ResNet20_Quant_Pruned_Tiled

December 6, 2024

[1]:
```python
import argparse
import os
import time
import shutil

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.backends.cudnn as cudnn

from tensorboardX import SummaryWriter

import torchvision
import torchvision.transforms as transforms

from models import *

global best_prec
use_gpu = torch.cuda.is_available()
print('=> Building model...')


batch_size = 128
model_name = "resnet20_80prune_4tiles"
model = resnet20_quant()
print(model)

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


train_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=True,
    download=True,
    transform=transforms.Compose([
```

```
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize,
    ]))
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=2)


test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=False,
    download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        normalize,
    ]))

testloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,␣
 ↪shuffle=False, num_workers=2)


print_freq = 100 # every 100 batches, accuracy printed. Here, each batch␣
 ↪includes "batch_size" data points
# CIFAR10 has 50,000 training data, and 10,000 validation data.

def train(trainloader, model, criterion, optimizer, epoch):
    batch_time = AverageMeter()
    data_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    model.train()

    end = time.time()
    for i, (input, target) in enumerate(trainloader):
        # measure data loading time
        data_time.update(time.time() - end)

        input, target = input.cuda(), target.cuda()

        # compute output
        output = model(input)
        loss = criterion(output, target)

        # measure accuracy and record loss
        prec = accuracy(output, target)[0]
```

```python
        losses.update(loss.item(), input.size(0))
        top1.update(prec.item(), input.size(0))

        # compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # measure elapsed time
        batch_time.update(time.time() - end)
        end = time.time()


        if i % print_freq == 0:
            print('Epoch: [{0}][{1}/{2}]\t'
                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
                  'Data {data_time.val:.3f} ({data_time.avg:.3f})\t'
                  'Loss {loss.val:.4f} ({loss.avg:.4f})\t'
                  'Prec {top1.val:.3f}% ({top1.avg:.3f}%)'.format(
                   epoch, i, len(trainloader), batch_time=batch_time,
                   data_time=data_time, loss=losses, top1=top1))



def validate(val_loader, model, criterion ):
    batch_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    # switch to evaluate mode
    model.eval()

    end = time.time()
    with torch.no_grad():
        for i, (input, target) in enumerate(val_loader):

            input, target = input.cuda(), target.cuda()

            # compute output
            output = model(input)
            loss = criterion(output, target)

            # measure accuracy and record loss
            prec = accuracy(output, target)[0]
            losses.update(loss.item(), input.size(0))
            top1.update(prec.item(), input.size(0))
```

```
            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()

            if i % print_freq == 0:  # This line shows how frequently print out␣
 ↪the status. e.g., i%5 => every 5 batch, prints out
                print('Test: [{0}/{1}]\t'
                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
                  'Loss {loss.val:.4f} ({loss.avg:.4f})\t'
                  'Prec {top1.val:.3f}% ({top1.avg:.3f}%)'.format(
                   i, len(val_loader), batch_time=batch_time, loss=losses,
                   top1=top1))

    print(' * Prec {top1.avg:.3f}% '.format(top1=top1))
    return top1.avg


def accuracy(output, target, topk=(1,)):
    """Computes the precision@k for the specified values of k"""
    maxk = max(topk)
    batch_size = target.size(0)

    _, 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].view(-1).float().sum(0)
        res.append(correct_k.mul_(100.0 / batch_size))
    return res


class AverageMeter(object):
    """Computes and stores the average and current value"""
    def __init__(self):
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
```

```python
        self.count += n
        self.avg = self.sum / self.count


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 adjust_learning_rate(optimizer, epoch):
    """For resnet, the lr starts from 0.1, and is divided by 10 at 80 and 120␣
 ↪epochs"""
    adjust_list = [ 25, 35, 45]
    if epoch in adjust_list:
        for param_group in optimizer.param_groups:
            param_group['lr'] = param_group['lr'] * 0.1
```

```
=> Building model…
ResNet_Cifar(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
  (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
```

```
      (conv2): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): QuantConv2d(
        16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): QuantConv2d(
          16, 32, kernel_size=(1, 1), stride=(2, 2), bias=False
          (weight_quant): weight_quantize_fn()
        )
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      )
    )
    (1): BasicBlock(
      (conv1): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): QuantConv2d(
        32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      (downsample): Sequential(
        (0): QuantConv2d(
          32, 64, kernel_size=(1, 1), stride=(2, 2), bias=False
          (weight_quant): weight_quantize_fn()
        )
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AvgPool2d(kernel_size=8, stride=1, padding=0)
  (fc): Linear(in_features=64, out_features=10, bias=True)
)
Files already downloaded and verified
Files already downloaded and verified
```

[ ]:
```python
lr = 1e-2
weight_decay = 4e-4
epochs = 60
best_prec = 0

model = model.cuda()
criterion = nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,␣
 ↪weight_decay=weight_decay)
# weight decay: for regularization to prevent overfitting

if not os.path.exists('result'):
    os.makedirs('result')
fdir = 'result/'+str(model_name)
if not os.path.exists(fdir):
    os.makedirs(fdir)

for epoch in range(0, epochs):
    adjust_learning_rate(optimizer, epoch)

    train(trainloader, model, criterion, optimizer, epoch)

    # evaluate on test set
    print("Validation starts")
    prec = validate(testloader, model, criterion)

    # remember best precision and save checkpoint
    is_best = prec > best_prec
    best_prec = max(prec,best_prec)
    print('best acc: {:1f}'.format(best_prec))
    save_checkpoint({
        'epoch': epoch + 1,
        'state_dict': model.state_dict(),
        'best_prec': best_prec,
        'optimizer': optimizer.state_dict(),
    }, is_best, fdir)
```

[2]:
```python
PATH = "result/resnet20_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 testloader:
        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(testloader.dataset)

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

```
Test set: Accuracy: 9058/10000 (91%)
```

[3]:
```
#### Prune all the QuantConv2D layers' 80% weights with 1) unstructured, and 2)␣
 ↪structured manner.
############################################################
####################### UNSTRUCTURED #######################
############################################################
import torch.nn.utils.prune as prune

for layer in model.modules():
    if isinstance(layer, QuantConv2d):
        prune.l1_unstructured(layer, name='weight', amount=0.8)
        ### Check sparsity ###
        mask = layer.weight_mask
        sparsity_mask = (mask == 0).sum() / mask.nelement()
        print("Sparsity level: ", sparsity_mask)
```

```
Sparsity level:  tensor(0.7999, device='cuda:0')
Sparsity level:  tensor(0.7999, device='cuda:0')
Sparsity level:  tensor(0.7999, device='cuda:0')
Sparsity level:  tensor(0.7999, device='cuda:0')
Sparsity level:  tensor(0.7999, device='cuda:0')
Sparsity level:  tensor(0.7999, device='cuda:0')
Sparsity level:  tensor(0.7999, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
Sparsity level:  tensor(0.8008, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
```

```
Sparsity level:  tensor(0.8000, device='cuda:0')
Sparsity level:  tensor(0.7998, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
Sparsity level:  tensor(0.8000, device='cuda:0')
```

[4]:
```
## check accuracy after pruning
model.cuda()
model.eval()

test_loss = 0
correct = 0

with torch.no_grad():
    for data, target in testloader:
        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(testloader.dataset)

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

```
Test set: Accuracy: 1000/10000 (10%)
```

[ ]:
```
## Start finetuning (training here), and see how much you can recover your␣
 ↪accuracy ##
## You can change hyper parameters such as epochs or lr ##
lr = 1e-1
weight_decay = 1e-4
epochs = 60
best_prec = 0

model = model.cuda()
criterion = nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,␣
 ↪weight_decay=weight_decay)
# weight decay: for regularization to prevent overfitting

if not os.path.exists('result'):
    os.makedirs('result')
```

```
fdir = 'result/'+str(model_name)+str('_unstructured')

if not os.path.exists(fdir):
    os.makedirs(fdir)

for epoch in range(0, epochs):
    adjust_learning_rate(optimizer, epoch)

    train(trainloader, model, criterion, optimizer, epoch)

    # evaluate on test set
    print("Validation starts")
    prec = validate(testloader, model, criterion)

    # remember best precision and save checkpoint
    is_best = prec > best_prec
    best_prec = max(prec,best_prec)
    print('best acc: {:1f}'.format(best_prec))
    save_checkpoint({
        'epoch': epoch + 1,
        'state_dict': model.state_dict(),
        'best_prec': best_prec,
        'optimizer': optimizer.state_dict(),
    }, is_best, fdir)
```

[5]:

```
## check your accuracy again after finetuning
PATH = "result/resnet20_80prune_4tiles_unstructured/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])

model.cuda()
model.eval()

test_loss = 0
correct = 0

with torch.no_grad():
    for data, target in testloader:
        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(testloader.dataset)

print('\nTest set: Accuracy: {}/{} ({:.0f}%)\n'.format(
```

```
        correct, len(testloader.dataset),
        100. * correct / len(testloader.dataset)))
```

```
Test set: Accuracy: 8915/10000 (89%)
```

[6]:
```
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(layer,"-th layer prehooked")
        layer.register_forward_pre_hook(save_output)
####################################################

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

[ ]:
```
w_bit = 4
weight_q = model.layer1[2].conv1.weight_q # quantized value is stored during␣
 ↪the training
w_alpha = model.layer1[2].conv1.weight_quant.wgt_alpha   # alpha is defined in␣
 ↪your model already. bring it out here
w_delta = w_alpha / (2**(w_bit - 1) - 1)    # delta can be calculated by using␣
 ↪alpha and w_bit
weight_int = weight_q / w_delta # w_int can be calculated by weight_q and␣
 ↪w_delta
print(weight_int[:2, :2, :2]) # you should see clean integer numbers
```

[8]:
```
#### check your sparsity for weight_int is near 80% #####
#### Your sparsity could be >80% after quantization #####
for layer in model.modules():
    if isinstance(layer, QuantConv2d):
        #prune.remove(layer, 'weight')
```

```
        weight_q_temp = layer.weight_q
        w_alpha_temp = layer.weight_quant.wgt_alpha
        w_delta_temp = w_alpha_temp /(2**(w_bit-1)-1)
        weight_int_temp = weight_q_temp / w_delta_temp
        sparsity_weight_int = (weight_int_temp == 0).sum() / weight_int_temp.
↪nelement()
        print("Sparsity level: ", sparsity_weight_int)
```

```
Sparsity level:  tensor(0.8095, device='cuda:0')
Sparsity level:  tensor(0.8077, device='cuda:0')
Sparsity level:  tensor(0.8077, device='cuda:0')
Sparsity level:  tensor(0.7999, device='cuda:0')
Sparsity level:  tensor(0.8099, device='cuda:0')
Sparsity level:  tensor(0.7999, device='cuda:0')
Sparsity level:  tensor(0.8066, device='cuda:0')
Sparsity level:  tensor(0.8101, device='cuda:0')
Sparsity level:  tensor(0.8008, device='cuda:0')
Sparsity level:  tensor(0.8109, device='cuda:0')
Sparsity level:  tensor(0.8013, device='cuda:0')
Sparsity level:  tensor(0.8082, device='cuda:0')
Sparsity level:  tensor(0.8001, device='cuda:0')
Sparsity level:  tensor(0.8060, device='cuda:0')
Sparsity level:  tensor(0.8115, device='cuda:0')
Sparsity level:  tensor(0.7998, device='cuda:0')
Sparsity level:  tensor(0.8102, device='cuda:0')
Sparsity level:  tensor(0.8098, device='cuda:0')
Sparsity level:  tensor(0.8107, device='cuda:0')
Sparsity level:  tensor(0.8015, device='cuda:0')
```

[9]:
```
act = save_output.outputs[4][0]
act_alpha  = model.layer1[2].conv1.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)
```

[10]:
```
## This cell is provided

conv_int = torch.nn.Conv2d(in_channels = 16, out_channels=16, kernel_size = 3,␣
↪padding=1, bias = False)
conv_int.weight = torch.nn.parameter.Parameter(weight_int)
#conv_int.bias = model.features[27].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)
bn1 = model.layer1[2].bn1
relu = model.layer1[2].relu
output_recovered = relu(bn1(output_recovered))
```

[11]:
```
print("PSUM recovered error:")
print(abs((save_output.outputs[5][0] - output_recovered)).mean().item())
```

```
PSUM recovered error:
0.00013859561295248568
```

[13]:
```
# act_int.size = torch.Size([128, 16, 32, 32])  <- batch_size, input_ch, ni, nj
a_int = act_int[0,:,0:6,0:6]  # pick only one input out of batch
# a_int.size() = [16, 6, 6]

# conv_int.weight.size() = torch.Size([16, 16, 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([16, 16, 9])

padding = 0
stride = 1
array_size = 8 # row and column number

nig = range(a_int.size(1))  ## ni group [0,1,...5]
njg = range(a_int.size(2))  ## nj group
nijg = range(a_int.size(1) * a_int.size(2))

icg = range(int(w_int.size(1)))  ## input channel [0,...16]
ocg = range(int(w_int.size(0)))  ## output channel

ic_tileg = range(int(w_int.size(1) / array_size)) ## [0,1]
oc_tileg = range(int(w_int.size(0) / array_size)) ## [0,1]

kijg = range(w_int.size(2)) # [0, .. 8]
ki_dim = int(math.sqrt(w_int.size(2)))  ## Kernel's 1 dim size, 3

######## Padding before Convolution #######
a_pad = torch.zeros(len(icg), len(nig)+padding*2, len(njg)+padding*2).cuda()
# a_pad.size() = [16, 6+0pad, 6+0pad]
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() = [16, (6+0pad)*(6+0pad)]

a_pad_temp = torch.reshape(a_pad, (len(ic_tileg), array_size, -1))
### Input(a_tile) format: [ic_tile, oc_tile, ic index(row#), nij (time step)]␣
↪###
a_tile = torch.tile(a_pad_temp[:, np.newaxis, :, :], (1, len(oc_tileg), 1, 1))
### Weight(w_tile) format: [ic_tile, oc_tile, ic index(row#), io index(col#),␣
↪kij] ###
w_tile = torch.transpose(torch.reshape(torch.stack([w_int[i:i+array_size, j:
↪j+array_size] \
                                    for i in range(0, len(icg), array_size) \
                                    for j in range(0, len(ocg), array_size)]), \
                        (len(ic_tileg), len(oc_tileg), array_size, array_size,␣
↪len(kijg))), 0, 1)
```

[14]:
```
############################################
p_nijg = range(a_tile.size(3)) ## paded activation's nij group [0, ...6*6-1]
### psum(psum) format: [ic_tile, oc_tile, oc index(col#), nij(time step), kij]␣
↪###
psum = torch.zeros(len(ic_tileg), len(oc_tileg), array_size, len(p_nijg),␣
↪len(kijg)).cuda()

for ic_tile in ic_tileg:
    for oc_tile in oc_tileg:
        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_tile[ic_tile, oc_tile, :, :,␣
↪kij])
                psum[ic_tile, oc_tile, :, nij, kij] = m(a_tile[ic_tile,␣
↪oc_tile, :, nij]).cuda()
```

[15]:
```
######## 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,ni,nj]+\
                        psum[ic_tile,oc_tile,:,int(math.
↪sqrt(len(p_nijg)))*(ni+ki) + (nj+kj),len(kig)*ki+kj]
```

[16]:

```
difference = (out - output_int[0, :, 1:5, 1:5])
print(difference.abs().sum())
```

```
tensor(0.0002, device='cuda:0', grad_fn=<SumBackward0>)
```

[36]:

```
### Output(out) format: [oc index(col#), o_nij] ###
out = torch.reshape(out, (len(ocg), -1))
```

[ ]:

```
out.size()
```

[21]:

```
### show this cell partially. The following cells should be printed by students␣
↪###
for p in range(a_tile.size(0)):
    for q in range(a_tile.size(1)):
        X = a_tile[p,q,:,:]  # [array row num, time_steps] only 36 values in an␣
↪image at this layer
        bit_precision = 4
        file = open(f"activation_{p}{q}.txt", 'w') #write to file
        file.write('#time0row7[msb-lsb],time0row6[msb-lst],....
↪,time0row0[msb-lst]#\n')
        file.write('#time1row7[msb-lsb],time1row6[msb-lst],....
↪,time1row0[msb-lst]#\n')
        file.write('#................#\n')
        for i in range(X.size(1)):  # time step
            for j in range(X.size(0)): # row #
                X_bin = '{0:04b}'.format(round(X[7-j,i].item()))
                for k in range(bit_precision):
                    file.write(X_bin[k])
                #file.write(' ')  # for visibility with blank between words,␣
↪you can use
            file.write('\n')
        file.close() #close file
```

[ ]:

```
a_tile[0,0,:,0]
```

[27]:

```
### Complete this cell ###
for p in range(a_tile.size(0)):
    for q in range(a_tile.size(1)):
        bit_precision = 4
        for k in range(w_int.size(2)):
            W = w_tile[p,q,:,:,k]  # w_int[array col num, array row num, kij]
            file = open(f"weight_kij{k}_{p}{q}.txt", 'w') #write to file
            file.write('#col0row7[msb-lsb],col0row6[msb-lst],....
↪,col0row0[msb-lst]#\n')
            file.write('#col1row7[msb-lsb],col1row6[msb-lst],....
↪,col1row0[msb-lst]#\n')
            file.write('#................#\n')
            for i in range(W.size(0)):  # col #
                for j in range(W.size(1)): # row #
                    temp=round(W[i,7-j].item())
                    if(temp < 0 ):
                        temp=temp+16
                    W_bin = '{0:04b}'.format(temp)
                    for k in range(bit_precision):
                        file.write(W_bin[k])
                    #file.write(' ')  # for visibility with blank between␣
↪words, you can use
                file.write('\n')
            file.close() #close file
```

[ ]:

```
w_tile[0,0,0,:,0] # check this number with your 2nd line in weight.txt
```

0.1 PSUM writing

[30]:

```
### Complete this cell ###
for p in range(a_tile.size(0)):
    for q in range(a_tile.size(1)):
        bit_precision = 16
        for k in range(psum.size(2)):
            psum_temp = psum[p,q,:,:,k];
            file = open(f"psum_kij{k}_{p}{q}.txt", 'w') #write to file
            file.write('#time0col7[msb-lsb],time0col6[msb-lst],....
↪,time0col0[msb-lst]#\n')
            file.write('#time1col7[msb-lsb],time1col6[msb-lst],....
↪,time1col0[msb-lst]#\n')
            for i in range(psum_temp.size(1)):  # nijg #
                for j in range(psum_temp.size(0)): # col #
                    temp=round(psum_temp[7-j,i].item())
                    if(temp < 0 ):
                        temp=temp+65536
                    W_bin = '{0:016b}'.format(temp)
```

```
                    for b in range(bit_precision):
                        file.write(W_bin[b])
                    #file.write(' ')  # for visibility with blank between␣
 ↪words, you can use
                file.write('\n')
            file.close() #close file
```

[ ]:
```
psum[0,0,:,0,0]
```

0.2 Output Writing

[40]:
```
bit_precision = 16
file = open('out_all.txt', 'w') #write to file
file.write('#time0col15[msb-lsb],time0col14[msb-lst],....
 ↪,time0col0[msb-lst]#\n')
file.write('#time1col15[msb-lsb],time1col14[msb-lst],....
 ↪,time1col0[msb-lst]#\n')
file.write('#...............#\n')
#print(torch.reshape(output_int[0,:,:,:], (output_int[0,:,:,:].size(0), -1)).
 ↪size())
out_p = torch.relu(out)
for i in range(out_p.size(1)):  # nijg #
    for j in range(out_p.size(0)): # row #
        temp=round(out_p[15-j,i].item())
        if(temp < 0 ):
            temp=temp+65536
        W_bin = '{0:016b}'.format(temp)
        for b in range(bit_precision):
            file.write(W_bin[b])
        #file.write(' ')  # for visibility with blank between words, you can use
    file.write('\n')
file.close() #close file
```

[ ]:
```
out_p[:,0]
```