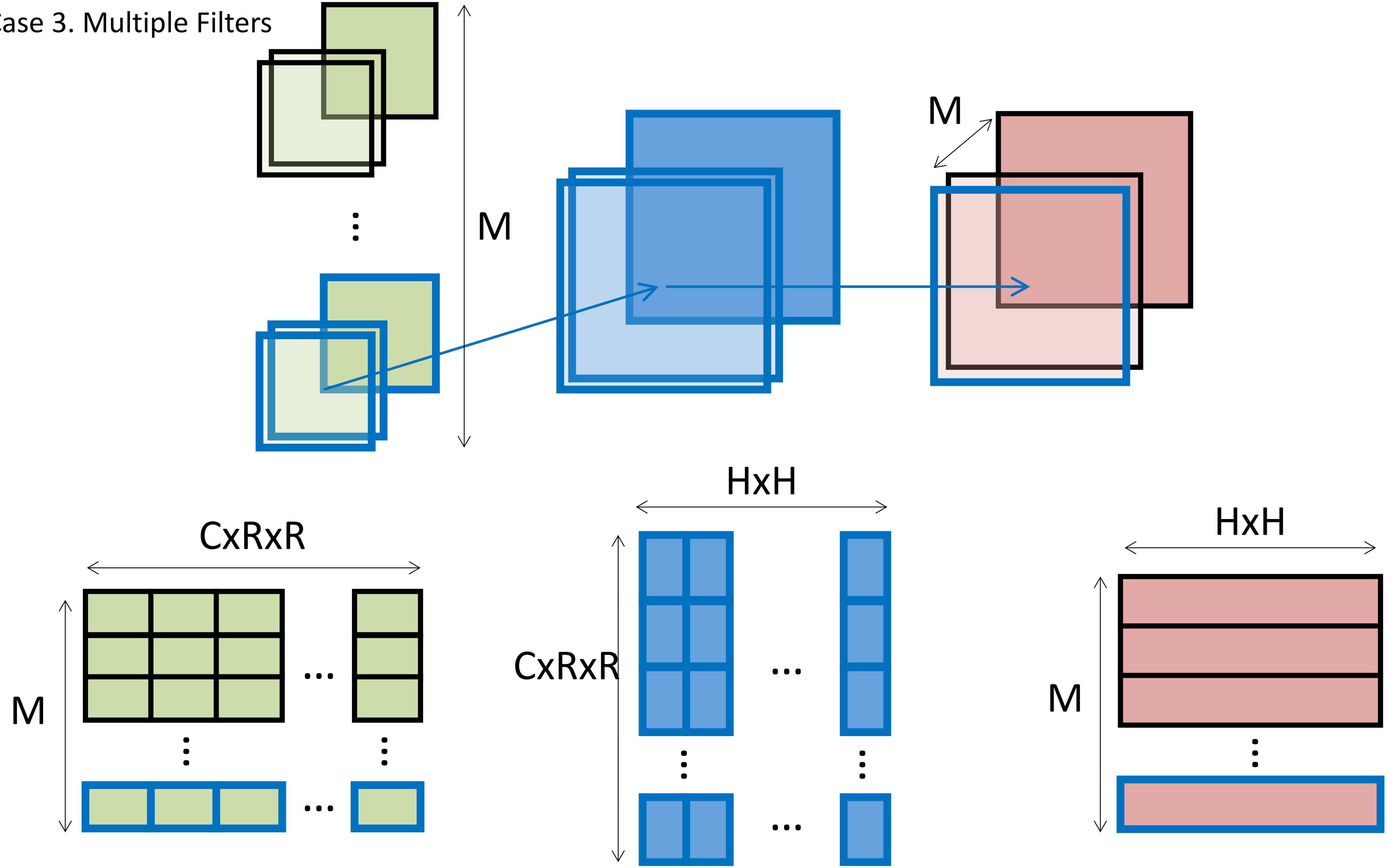


R
R
H
H
H
H
RxR
HxH
RxR
...
HxH

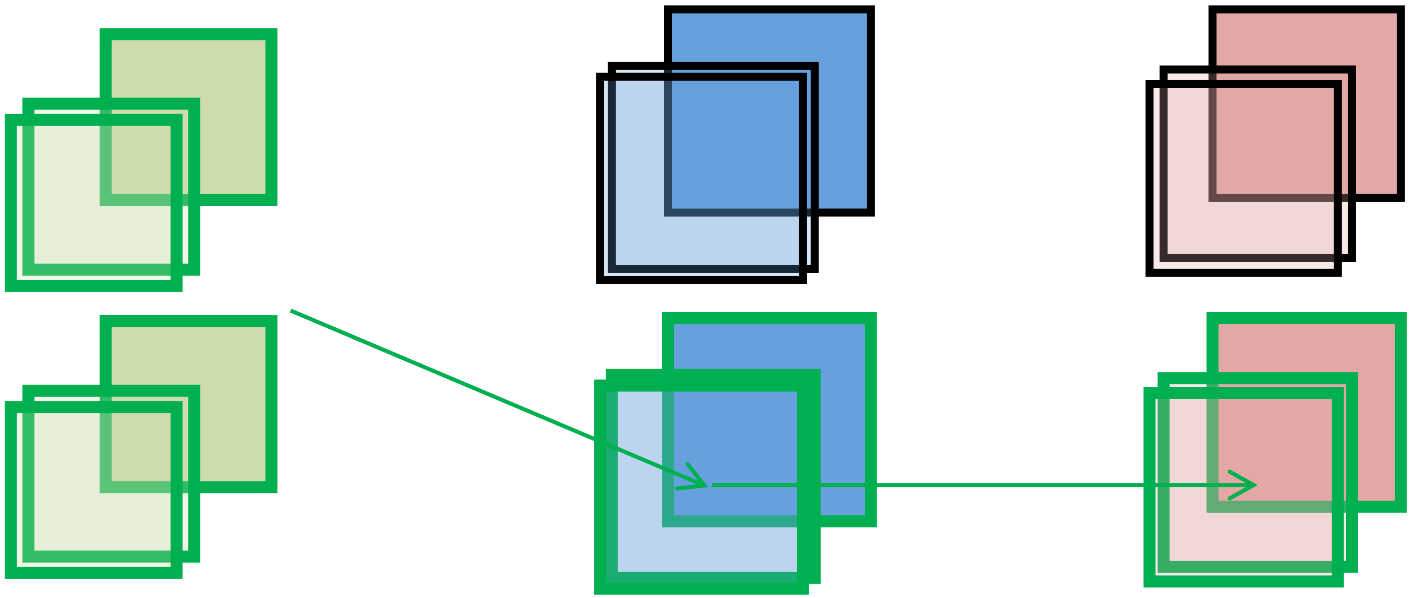
Case 2. Multiple Channels

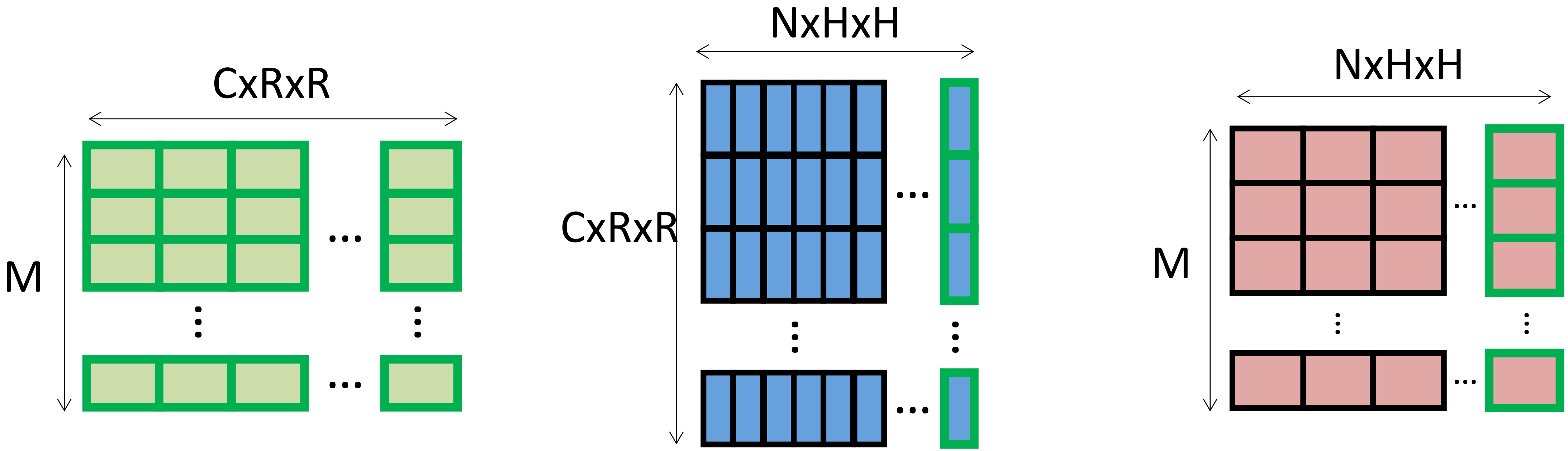


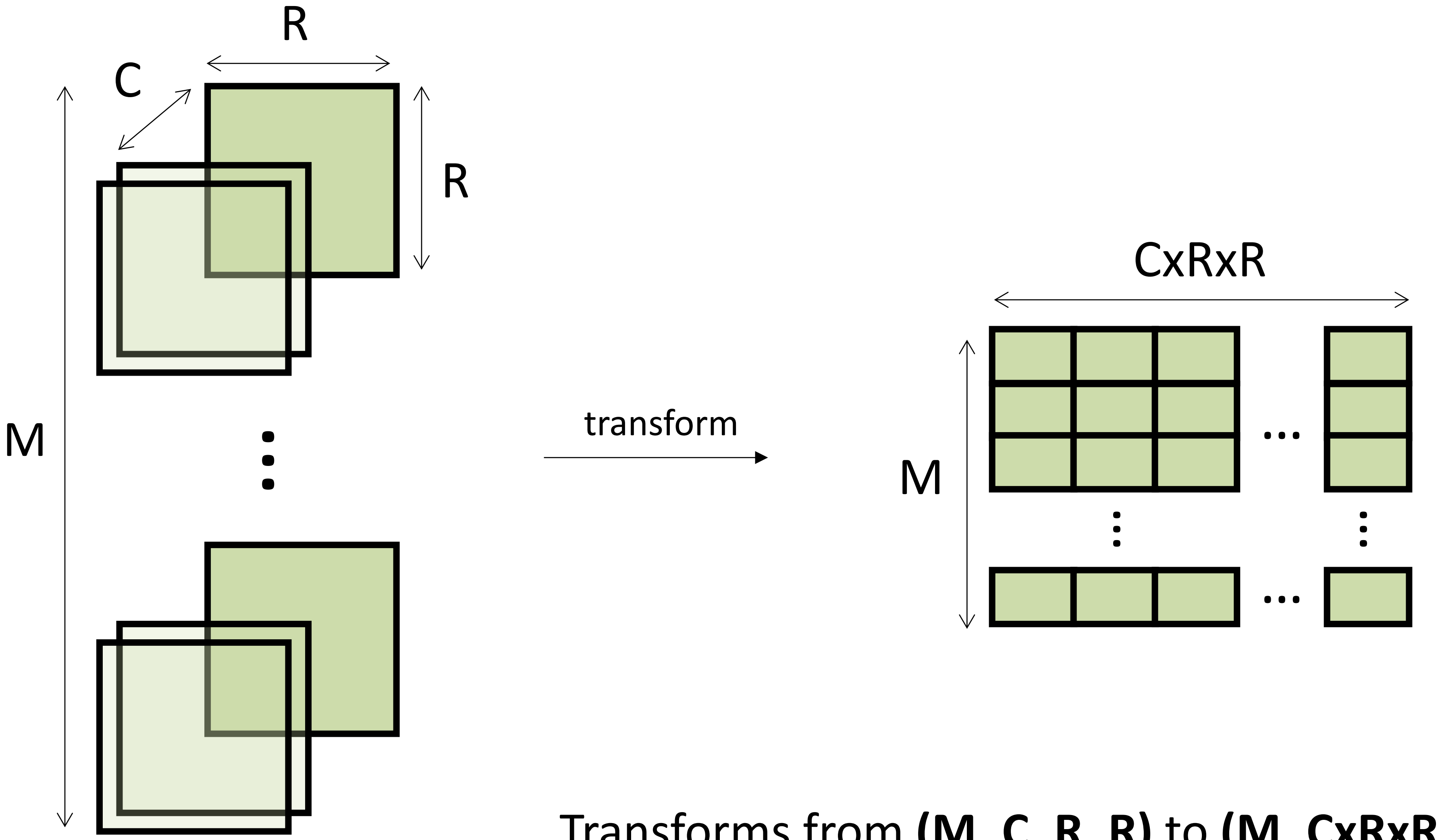
Case 3. Multiple Filters



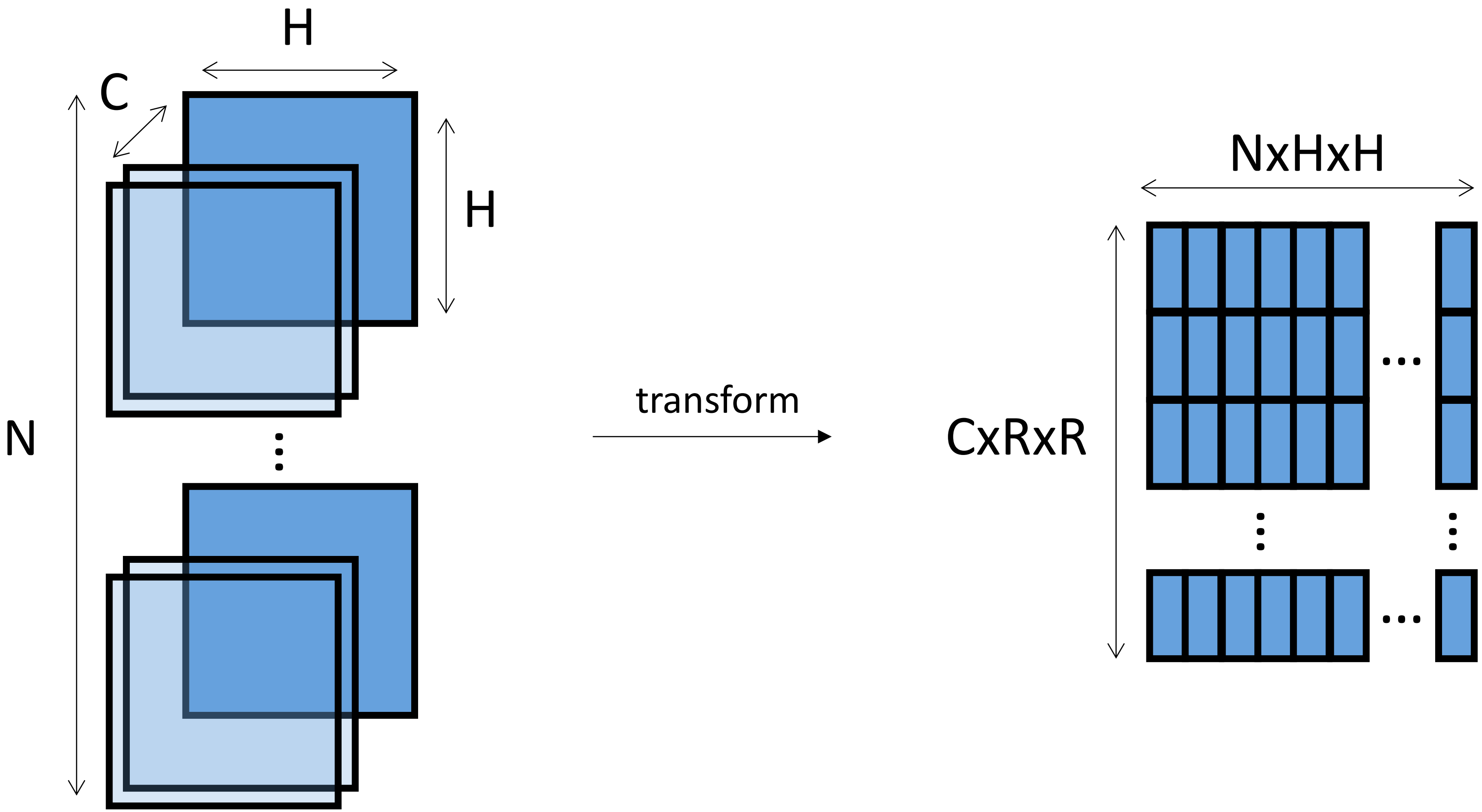
Case 4. Multiple Data



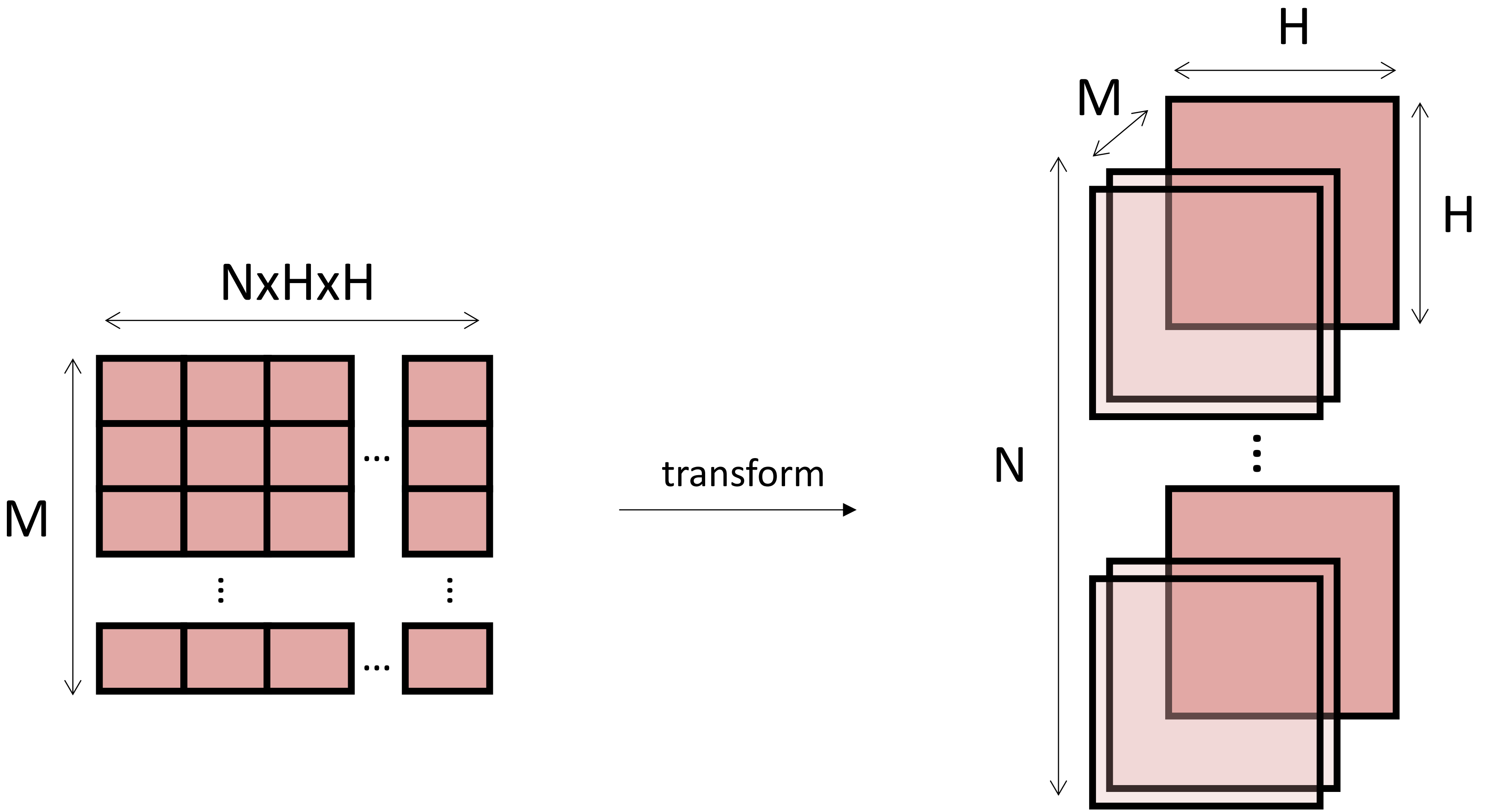




Transforms from **(M, C, R, R)** to **(M, CxRxR)**



Transforms from **(N, C, H, H)** to **(CxRxR, NxHxH)**



Transforms from **(M, NxHxH)** to **(N, M, H, H)**

- torch.reshape

```python
A = torch.Tensor(2, 3, 4, 5)
print(A.size())
```

```
torch.Size([2, 3, 4, 5])
```

```python
A = A.reshape(2*3, 4, 5)
print(A.size())
```

```
torch.Size([6, 4, 5])
```

```python
A = A.reshape(2, 3*4, 5)
print(A.size())
```

```
torch.Size([2, 12, 5])
```

```python
A = A.reshape(2*3*4*5)
print(A.size())
```

```
torch.Size([120])
```

- torch.transpose

```python
B = torch.Tensor(2, 3, 4, 5)
print(B.size())
```

```
torch.Size([2, 3, 4, 5])
```

```python
B = B.transpose(0, 1)
print(B.size())
```

```
torch.Size([3, 2, 4, 5])
```

```python
B = B.transpose(1, 2)
print(B.size())
```

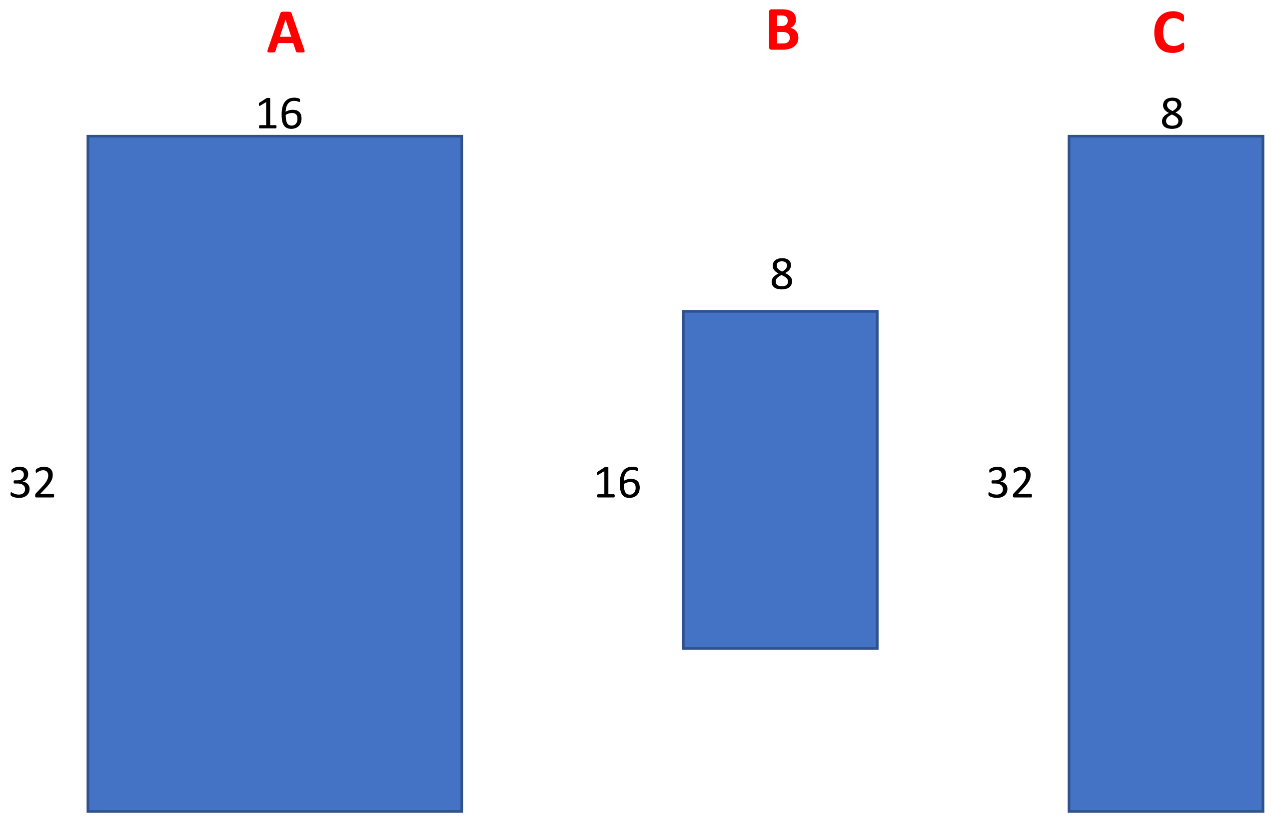
```
torch.Size([3, 4, 2, 5])
```

```python
B = B.transpose(2, 3)
print(B.size())
```

```
torch.Size([3, 4, 5, 2])
```

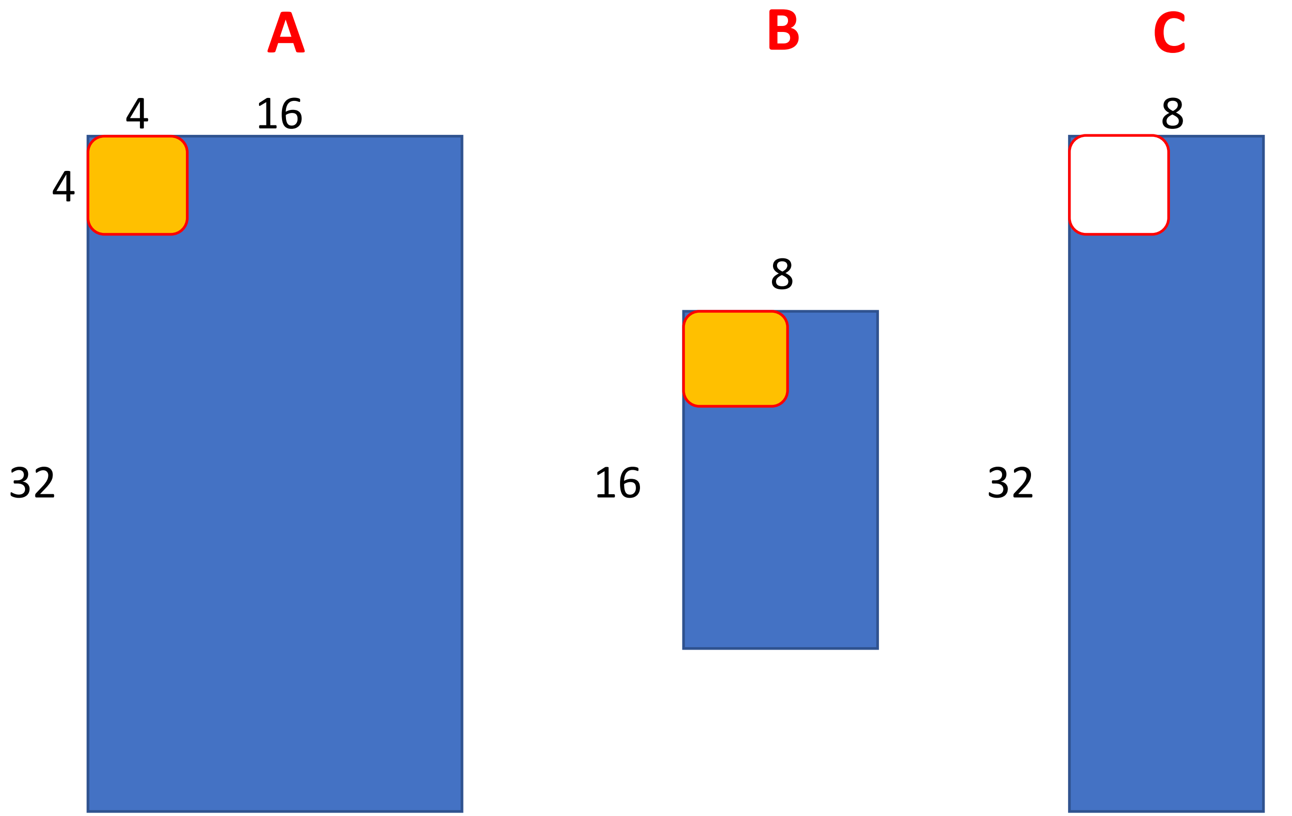
Pipelined Tile based Multiplications

- First, CPU will write tiles A and B to the local memory of systolic array



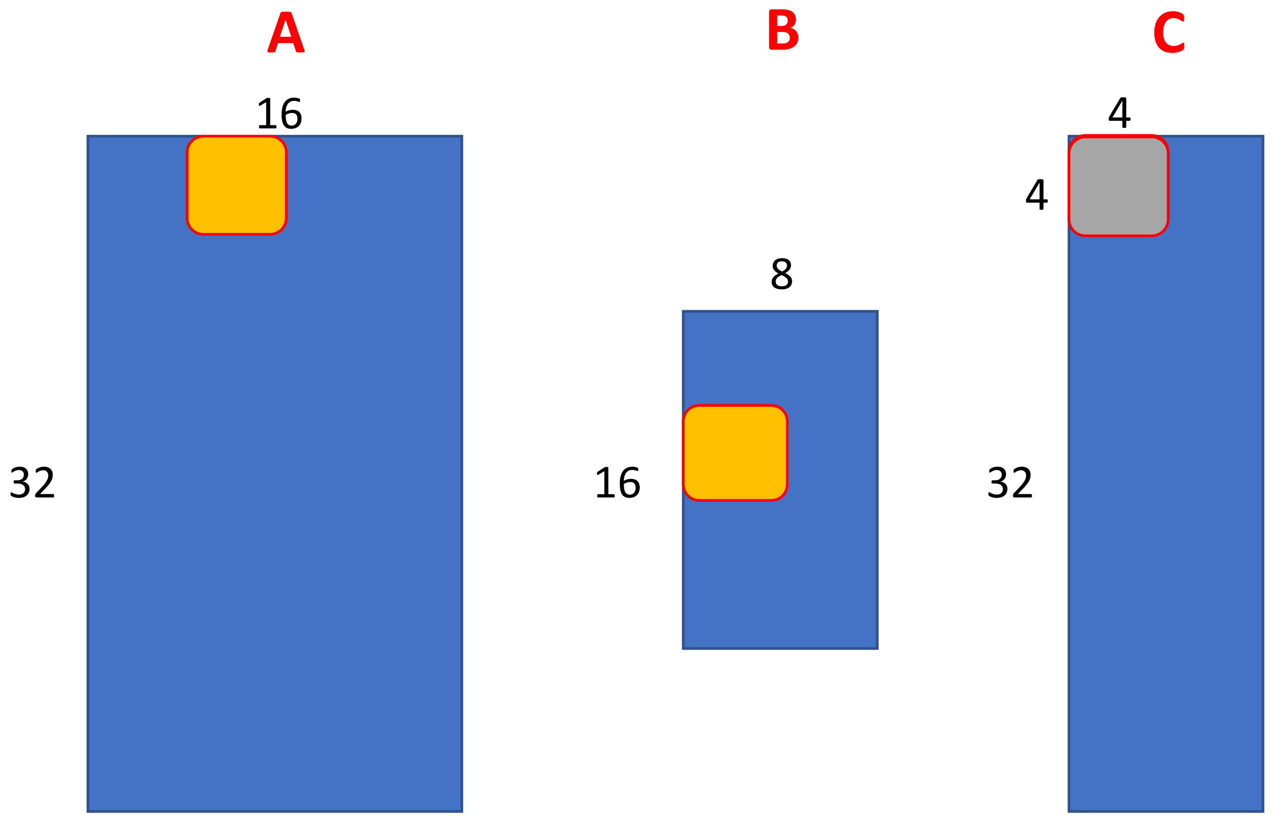
Pipelined Tile based Multiplications

- 64 multiplications + accumulation in parallel give a 4x4 partial sum
- 1st clock cycle



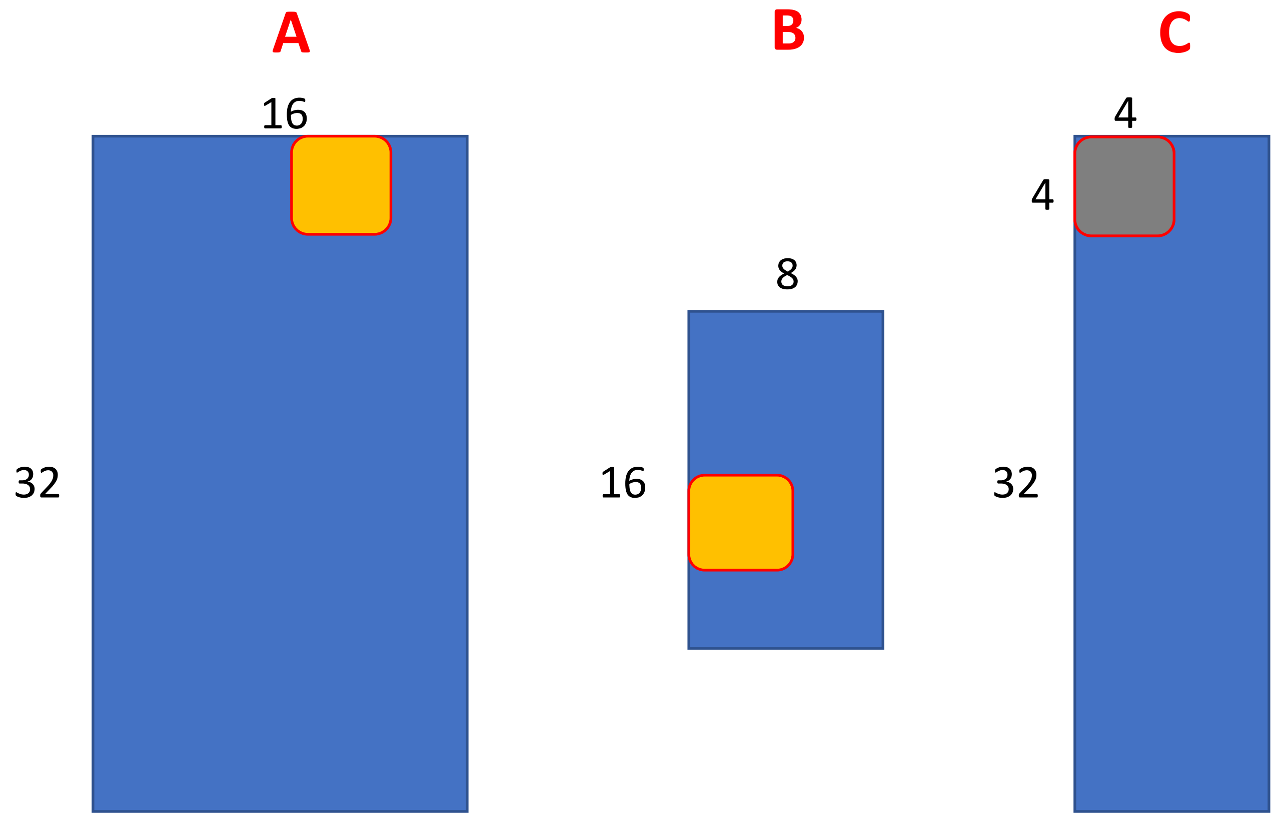
Pipelined Tile based Multiplications

- 64 multiplications + accumulation in parallel give a 4x4 partial sum
- 2nd clock cycle



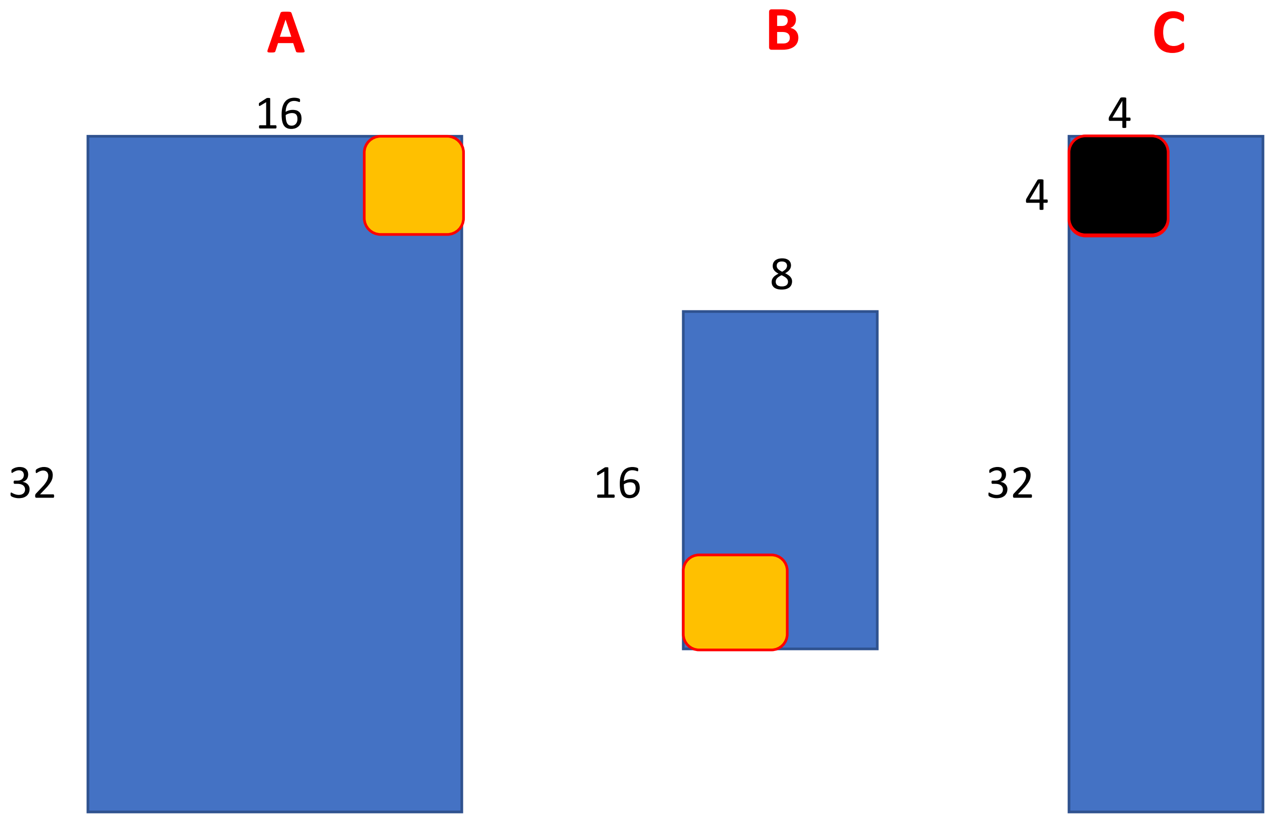
Pipelined Tile based Multiplications

- 64 multiplications + accumulation in parallel give a 4x4 partial sum
- 3rd clock cycle



Pipelined Tile based Multiplications

- 64 multiplications + accumulation in parallel give a 4x4 partial sum
- 4^{th} clock cycle



TODO – Lowering

```python
def conv2d(inputs, weights, padding, tiling, tile_size, bias=None):
    o_chn, i_chn, kernel_size, _ = weights.size()
    bs, i_chn, res, _ = inputs.size()

    # Lower Weights
    weights_transformed = weight_lowering()

    # Lower Inputs
    inputs_transformed = inputs_lowering()

    # Compute Outputs
    if tiling == False:
        lowered_outputs = weights_transformed @ inputs_transformed
    else:
        lowered_outputs = mmul_tiling(weights_transformed, inputs_transformed, tile_size)

    # Lift Outputs
    outputs = outputs_lifting()

    return outputs
```

TODO – Lowering

```python
def weight_lowering():
        ## TODO ##
        ## Hint: Use torch.reshape ##
        return lowered_weights
```

```python
def outputs_lifting():
        ## TODO ##
        ## Hint: Use torch.reshape & torch.transpose ##
        return outputs
```

- Complete `weight_lowering & outputs_lifting`
  - lowering : Transform a 4D Tensor into a 2D one.
  - lifting : Transform a 2D Tensor into a 4D one.
  - Use `torch.reshape & torch.transpose`

TODO – Tiling

- Complete `mmul_tiling` function
  - performs matA @ matB.
  - Use multi-level for loop.
  - Use `tmul` for multiplication
  - ex) `tileC += tmul(tileA, tileB, t)`

```
def tmul(tileA, tileB, t):

    # Check if the input dimension <= tile_siz
e
    assert tileA.size(0) <= t
    assert tileA.size(1) <= t
    assert tileB.size(1) <= t

    return tileA @ tileB
```

```
def mmul_tiling(matA, matB, t):
    a, c = matA.size()
    _, b = matB.size()
    matC = torch.zeros(a, b)

    ## TODO ##
    # Hint: Design a 3-level for loop

    return matC
```

TODO – Verification

- Lowering Test

```
# Lowering Test
lowering = True
tiling = False

layer.set_mode(lowering, tiling)

inputs = torch.randn(BS, I_CHN, RES_Y, RES_X)
layer.lowering_test(inputs)
```

```
Input size:        torch.Size([8, 32, 32, 32])
Weight size:       torch.Size([64, 32, 3, 3])
Output size:       torch.Size([8, 64, 32, 32])
=============================================
Correctness:       True
```

- Tiling Test

```
# Tiling test
lowering = True
tiling = True
tile_size = 32

layer.set_mode(lowering, tiling, tile_size)

inputs = torch.randn(BS, I_CHN, RES_Y, RES_X)
layer.lowering_test(inputs)
```

```
Input size:        torch.Size([8, 32, 32, 32])
Weight size:       torch.Size([64, 32, 3, 3])
Output size:       torch.Size([8, 64, 32, 32])
=============================================
Correctness:       True
```