Advanced PyTorch

YOU WERE HERE

YOU ARE HERE

YOU ARE GOING HERE

Topics

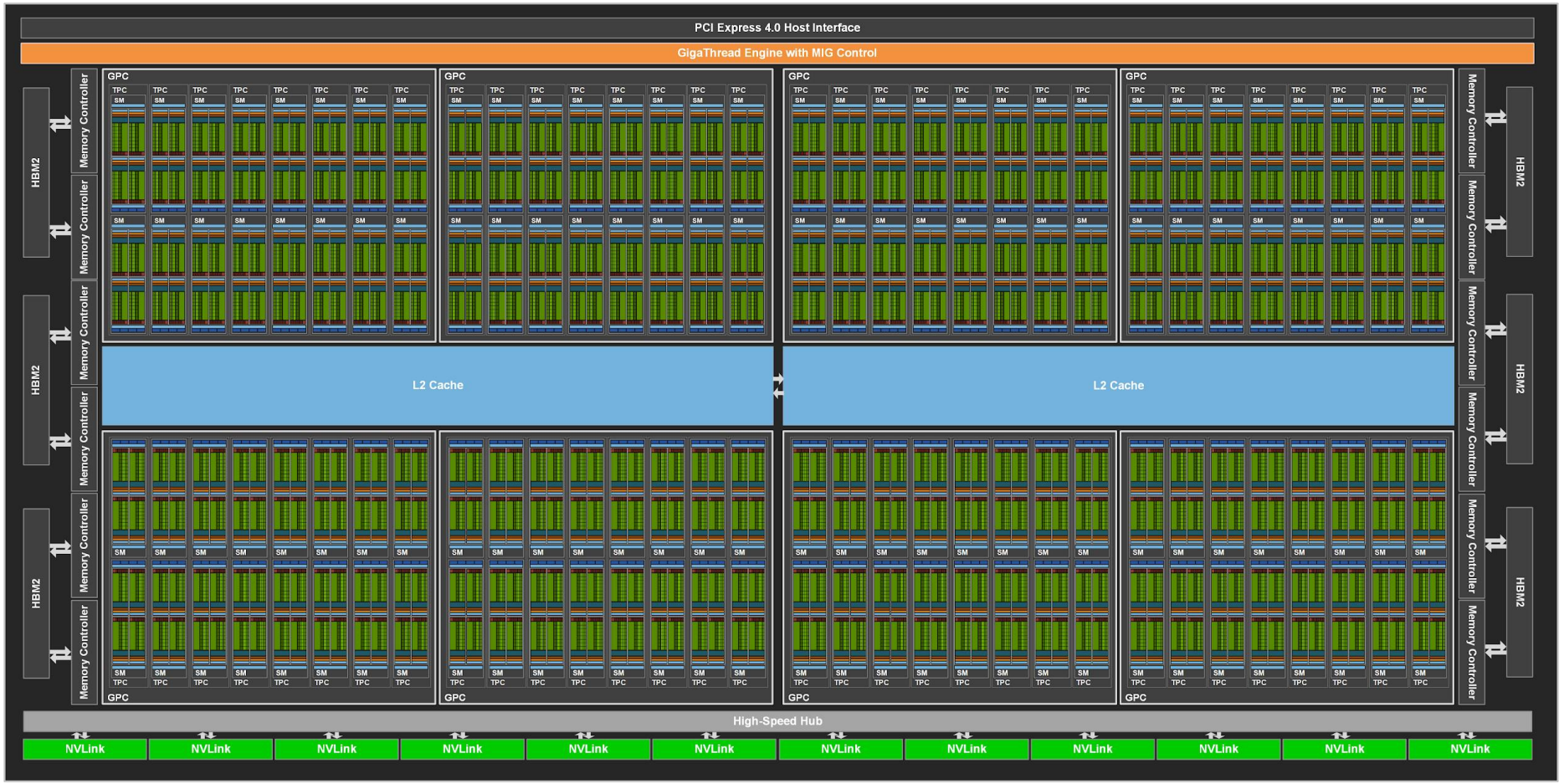
- Vectorization.
- Convolution-like operations.
- Data loading and handling.
- Intermediate results and hooks. (?)

Topics

- **Vectorization.**
- Convolution-like operations.
- Data loading and handling.
- Intermediate results and hooks. (?)

Vectorization

NVIDIA TESLA A100



Vectorization

- GPUs have many small “cores”.
- Many operations can run in parallel (asynchronously).
- Vectorized operations utilize this design.

Elementwise Operations

- `+, -, *, /, min, max, ...`

```
c = a + b
d = a - b
...
```

Linear Algebra

- `torch.matmul / @`

```
mat = torch.rand(size=(N, M))          # N×M
vec = torch.rand(size=(M,))            # M
out = mat @ vec                        # N
```

```
mat1 = torch.rand(size=(N, K))         # N×K
mat2 = torch.rand(size=(K, M))         # K×M
out = mat1 @ mat2                      # N×M
```

```
bmat1 = torch.rand(size=(B, N, K))     # B×N×K
bmat2 = torch.rand(size=(B, K, M))     # B×K×M
out = bmat1 @ bmat2                    # B×N×M
```

Pro Tip: Broadcasting

- Some dimensions of argument may be redundant:
  - Add a single row to each row in a matrix.

```
a = torch.rand(size=(N, 1))          # N×1
b = torch.rand(size=(1, M))          # 1×M
out = a + b                          # N×M
```

  - Multiply a batch of matrices by a single matrix.

```
bmat1 = torch.rand(size=(B, N, K))   # B×N×K
mat2 = torch.rand(size=(K, M))       # K×M
out = bmat1 @ mat2                   # B×N×M
```

Advanced Tensor Multiplication

- Einstein summation convention (`torch.einsum`):
  - Each dimension in each operand has a letter.
  - Multiply over dimensions with the same letter.
  - Sum over dimensions which are not in output.

```
mat = torch.rand(size=(N, M))                # N×M
vec = torch.rand(size=(M,))                  # M

out = torch.einsum(          , mat, vec)     # N
```

Advanced Tensor Multiplication

- Batch matrix-multiplication with `torch.einsum`:

$$\mathbf{out}[b, n, m] = \sum_{k} \mathbf{mat1}[b, n, k] \cdot \mathbf{mat2}[b, k, m]$$

```
mat1 = torch.rand(size=(B, N, K))                    # B×N×K
mat2 = torch.rand(size=(B, K, M))                    # B×K×M

out = torch.einsum(                 , mat1, mat2)   # B×N×M
```

Advanced Tensor Multiplication

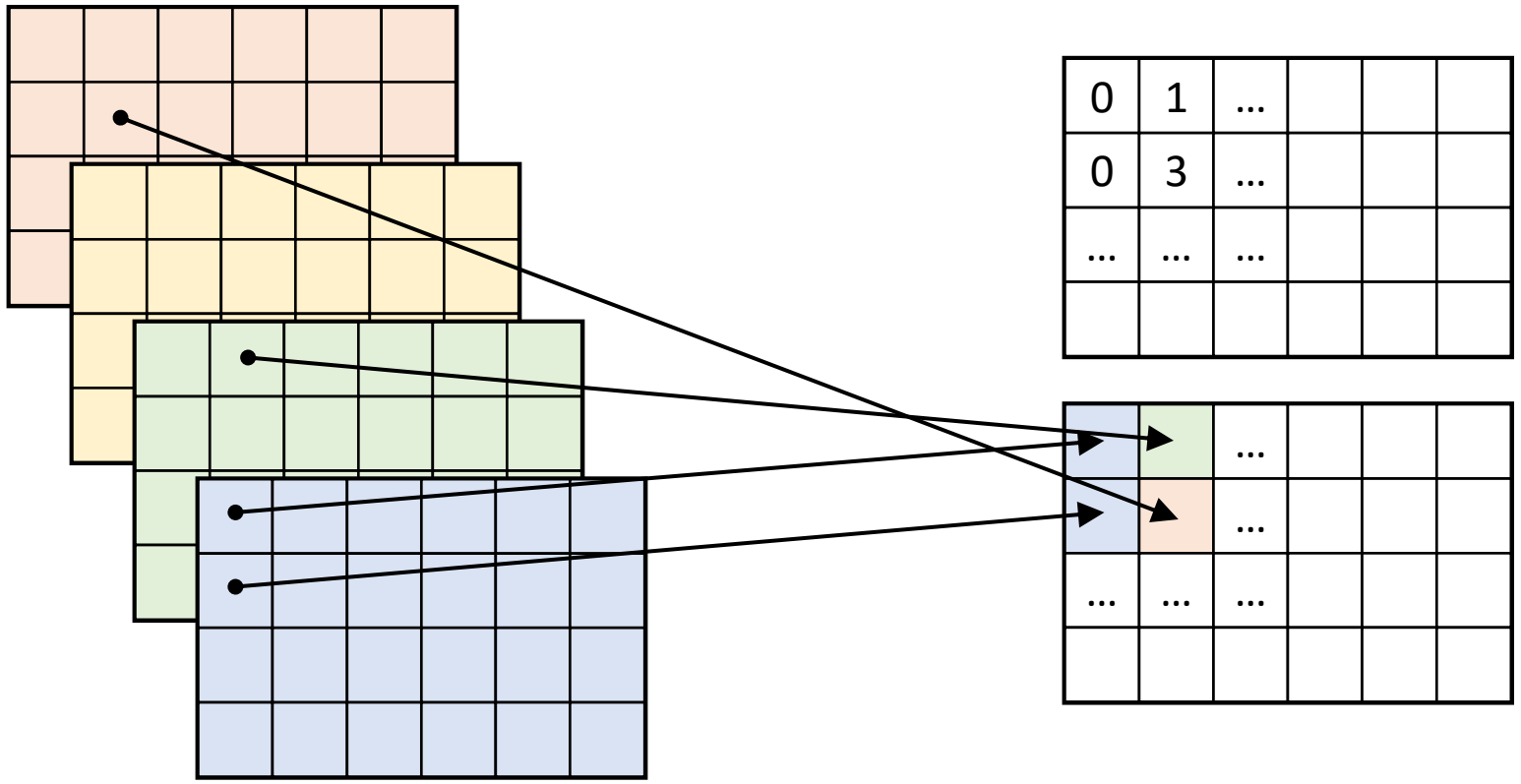
- More complex example:

$$\mathbf{out}[i,j,k] = \sum_{l}\sum_{m} \mathbf{x}[i,k,m,l,j] \cdot \mathbf{y}[i,l,j,k] \cdot \mathbf{z}[i]$$

```
x = torch.rand(size=(I, K, M, L, J))
y = torch.rand(size=(I, L, J, K))
z = torch.rand(size=(I,))

out = torch.einsum("ikmlj,iljk,i->ijk", x, y, z)
```

Gather

- Sample elements from a tensor according to an index.



Gather

- Sample elements from a tensor according to an index.

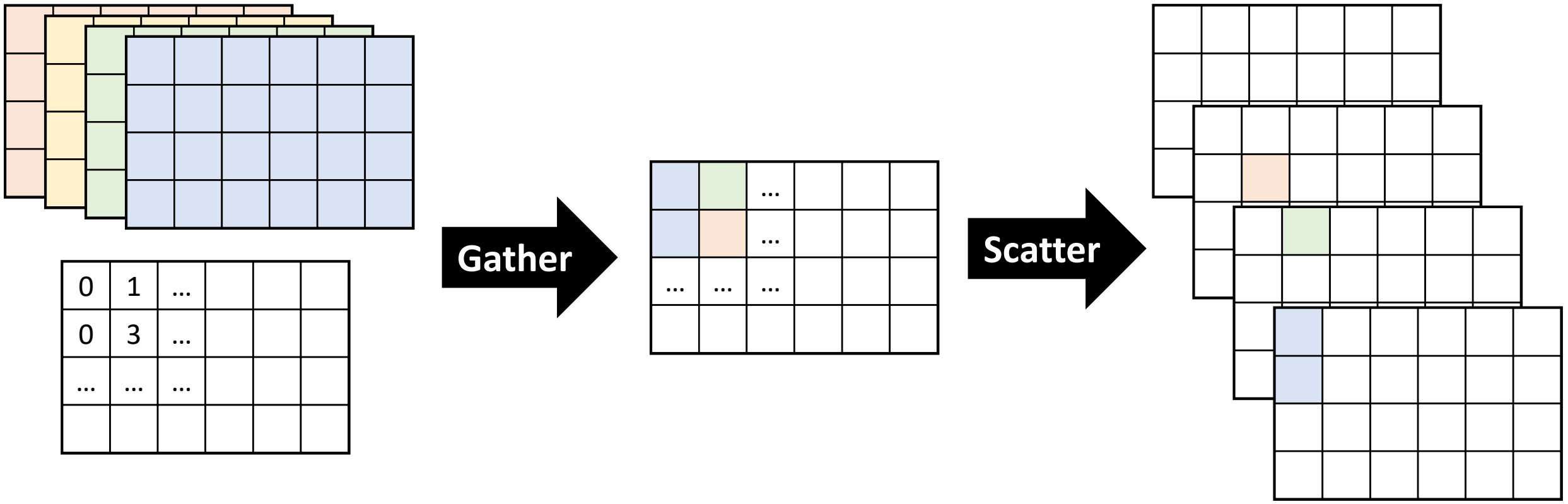
```
dim = 0
src = torch.rand(size=(4, 4, 6))
index = torch.randint(low=0, high=4, size=(1, 4, 6))
#index = torch.randint(low=0, high=src.size(dim), size=(num_samples, src.size(1), src.size(2))

# naive
out = src.zeros_like(size=(1, 4, 6))
for i in range(1):
  for j in range(4):
    for k in range(6):
      out[i, j, k] = src[index[i, j, k], j, k]

# vectorized
out = torch.gather(src, dim, index)
```

Gather

- Sample elements from a tensor according to an index.

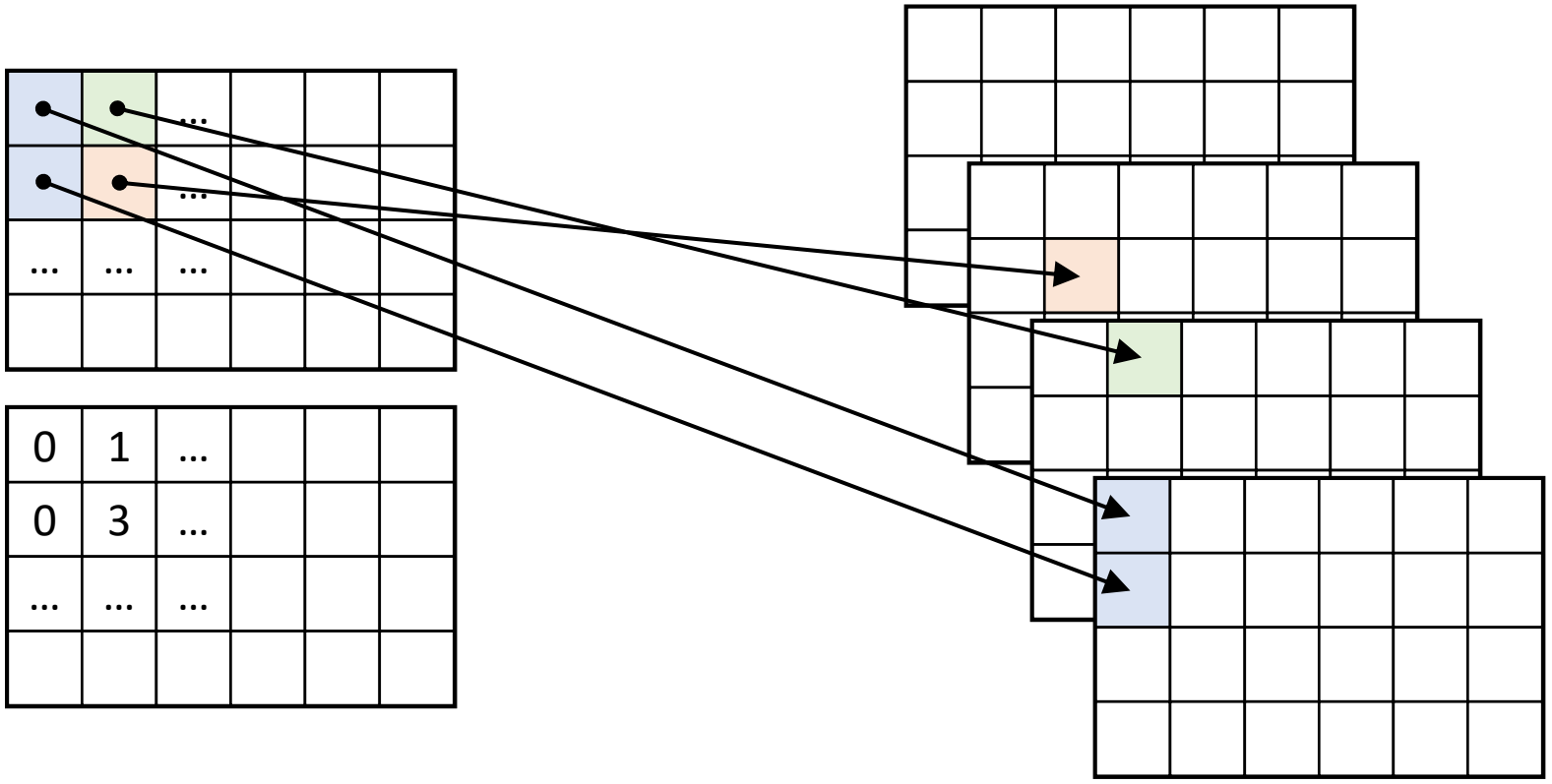
```
dim = 1
src = torch.rand(size=(4, 4, 6))
index = torch.randint(low=0, high=4, size=(4, 10, 6))
#index = torch.randint(low=0, high=src.size(dim), size=(src.size(0), num_samples, src.size(2))

# naive
out = src.zeros_like(size=(4, 10, 6))
for i in range(4):
  for j in range(10):
    for k in range(6):
      out[i, j, k] = src[i, index[i, j, k], k]

# vectorized
out = torch.gather(src, dim, index)
```

Gather and Scatter



Scatter (Add)

- The opposite (backward) of *Gather*.



Scatter (Add)

- The opposite (backward) of *Gather*.

```
dim = 0
src = torch.rand(size=(1, 4, 6))
index = torch.randint(low=0, high=4, size=(1, 4, 6))
#index = torch.randint(low=0, high=size, size=src.size())

# naive
out = src.zeros_like(size=(4, 4, 6))
for i in range(1):
  for j in range(4):
    for k in range(6):
      out[index[i, j, k], j, k] += src[i, j, k]

# vectorized in-place (assume out.shape == (4, 4, 6))
out.scatter_add_(dim, index, src)
```

Scatter (Add)

- The opposite (backward) of *Gather*.

```
dim = 1
src = torch.rand(size=(4, 10, 6))
index = torch.randint(low=0, high=4, size=(4, 10, 6))
#index = torch.randint(low=0, high=size, size=src.size())

# naive
out = src.zeros_like(size=(4, 4, 6))
for i in range(4):
  for j in range(10):
    for k in range(6):
      out[i, index[i, j, k], k] += src[i, j, k]

# vectorized in-place (assume out.shape == (4, 4, 6))
out.scatter_add_(dim, index, src)
```

Topics

- Vectorization.

- **Convolution-like operations.**

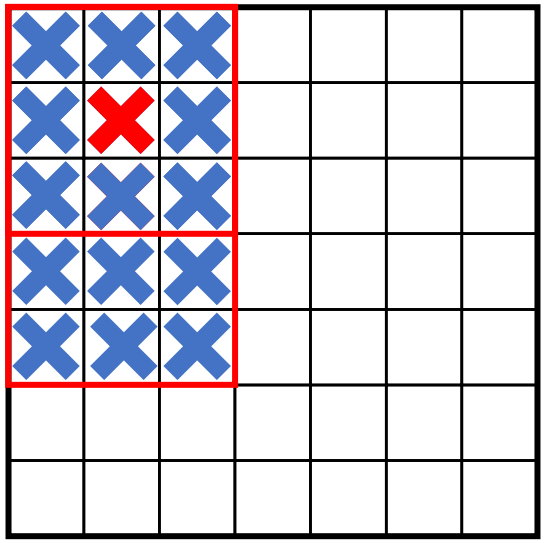
- Data loading and handling.

- Intermediate results and hooks. (?)

Quick Recap

- `kernel_size`
  - Number of elements in each block.

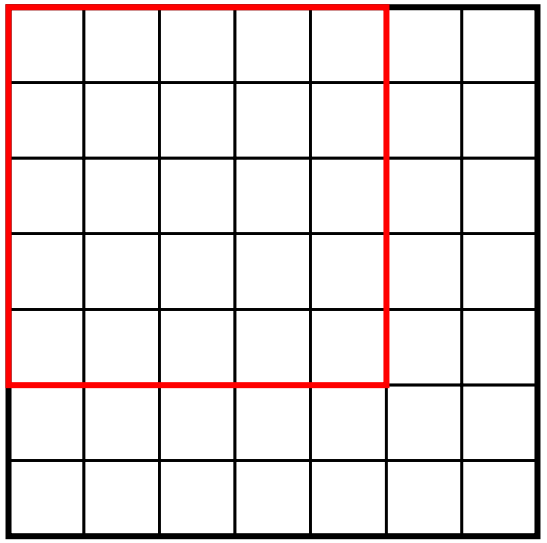
`kernel_size = (3,3)`



Quick Recap

- `kernel_size`
  - Number of elements in each block.
- `stride`
  - Distance between adjacent blocks.

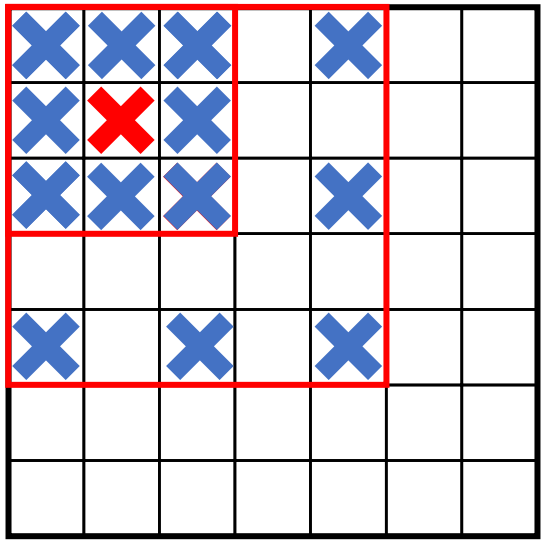
`stride = (2,2)`



Quick Recap

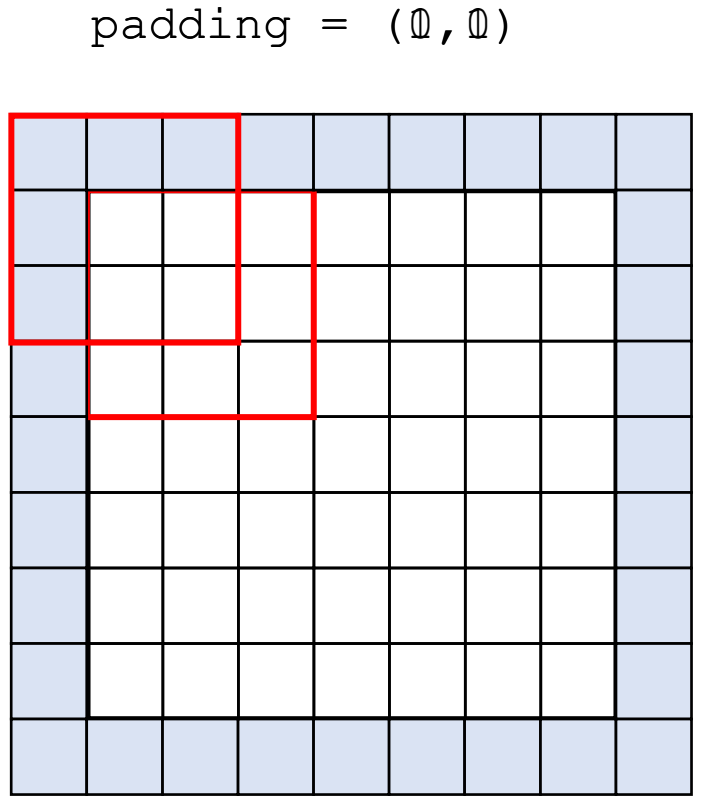
- `kernel_size`
  - Number of elements in each block.
- `stride`
  - Distance between adjacent blocks.
- `dilation`
  - Distance between adjacent elements in a block.

`dilation = (2,2)`



Quick Recap

- `kernel_size`
  - Number of elements in each block.
- `stride`
  - Distance between adjacent blocks.
- `dilation`
  - Distance between adjacent elements in a block.
- `padding`
  - Number of elements added at the borders of the image.

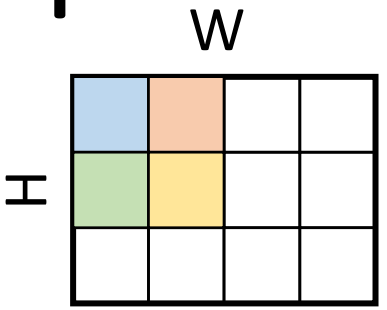


Example

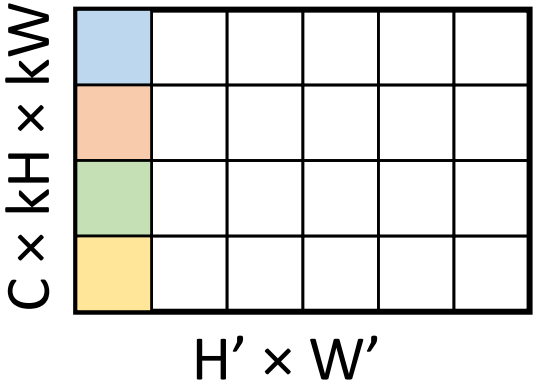
```
# Simple Conv2d layer
conv = nn.Conv2d(..., kernel_size=3)

# With stride
conv = nn.Conv2d(..., kernel_size=3, stride=2)

# With padding
conv = nn.Conv2d(..., kernel_size=3, padding=1)

# Different (H, W) dimensions
conv = nn.Conv2d(..., kernel_size=(3, 1), stride=(2, 1), padding=(1, 0))

# Simple MaxPool2d layer
Pool = nn.MaxPool2d(kernel_size=2, stride=2)
```

Implementation Convolution-like Ops

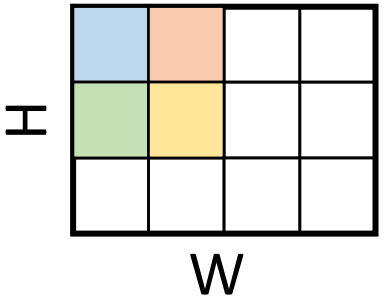
- Extract blocks from an image:
  - `F.unfold` (usually called “im2col”).



- Apply an operation on each block:
  - Linear, max pooling, etc.

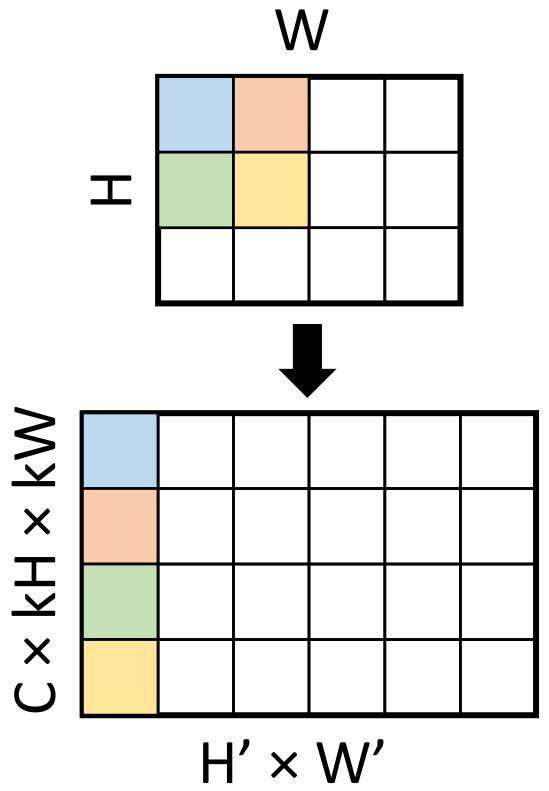


- Combine blocks into an image:
  - `F.fold` (usually called “col2im”).



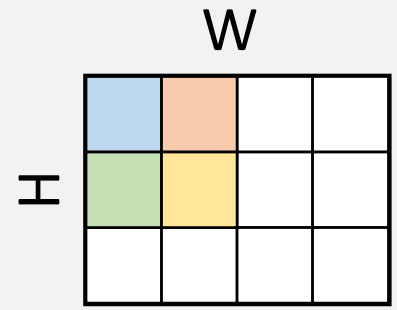
Using `F.unfold`

- Receives a batch of images:
  - Image shape: N, C, H, W
- Extract blocks:
  - Block size: C × kH × kW
  - Num of block per image: H’ × W’
- Return a batch of blocks:
  - Output shape: N, C × kH × kW, H’ × W’
  - Memory alignment: N, C, kH, kW, H’, W’



Example

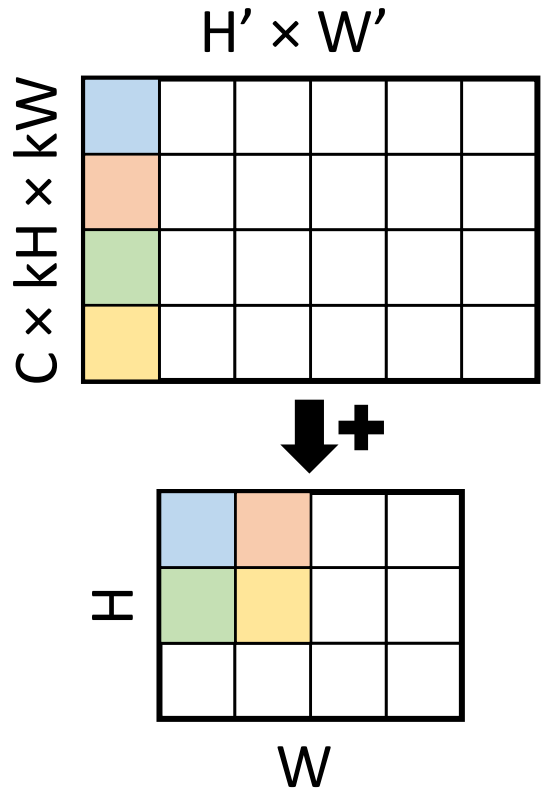
```
image = torch.rand(size=(2, 3, 3, 4))

# extract blocks
blocks = F.unfold(image, kernel_size=2)
# shape: (batch, block_size, num_blocks)
# blocks.shape == (2, 3 * 2 * 2, 2 * 3)

# view channels as a dimension
# shape: (batch, channels, kernel_size, num_blocks)
blocks = blocks.view(blocks.size(0), image.size(1), -1, blocks.size(-1))
# blocks.shape == (2, 3, 2 * 2, 2 * 3)
```



- What is affected by `kernel_size`?
- What is affected by: `stride`? `padding`?

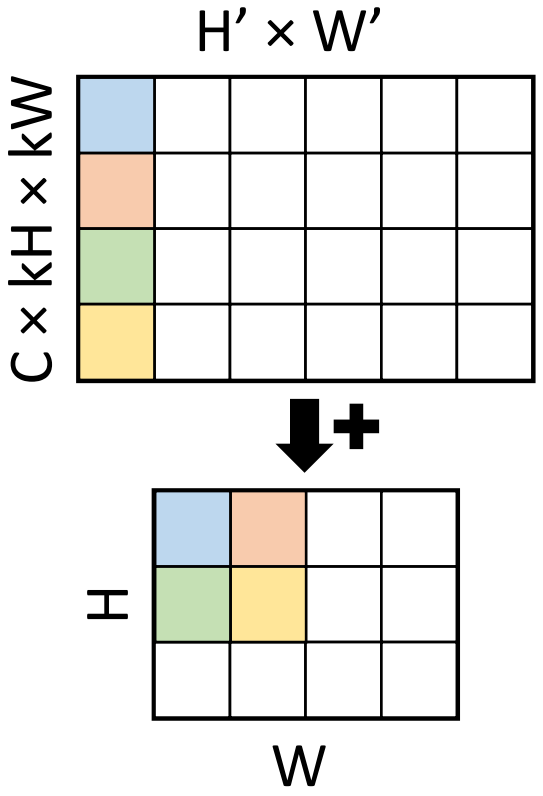
Using `F.fold`

- The opposite of `F.unfold`.
- Receives `blocks`.
- Receives `output_size`.
  - Creates an image `output` of that size.
  - Initialize `output` with zeros.
- Iterates over the blocks.
  - Adds each element to its place in `output`.

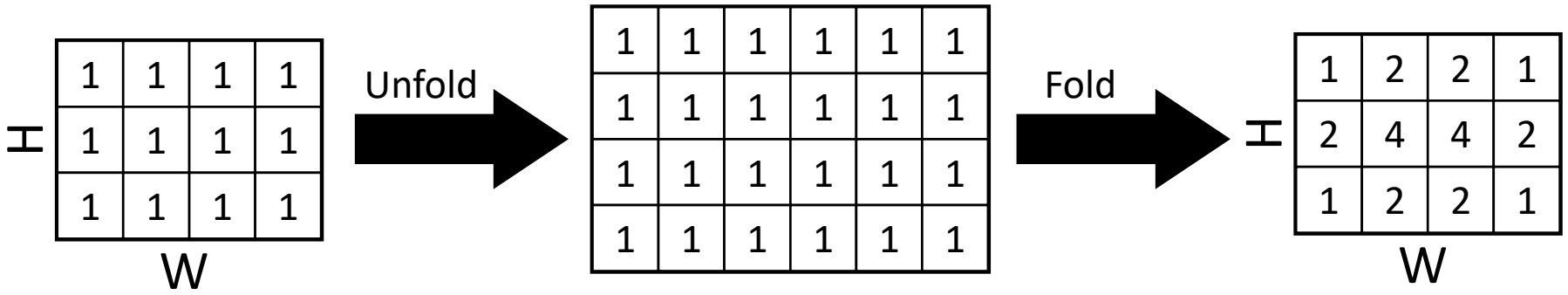


Example

```
blocks = torch.rand(size=(2, 3 * 2 * 2, 2 * 3))
output_size = (3, 4)

# fold back into an image
output = F.fold(blocks, output_size, kernel_size=2)
```



- What is the value in `output[1,1]`?



Topics

- Vectorization.
- Convolution-like operations.
- **Data loading and handling.**
- Intermediate results and hooks. (?)

Datasets and Dataloaders

- Datasets are collection of data samples.
  - Collection of images and labels.
  - Collection of pairs of images.

- Dataloaders help loading data from Datasets:
  - Create batches.
  - Support multi-processing.

Datasets

- Should implement `__len__` and `__getitem__`.
- Usually returns a tensor, a tuple of tensors or dict of tensors.

```
class MyDataset(Dataset):
  def __init__(self, img_paths, transform):
    self.img_paths = img_paths
    self.transform = transform

  def __len__(self):                  # == len(dataset)
    return len(self.img_paths)

  def __getitem__(self, index):  # == dataset[index]
    img = load_image(self.img_paths[index])
    img = self.transform(img)
    return {"img": img}
```

Dataloaders

- Receives a dataset and batch size.
- Returns an iterator.

```
transform = transforms.Compose([
  transforms.CenterCrop(256),
  transforms.ToTensor()
])
dataset = MyDataset(img_paths=["a.jpg", "b.jpg"], transform=transform)

dataloader = DataLoader(dataset, batch_size=32)

for batch in dataloader:
  img = batch["img"]
  # shape: 32,3,256,256
  # do something
```

Topics

- Vectorization.
- Convolution-like operations.
- Data loading and handling.
- **Intermediate results and hooks. (?)**

Accessing Intermediate Results

- Why would one access intermediate results?
  - Feature extraction.
  - Regularization.
  - Special loss.

Register a Hook

- A hook is function registered to the module.
- It's called by the module when the module is called.

```
def my_hook(module, input, output):
  # module: the module being hooked
  # input:  a tuple of inputs
  # output: a tuple of outputs
  print("hook!", input[0].shape, output[0].shape)

# register the hook
net.conv1.register_forward_hook(my_hook)

# use the hook
y = net(x)
# printed: "hook! torch.Size([...]) torch.Size([...])"
```

Remove a Hook

- A remove handle is returned during the registration.

```
# register the hook
handle = net.conv1.register_forward_hook(my_hook)

# remove the hook
handle.remove()

y = net(x)
# nothing is printed
```

Useful hooks

- Basic input/output recorder.
- Regularizer.

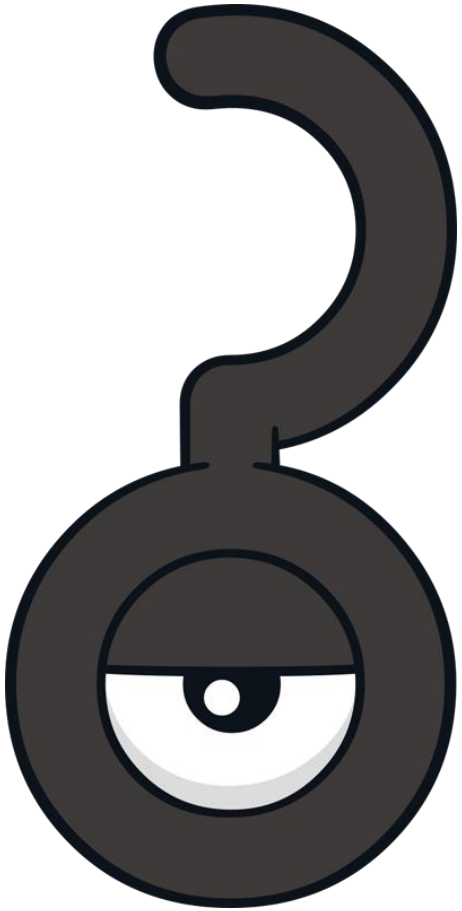
```
class Regularizer:
  def __init__(self, module):
    self.loss = None
    self._handle = module.register_forward_hook(self)

  def __del__(self):
    self._handle.remove()

  def __call__(self, input, output):
    self.loss = input[0].pow(2).mean()
```

Training and Inference

```python
def train_loop(dataloader, model, criterion, optimizer):
  size = len(dataloader.dataset)
  model.train()
  running_loss, running_corrects = 0, 0

  # iterate through all batches
  for batch, (X, y) in enumerate(dataloader):
    # move data to device
    X, y = X.to(device), y.to(device)
    # forward pass
    pred = model(X)
    loss = criterion(pred, y)
    # new gradients per batch
    optimizer.zero_grad()
    # backward pass
    loss.backward()
    # update gradients
    optimizer.step()

    running_loss += loss.item()
    running_corrects += (pred.argmax(1) == y).type(torch.float).sum().item()

  epoch_loss = running_loss / size
  epoch_accuracy = 100 * running_corrects / size
  return epoch_loss, epoch_accuracy
```

```python
def inference_loop(dataloader, model, criterion):
  size = len(dataloader.dataset)
  model.eval()
  running_loss, running_corrects = 0, 0

  # disregard gradients when not training
  with torch.no_grad():
    # iterate through all batches
    for X, y in dataloader:
      # move data to device
      X, y = X.to(device), y.to(device)
      # forward pass
      pred = model(X)
      # save data for evaluation measures (loss & accuracy)
      running_loss += criterion(pred, y).item()
      running_corrects += (pred.argmax(1) == y).type(torch.float).sum().item()

  epoch_loss = running_loss / size
  epoch_accuracy = 100 * running_corrects / size
  return epoch_loss, epoch_accuracy
```


Based on Pytorch tutorials



Next week:

Visualization and Understanding