```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.backends.cudnn as cudnn
import torchvision
import torchvision.transforms as transforms
import os, time, shutil
from models import *
```

```
# Device Selection
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
cudnn.benchmark = True
torch.backends.cudnn.fastest = True   # maximize RTX 3080 throughput
print(f"=> Using device: {device}")
```

```
=> Using device: cuda
```

```
# Training Parameters
batch_size = 128
epochs = 250
lr = 0.01
```

```
best_acc = 0
save_dir = "result/VGG16_quant"
os.makedirs(save_dir, exist_ok=True)
```

```
model = VGG16_quant().to(device)
if torch.cuda.device_count() > 1:
    print(f"=> Using {torch.cuda.device_count()} GPUs")
    model = nn.DataParallel(model)

criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9, weight_decay=5e-4)
#scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[25, 40], gamma=0.1)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)
scaler = torch.cuda.amp.GradScaler()

normalize = transforms.Normalize(mean=[0.491, 0.482, 0.447],
                                 std=[0.247, 0.243, 0.262])

train_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    normalize,
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    normalize,
])

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=train_transform)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=test_transform)

# train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
#                                            shuffle=True, num_workers=4, pin_memory=True)
# test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
#                                           shuffle=False, num_workers=4, pin_memory=True)

train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=256,              # 128 → 256 (fits in 10 GB easily)
    shuffle=True,
    num_workers=os.cpu_count(),  # use all CPU cores
    pin_memory=True,
    persistent_workers=True,
    prefetch_factor=4,           # overlap data loading with compute
)
test_loader = torch.utils.data.DataLoader(
    test_dataset,
    batch_size=512,              # eval can use larger batch
    shuffle=False,
    num_workers=os.cpu_count(),
    pin_memory=True,
    persistent_workers=True,
)

class AverageMeter:
    def __init__(self): self.reset()
    def reset(self): self.val=self.avg=self.sum=self.count=0
    def update(self, val, n=1):
        self.val = val; self.sum += val*n; self.count += n; self.avg = self.sum/self.count

def accuracy(output, target, topk=(1,)):
    maxk = max(topk)
    _, pred = output.topk(maxk, 1, True, True)
    pred = pred.t()
    correct = pred.eq(target.view(1, -1).expand_as(pred))
    res = []
    for k in topk:
        correct_k = correct[:k].reshape(-1).float().sum(0)
        res.append(correct_k.mul_(100.0 / target.size(0)))
    return res

def save_checkpoint(state, is_best, fdir):
    filepath = os.path.join(fdir, 'checkpoint.pth')
    torch.save(state, filepath)
    if is_best:
        shutil.copyfile(filepath, os.path.join(fdir, 'model_best.pth.tar'))


# def train(train_loader, model, criterion, optimizer, epoch):
#     model.train()
#     losses, top1 = AverageMeter(), AverageMeter()
#     start = time.time()

#     for i, (inputs, targets) in enumerate(train_loader):
#         inputs, targets = inputs.to(device, non_blocking=True), targets.to(device, non_blocking=True)

#         outputs = model(inputs)
#         loss = criterion(outputs, targets)

#         prec1 = accuracy(outputs, targets)[0]
#         losses.update(loss.item(), inputs.size(0))
#         top1.update(prec1.item(), inputs.size(0))

#         optimizer.zero_grad()
#         loss.backward()
#         optimizer.step()

#         if i % 100 == 0:
#             print(f"Epoch [{epoch}] [{i}/{len(train_loader)}] "
#                   f"Loss {losses.val:.4f} ({losses.avg:.4f})  "
#                   f"Acc {top1.val:.2f}% ({top1.avg:.2f}%)")

#     print(f" Epoch {epoch} done in {time.time()-start:.1f}s | Train Acc: {top1.avg:.2f}% | Loss: {losses.avg:.4f}")

def train(train_loader, model, criterion, optimizer, epoch):
    model.train()
    losses, top1 = AverageMeter(), AverageMeter()
    start = time.time()

    current_lr = optimizer.param_groups[0]['lr']

    for i, (inputs, targets) in enumerate(train_loader):
        inputs = inputs.to(device, non_blocking=True)
        targets = targets.to(device, non_blocking=True)

        optimizer.zero_grad(set_to_none=True)

        with torch.cuda.amp.autocast():
            outputs = model(inputs)
            loss = criterion(outputs, targets)

        # accuracy in full precision is fine (small overhead)
        prec1 = accuracy(outputs, targets)[0]
        losses.update(loss.item(), inputs.size(0))
        top1.update(prec1.item(), inputs.size(0))

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        if i % 100 == 0:
            print(
                f"Epoch [{epoch}] [{i}/{len(train_loader)}] "
                f"LR {current_lr:.5e}  "
                f"Loss {losses.val:.4f} ({losses.avg:.4f})  "
                f"Acc {top1.val:.2f}% ({top1.avg:.2f}%)"
            )

    print(
        f" Epoch {epoch} done in {time.time()-start:.1f}s | "
        f"LR: {current_lr:.5e} | Train Acc: {top1.avg:.2f}% | Loss: {losses.avg:.4f}"
    )

def validate(val_loader, model, criterion, epoch):
    model.eval()
    losses, top1 = AverageMeter(), AverageMeter()
    with torch.no_grad():
        for i, (inputs, targets) in enumerate(val_loader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            prec1 = accuracy(outputs, targets)[0]
            losses.update(loss.item(), inputs.size(0))
            top1.update(prec1.item(), inputs.size(0))
    print(f"Validation Epoch {epoch}: Acc {top1.avg:.2f}% | Loss {losses.avg:.4f}")
    return top1.avg
```

```
C:\Users\Nazih Bitar\AppData\Local\Temp\ipykernel_71228\487841380.py:10: FutureWarning: `torch.cuda.amp.GradScaler(args...)` is
  scaler = torch.cuda.amp.GradScaler()
```

```
# Training Loop
for epoch in range(1, epochs+1):
    train(train_loader, model, criterion, optimizer, epoch)
    val_acc = validate(test_loader, model, criterion, epoch)
    scheduler.step()

    is_best = val_acc > best_acc
    best_acc = max(val_acc, best_acc)

    save_checkpoint({
        'epoch': epoch,
        'state_dict': model.state_dict(),
        'best_acc': best_acc,
        'optimizer': optimizer.state_dict(),
    }, is_best, save_dir)

    print(f"Epoch {epoch} complete | Best Acc: {best_acc:.2f}%\n")

print("Training completed. Best accuracy: {:.2f}%".format(best_acc))
```

```
C:\Users\Nazih Bitar\AppData\Local\Temp\ipykernel_71228\487841380.py:116: FutureWarning: `torch.cuda.amp.autocast(args...)` is
  with torch.cuda.amp.autocast():
Epoch [1] [0/196] LR 1.00000e-02  Loss 2.5297 (2.5297)  Acc 10.94% (10.94%)
Epoch [1] [100/196] LR 1.00000e-02  Loss 1.5735 (1.7499)  Acc 41.80% (35.98%)
 Epoch 1 done in 159.2s | LR: 1.00000e-02 | Train Acc: 43.06% | Loss: 1.5654
Validation Epoch 1: Acc 52.04% | Loss 1.3803
Epoch 1 complete | Best Acc: 52.04%

Epoch [2] [0/196] LR 9.99938e-03  Loss 1.2129 (1.2129)  Acc 54.69% (54.69%)
Epoch [2] [100/196] LR 9.99938e-03  Loss 0.9911 (1.1802)  Acc 64.06% (57.64%)
 Epoch 2 done in 6.8s | LR: 9.99938e-03 | Train Acc: 59.78% | Loss: 1.1259
Validation Epoch 2: Acc 59.76% | Loss 1.1774
Epoch 2 complete | Best Acc: 59.76%

Epoch [3] [0/196] LR 9.99753e-03  Loss 0.9631 (0.9631)  Acc 65.62% (65.62%)
Epoch [3] [100/196] LR 9.99753e-03  Loss 0.8860 (0.9644)  Acc 67.19% (66.29%)
 Epoch 3 done in 6.8s | LR: 9.99753e-03 | Train Acc: 67.39% | Loss: 0.9310
Validation Epoch 3: Acc 66.55% | Loss 0.9770
Epoch 3 complete | Best Acc: 66.55%
Epoch [4] [0/196] LR 9.99445e-03  Loss 0.8209 (0.8209)  Acc 72.27% (72.27%)
Epoch [4] [100/196] LR 9.99445e-03  Loss 0.7914 (0.8242)  Acc 71.09% (70.82%)
 Epoch 4 done in 6.8s | LR: 9.99445e-03 | Train Acc: 71.51% | Loss: 0.8123
Validation Epoch 4: Acc 70.19% | Loss 0.8685
Epoch 4 complete | Best Acc: 70.19%

Epoch [5] [0/196] LR 9.99013e-03  Loss 0.6458 (0.6458)  Acc 78.91% (78.91%)
Epoch [5] [100/196] LR 9.99013e-03  Loss 0.7250 (0.7507)  Acc 74.61% (73.69%)
 Epoch 5 done in 6.7s | LR: 9.99013e-03 | Train Acc: 74.02% | Loss: 0.7408
Validation Epoch 5: Acc 72.78% | Loss 0.7853
Epoch 5 complete | Best Acc: 72.78%

Epoch [6] [0/196] LR 9.98459e-03  Loss 0.7144 (0.7144)  Acc 75.39% (75.39%)
Epoch [6] [100/196] LR 9.98459e-03  Loss 0.7054 (0.6834)  Acc 75.00% (75.94%)
 Epoch 6 done in 6.6s | LR: 9.98459e-03 | Train Acc: 76.47% | Loss: 0.6745
Validation Epoch 6: Acc 73.67% | Loss 0.7805
Epoch 6 complete | Best Acc: 73.67%

Epoch [7] [0/196] LR 9.97781e-03  Loss 0.5927 (0.5927)  Acc 77.73% (77.73%)
Epoch [7] [100/196] LR 9.97781e-03  Loss 0.6590 (0.6327)  Acc 79.30% (77.88%)
 Epoch 7 done in 6.7s | LR: 9.97781e-03 | Train Acc: 78.17% | Loss: 0.6276
Validation Epoch 7: Acc 69.19% | Loss 0.9573
Epoch 7 complete | Best Acc: 73.67%

Epoch [8] [0/196] LR 9.96980e-03  Loss 0.6022 (0.6022)  Acc 79.30% (79.30%)
Epoch [8] [100/196] LR 9.96980e-03  Loss 0.6839 (0.5925)  Acc 75.39% (79.45%)
 Epoch 8 done in 6.7s | LR: 9.96980e-03 | Train Acc: 79.54% | Loss: 0.5872
Validation Epoch 8: Acc 78.59% | Loss 0.6417
Epoch 8 complete | Best Acc: 78.59%

Epoch [9] [0/196] LR 9.96057e-03  Loss 0.5419 (0.5419)  Acc 80.08% (80.08%)
Epoch [9] [100/196] LR 9.96057e-03  Loss 0.5297 (0.5526)  Acc 80.08% (80.57%)
 Epoch 9 done in 6.7s | LR: 9.96057e-03 | Train Acc: 80.81% | Loss: 0.5498
Validation Epoch 9: Acc 74.35% | Loss 0.7656
Epoch 9 complete | Best Acc: 78.59%

Epoch [10] [0/196] LR 9.95012e-03  Loss 0.4059 (0.4059)  Acc 86.72% (86.72%)
```

```
PATH = "result/VGG16_quant/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
device = torch.device("cuda")

model.cuda()
model.eval()

test_loss = 0
correct = 0

with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device) # loading to GPU
        output = model(data)
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('\nTest set: Accuracy: {}/{} ({:.0f}%)\n'.format(
        correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

```
Test set: Accuracy: 9080/10000 (91%)
```

```
class SaveOutput:
    def __init__(self):
        self.outputs = []
    def __call__(self, module, module_in):
        self.outputs.append(module_in)
    def clear(self):
        self.outputs = []

######### Save inputs from selected layer ##########
save_output = SaveOutput()
i = 0

for layer in model.modules():
    i = i+1
    if isinstance(layer, QuantConv2d):
        print(i,"-th layer prehooked")
        layer.register_forward_pre_hook(save_output)
####################################################

dataiter = iter(test_loader)
images, labels = next(dataiter)
images = images.to(device)
out = model(images)
```

```
3 -th layer prehooked
7 -th layer prehooked
12 -th layer prehooked
16 -th layer prehooked
21 -th layer prehooked
25 -th layer prehooked
29 -th layer prehooked
34 -th layer prehooked
38 -th layer prehooked
42 -th layer prehooked
47 -th layer prehooked
51 -th layer prehooked
55 -th layer prehooked
```

```
weight_q = model.features[3].weight_q
w_alpha = model.features[3].weight_quant.wgt_alpha
w_bit = 4

weight_int = weight_q / (w_alpha / (2**(w_bit-1)-1))
print(weight_int)
```

```
tensor([[[[-1.0000,  0.0000, -5.0000],
          [ 0.0000, -2.0000, -3.0000],
          [-2.0000, -4.0000,  4.0000]],

         [[-3.0000,  3.0000,  2.0000],
          [ 4.0000,  2.0000, -3.0000],
          [-0.0000, -7.0000,  0.0000]],

         [[-4.0000,  6.0000, -1.0000],
          [ 7.0000, -2.0000, -7.0000],
          [-5.0000, -7.0000,  5.0000]],

         [[ 1.0000,  4.0000, -7.0000],
          [ 5.0000, -7.0000, -3.0000],
          [-3.0000, -2.0000,  7.0000]],

         [[-3.0000,  2.0000,  2.0000],
          [ 5.0000,  2.0000, -6.0000],
          [-2.0000, -7.0000,  3.0000]],

         [[ 2.0000,  1.0000, -4.0000],
          [ 3.0000, -7.0000, -3.0000],
          [-5.0000, -0.0000,  7.0000]],

         [[ 5.0000, -7.0000,  4.0000],
          [-6.0000, -5.0000,  7.0000],
          [ 1.0000,  7.0000, -4.0000]],

         [[-3.0000, -2.0000,  1.0000],
          [-5.0000,  2.0000, -1.0000],
          [ 2.0000, -1.0000, -4.0000]]],


        [[[ 1.0000,  0.0000, -0.0000],
          [-2.0000, -1.0000, -1.0000],
          [-1.0000, -3.0000, -4.0000]],

         [[ 1.0000, -0.0000,  2.0000],
          [ 7.0000,  5.0000, -0.0000],
          [ 1.0000,  7.0000,  3.0000]],

         [[-2.0000, -3.0000, -1.0000],
          [ 6.0000, -2.0000, -3.0000],
          [ 1.0000,  4.0000, -2.0000]],

         [[-1.0000,  0.0000,  1.0000],
          [-3.0000, -2.0000, -1.0000],
          [ 1.0000, -2.0000,  0.0000]],

         [[-1.0000,  1.0000, -0.0000],
          [-3.0000, -0.0000,  1.0000],
          [ 0.0000, -3.0000, -0.0000]],

         [[-2.0000,  5.0000,  4.0000],
          [ 6.0000,  7.0000,  7.0000],
          [ 2.0000,  7.0000,  7.0000]],
```

```
act = save_output.outputs[1][0]
act_alpha  = model.features[3].act_alpha
act_bit = 4
act_quant_fn = act_quantization(act_bit)

act_q = act_quant_fn(act, act_alpha)

act_int = act_q / (act_alpha / (2**act_bit-1))
print(act_int)
```

```
tensor([[[[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          ...,
          [ 0.0000,  2.0000,  3.0000,  ...,  2.0000,  3.0000,  6.0000],
          [ 0.0000,  1.0000,  2.0000,  ...,  1.0000,  3.0000,  4.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  1.0000,  2.0000]],

         [[ 3.0000,  3.0000,  2.0000,  ...,  1.0000,  2.0000,  3.0000],
          [ 1.0000,  1.0000,  1.0000,  ...,  0.0000,  0.0000,  2.0000],
          [ 1.0000,  0.0000,  1.0000,  ...,  0.0000,  0.0000,  2.0000],
          ...,
          [ 2.0000,  3.0000,  4.0000,  ...,  3.0000,  3.0000,  7.0000],
          [ 3.0000,  3.0000,  4.0000,  ...,  4.0000,  3.0000,  6.0000],
          [ 2.0000,  2.0000,  1.0000,  ...,  0.0000,  0.0000,  2.0000]],

         [[ 8.0000,  4.0000,  3.0000,  ...,  3.0000,  3.0000,  1.0000],
          [10.0000,  7.0000,  7.0000,  ...,  5.0000,  5.0000,  1.0000],
          [10.0000,  7.0000,  6.0000,  ...,  6.0000,  5.0000,  1.0000],
          ...,
          [ 0.0000,  3.0000,  3.0000,  ...,  4.0000,  2.0000, 12.0000],
          [ 0.0000,  2.0000,  2.0000,  ...,  3.0000,  2.0000,  8.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  2.0000]],

         ...,

         [[ 6.0000, 11.0000, 11.0000,  ...,  9.0000,  7.0000,  7.0000],
          [ 4.0000,  7.0000,  8.0000,  ...,  7.0000,  6.0000,  5.0000],
          [ 4.0000,  7.0000,  8.0000,  ...,  6.0000,  6.0000,  5.0000],
          ...,
          [ 2.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 2.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 4.0000,  5.0000,  3.0000,  ...,  3.0000,  4.0000,  2.0000]],

         [[ 3.0000,  3.0000,  4.0000,  ...,  2.0000,  2.0000,  7.0000],
          [ 5.0000,  4.0000,  4.0000,  ...,  3.0000,  4.0000,  4.0000],
          [ 5.0000,  4.0000,  3.0000,  ...,  4.0000,  4.0000,  4.0000],
          ...,
          [ 3.0000,  4.0000,  4.0000,  ...,  5.0000,  6.0000,  3.0000],
          [ 3.0000,  6.0000,  5.0000,  ...,  7.0000,  5.0000,  6.0000],
          [ 5.0000,  5.0000,  5.0000,  ...,  8.0000,  3.0000,  6.0000]],

         [[ 0.0000,  1.0000,  1.0000,  ...,  2.0000,  3.0000,  3.0000],
          [ 2.0000,  2.0000,  2.0000,  ...,  3.0000,  3.0000,  3.0000],
          [ 1.0000,  2.0000,  2.0000,  ...,  3.0000,  3.0000,  3.0000],
          ...,
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]]],


        [[[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 3.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 3.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          ...,
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]],
```

```
# Changed the code to mach the dimensions
conv_int = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3, padding=1)
conv_int.weight = torch.nn.parameter.Parameter(weight_int)
conv_int.bias = model.features[3].bias
output_int = conv_int(act_int)
```

```
output_recovered = output_int * (act_alpha / (2**act_bit-1)) * (w_alpha / (2**(w_bit-1)-1))
print(output_recovered)
```

```
tensor([[[[-6.4051e+00, -4.4886e+00, -6.8590e+00,  ..., -6.5564e+00,
           -4.9425e+00, -5.3964e+00],
          [-5.3964e+00, -6.9094e+00, -8.1703e+00,  ..., -6.2538e+00,
           -3.9843e+00, -5.1947e+00],
          [-4.5390e+00, -6.0520e+00, -4.7912e+00,  ..., -5.6486e+00,
           -5.5981e+00, -4.8416e+00],
          ...,
          [-7.1111e+00, -4.2364e+00, -4.9929e+00,  ..., -1.9669e+00,
           -1.2508e+01, -4.9425e+00],
          [-3.4799e+00, -3.2782e+00, -4.0851e+00,  ...,  2.9756e+00,
           -6.0016e+00,  3.0765e+00],
          [-3.4295e+00, -2.7234e+00, -2.7739e+00,  ..., -8.0190e+00,
           -3.6817e+00,  1.0087e-01]],

         [[ 1.2356e+01,  2.1434e+01,  2.3149e+01,  ...,  1.7349e+01,
            1.7299e+01,  1.2558e+01],
          [ 1.1953e+01,  1.8761e+01,  2.0880e+01,  ...,  1.4928e+01,
            1.4323e+01,  8.7755e+00],
          [ 8.8259e+00,  1.4374e+01,  1.7097e+01,  ...,  1.1801e+01,
            1.2810e+01,  7.3129e+00],
          ...,
          [ 3.4295e+00, -1.6139e+00, -1.5130e+00,  ...,  4.0347e-01,
           -3.8834e+00, -1.0591e+00],
          [ 7.1111e+00,  4.0347e+00,  2.2191e+00,  ...,  1.9669e+00,
            4.5390e-01, -1.7147e+00],
          [ 6.7077e+00,  5.7999e+00,  4.5895e+00,  ...,  3.9843e+00,
            2.5217e+00,  1.0087e-01]],

         [[-2.5217e+00,  2.0173e-01,  1.5634e+00,  ...,  2.1182e+00,
            4.7912e+00,  6.0520e+00],
          [-3.5808e+00,  1.7147e+00,  2.0678e+00,  ...,  5.1442e+00,
            7.7668e+00,  8.9772e+00],
          [-3.4799e+00,  1.1600e+00,  2.3199e+00,  ...,  4.2869e+00,
            7.3633e+00,  8.8259e+00],
          ...,
          [-2.4712e+00, -3.3286e+00, -4.4382e+00,  ..., -1.6643e+00,
           -1.7147e+00, -5.9007e+00],
          [-2.2695e+00, -2.0678e+00, -3.0260e+00,  ..., -2.0173e+00,
           -1.1095e+00, -3.4799e+00],
          [ 6.5564e-01,  7.0607e-01, -1.0087e-01,  ...,  1.7652e+00,
            7.5651e-01, -2.5217e-01]],

         ...,

         [[ 7.4137e+00,  4.6399e+00,  5.7494e+00,  ...,  3.2278e+00,
            3.0260e+00,  1.2608e+00],
          [ 1.3214e+01,  5.8503e+00,  5.4468e+00,  ...,  4.5390e+00,
            1.3617e+00, -3.5304e-01],
          [ 1.5937e+01,  9.8850e+00,  9.5320e+00,  ...,  8.5737e+00,
            3.4295e+00,  1.0087e-01],
          ...,
          [ 1.7551e+01,  2.7385e+01,  2.8192e+01,  ...,  2.5721e+01,
            2.9352e+01,  1.9165e+01],
          [ 1.7299e+01,  2.5066e+01,  2.4561e+01,  ...,  2.6932e+01,
            2.8596e+01,  1.9215e+01],
          [ 1.1095e+01,  1.5987e+01,  1.6694e+01,  ...,  1.4777e+01,
            1.7753e+01,  1.2255e+01]],
```

```
conv_ref = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3, padding=1)
conv_ref.weight = model.features[3].weight_q
conv_ref.bias = model.features[3].bias
output_ref = conv_ref(act)
print(output_ref)
```

```
tensor([[[[-6.8312e+00, -4.6004e+00, -6.6471e+00,  ..., -6.4628e+00,
           -4.5477e+00, -5.2338e+00],
          [-5.2520e+00, -6.6680e+00, -7.6366e+00,  ..., -5.7946e+00,
           -4.3476e+00, -5.5093e+00],
          [-4.4911e+00, -5.8073e+00, -5.7786e+00,  ..., -5.0214e+00,
           -5.6585e+00, -5.0362e+00],
          ...,
          [-7.2218e+00, -4.3894e+00, -5.1152e+00,  ..., -2.3635e+00,
           -1.1406e+01, -5.1236e+00],
          [-3.4618e+00, -4.1977e+00, -3.6684e+00,  ...,  2.7657e+00,
           -6.0028e+00,  2.7026e+00],
          [-3.6125e+00, -2.8890e+00, -2.9863e+00,  ..., -7.6201e+00,
           -3.8228e+00,  9.4766e-02]],

         [[ 1.1935e+01,  2.1229e+01,  2.2781e+01,  ...,  1.7140e+01,
            1.7372e+01,  1.2129e+01],
          [ 1.1960e+01,  1.8554e+01,  2.0840e+01,  ...,  1.5018e+01,
            1.4642e+01,  8.5652e+00],
          [ 8.9612e+00,  1.4826e+01,  1.7009e+01,  ...,  1.2316e+01,
            1.3610e+01,  7.5507e+00],
          ...,
          [ 3.6061e+00, -1.1549e+00, -1.2740e+00,  ...,  2.0906e-01,
           -3.8508e+00, -5.0640e-01],
          [ 6.9336e+00,  3.8924e+00,  2.3925e+00,  ...,  2.4351e+00,
            6.4529e-01, -1.4358e+00],
          [ 6.3803e+00,  5.4279e+00,  4.8181e+00,  ...,  4.2347e+00,
            2.8734e+00,  2.8294e-01]],

         [[-2.6868e+00, -5.6085e-02,  1.1637e+00,  ...,  2.2523e+00,
            5.1303e+00,  6.3522e+00],
          [-3.6583e+00,  1.3760e+00,  1.8128e+00,  ...,  5.1345e+00,
            8.3226e+00,  9.2070e+00],
          [-3.0431e+00,  1.4509e+00,  2.3351e+00,  ...,  3.6976e+00,
            6.9853e+00,  8.4996e+00],
          ...,
          [-2.2823e+00, -3.3199e+00, -4.5458e+00,  ..., -2.0034e+00,
           -1.9986e+00, -5.8240e+00],
          [-1.8909e+00, -2.0561e+00, -3.3767e+00,  ..., -1.9545e+00,
           -6.5327e-01, -2.8209e+00],
          [ 6.6080e-01,  4.5730e-01, -4.6869e-01,  ...,  1.7470e+00,
            1.1293e+00,  1.7601e-01]],

         ...,

         [[ 7.6590e+00,  4.5347e+00,  5.3635e+00,  ...,  3.3065e+00,
            2.9328e+00,  1.2734e+00],
          [ 1.3499e+01,  6.0717e+00,  5.9419e+00,  ...,  4.4116e+00,
            9.2363e-01, -5.8683e-01],
          [ 1.5846e+01,  9.7172e+00,  9.8443e+00,  ...,  8.6647e+00,
            3.2167e+00,  3.3241e-01],
          ...,
          [ 1.7715e+01,  2.8320e+01,  2.8665e+01,  ...,  2.5815e+01,
            2.9816e+01,  1.9409e+01],
          [ 1.7063e+01,  2.5237e+01,  2.5226e+01,  ...,  2.6991e+01,
            2.8527e+01,  1.8991e+01],
          [ 1.1112e+01,  1.6364e+01,  1.7212e+01,  ...,  1.4905e+01,
            1.7978e+01,  1.2740e+01]],
```

```python
# act_int.size = torch.Size([128, 64, 32, 32])  <- batch_size, input_ch, ni, nj
a_int = act_int[0,:,:,:]  # pick only one input out of batch
# a_int.size() = [64, 32, 32]

# conv_int.weight.size() = torch.Size([64, 64, 3, 3])  <- output_ch, input_ch, ki, kj
w_int = torch.reshape(weight_int, (weight_int.size(0), weight_int.size(1), -1))  # merge ki, kj index to kij
# w_int.weight.size() = torch.Size([64, 64, 9])

padding = 1
stride = 1
# He hard coded these values :(
#array_size = 64 # row and column number
array_size = w_int.size(0)

nig = range(a_int.size(1))  ## ni group [0,1,...31]
njg = range(a_int.size(2))  ## nj group

icg = range(int(w_int.size(1)))  ## input channel [0,...63]
ocg = range(int(w_int.size(0)))  ## output channel


kijg = range(w_int.size(2)) # [0, .. 8]
ki_dim = int(math.sqrt(w_int.size(2)))  ## Kernel's 1 dim size

######## Padding before Convolution #######
a_pad = torch.zeros(len(icg), len(nig)+padding*2, len(njg)+padding*2).cuda()
# a_pad.size() = [64, 32+2pad, 32+2pad]
a_pad[ :, padding:padding+len(nig), padding:padding+len(njg)] = a_int.cuda()
a_pad = torch.reshape(a_pad, (a_pad.size(0), -1))  ## mergin ni and nj index into nij
# a_pad.size() = [64, (32+2pad)*(32+2pad)]
```

```python
print(act_int.shape)
print(weight_int.shape)
```

```
torch.Size([512, 8, 32, 32])
torch.Size([8, 8, 3, 3])
```

```
###########################################

p_nijg = range(a_pad.size(1)) ## paded activation's nij group [0, ...34*34-1]

psum = torch.zeros( array_size, len(p_nijg), len(kijg)).cuda()

for kij in kijg:
    for nij in p_nijg:     # time domain, sequentially given input
        m = nn.Linear(array_size, array_size, bias=False)
        m.weight = torch.nn.Parameter(w_int[:,:,kij])
        psum[:, nij, kij] = m(a_pad[:,nij]).cuda()
```

```
import math

a_pad_ni_dim = int(math.sqrt(a_pad.size(1))) # 32 + 2*pad = 34

o_ni_dim = int((a_pad_ni_dim - (ki_dim- 1) - 1)/stride + 1) #34 - 2 - 1 + 1 = 32
o_nijg = range(o_ni_dim**2) # [0, 32*32-1]

out = torch.zeros(len(ocg), len(o_nijg)).cuda()


### SFP accumulation ###
for o_nij in o_nijg:
    for kij in kijg:  #[0, ... 8]
        out[:,o_nij] = out[:,o_nij] + \
        psum[:, int(o_nij/o_ni_dim)*a_pad_ni_dim + o_nij%o_ni_dim + int(kij/ki_dim)*a_pad_ni_dim + kij%ki_dim, kij]
                ## 2nd index = (int(o_nij/30)*32 + o_nij%30) + (int(kij/3)*32 + kij%3)
```

```
out_2D = torch.reshape(out, (out.size(0), o_ni_dim, -1)) # nij -> ni & nj
difference = (out_2D - output_int[0,:,:,:])
print(difference.abs().sum())
```

```
tensor(0.0353, device='cuda:0', grad_fn=<SumBackward0>)
```

```
output_int[0,:,:,:]
```

```
tensor([[[   0.,   -8.,    5.,  ...,   43.,   66.,   79.],
         [  26.,   -2.,    6.,  ...,    4.,   -7.,  -17.],
         [   2.,  -16.,  -13.,  ...,   -5.,  -17.,  -10.],
         ...,
         [ -34.,  -48.,  -39.,  ...,  -14.,  -32.,  -28.],
         [ -23.,  -26.,  -21.,  ...,  -12.,   -2.,   46.],
         [  13.,   25.,   37.,  ...,   71.,   41.,  -15.]],

        [[  20.,  -15.,    1.,  ...,    2.,    8.,   36.],
         [   4.,  -68.,  -54.,  ...,  -30.,  -25.,   23.],
         [  42.,  -21.,  -32.,  ...,  -16.,  -16.,   24.],
         ...,
         [  60.,  198.,  155.,  ...,  207.,  228.,  100.],
         [  80.,  214.,  197.,  ...,  237.,  234.,  147.],
         [  17.,   81.,   78.,  ...,  105.,   89.,   70.]],

        [[ -26.,   17.,  -16.,  ...,  -10.,   -4.,  142.],
         [ -52.,   -1.,  -11.,  ...,  -25.,  -44.,  166.],
         [ -31.,    5.,   -4.,  ...,  -14.,  -55.,  165.],
         ...,
         [ -99.,  -44.,   38.,  ..., -119.,    2.,  165.],
         [ -47.,  -99.,   27.,  ...,  -27., -141.,  219.],
         [ -27.,  -50.,  -32.,  ...,   26., -123.,  115.]],

        ...,

        [[ -20.,  -12.,  -25.,  ...,    1.,   55.,  -93.],
         [  -3.,  -24.,   -5.,  ...,   -5.,   30., -133.],
         [ -10.,  -14.,  -32.,  ...,   -9.,   21., -124.],
         ...,
         [  27.,   17.,   -9.,  ...,   38.,  -47.,  -95.],
         [  31.,    9.,   -2.,  ...,  -16.,   21., -142.],
         [  28.,   21.,  -15.,  ...,  -58.,  -19., -137.]],

        [[  19.,   14.,   12.,  ...,   26.,   38.,   -1.],
         [  57.,   64.,   61.,  ...,   87.,  100.,   16.],
         [  46.,   66.,   59.,  ...,   76.,  100.,   18.],
         ...,
         [  14.,    1.,   -4.,  ...,   31.,   14.,  -33.],
         [  -4.,   -5.,  -12.,  ...,    2.,   28.,  -38.],
         [   3.,   10.,   27.,  ...,   11.,   36.,  -21.]],

        [[  36.,  109.,   50.,  ...,   49.,   14.,  -15.],
         [  84.,  198.,  119.,  ...,  111.,   77.,   -6.],
         [  69.,  185.,  119.,  ...,   99.,   84.,    9.],
         ...,
         [-104., -124.,  -47.,  ...,   -1.,  -40.,   77.],
         [ -98., -104.,  -79.,  ...,  -37.,  -16.,   22.],
         [ -77.,  -72.,  -90.,  ...,  -87.,  -16.,  -57.]]], device='cuda:0',
       grad_fn=<SliceBackward0>)
```

```
######## Easier 2D version ########

import math

kig = range(int(math.sqrt(len(kijg))))
kjg = range(int(math.sqrt(len(kijg))))

o_nig = range(int((math.sqrt(len(nijg))+2*padding -(math.sqrt(len(kijg))- 1) - 1)/stride + 1))
o_njg = range(int((math.sqrt(len(nijg))+2*padding -(math.sqrt(len(kijg)) - 1) - 1)/stride + 1))


out = torch.zeros(len(ocg), len(o_nig), len(o_njg)).cuda()



### SFP accumulation ###
for ni in o_nig:
    for nj in o_njg:
        for ki in kig:
            for kj in kjg:
                for ic_tile in ic_tileg:
                    for oc_tile in oc_tileg:
                        out[oc_tile*array_size:(oc_tile+1)*array_size, ni, nj] = out[oc_tile*array_size:(oc_tile+1)*array_size,
                        psum[ic_tile, oc_tile, :, int(math.sqrt(len(nijg)))*(ni+ki) + (nj+kj), len(kig)*ki+kj]
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```