In [1]:

```
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

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 = "VGG16_quant_project_part1"
model = VGG16_quant()

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)
```

```python
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]
        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
                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
        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 = [150, 225]
    if epoch in adjust_list:
        for param_group in optimizer.param_groups:
            param_group['lr'] = param_group['lr'] * 0.1
```

```
=> Building model...
Files already downloaded and verified
Files already downloaded and verified
```

In [2]:

```
# Let feature[27] be 8x8 (in_ch, out_ch)
model.features[24] = QuantConv2d(256, 8, kernel_size=3, stride=1, padding=1, bias=False)
model.features[25] = torch.nn.BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
model.features[27] = QuantConv2d(8, 8, kernel_size=3, stride=1, padding=1, bias=False)
model.features[30] = QuantConv2d(8, 512, kernel_size=3, stride=1, padding=1, bias=False)
model.features = torch.nn.Sequential(*(list(model.features[:28]) + list(model.features[29:])))
```

In [ ]:

```
lr = 1e-2
weight_decay = 2e-4
epochs = 120
best_prec = 0
```

```python
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)
```

In [3]:
```python
PATH = "result/VGG16_quant_project_part1/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
device = torch.device("cuda")
```

```python
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: 9004/10000 (90%)
```

In [4]:

```python
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()
for layer in model.modules():
    if isinstance(layer, QuantConv2d):
        print("prehooked")
        layer.register_forward_pre_hook(save_output)       ## Input for the module will be grapped
####################################################
dataiter = iter(trainloader)
images, labels = next(dataiter)
```

```
images = images.to(device)
out = model(images)
```

```
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
```

In [5]:
```
w_bit = 4
weight_q = model.features[27].weight_q # quantized value is stored during the training
w_alpha = model.features[27].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
```

In [6]:
```
act_bit = 4
act = save_output.outputs[8][0]  # input of the quantconv layer
act_alpha  = model.features[27].act_alpha
act_delta = act_alpha/(2**act_bit-1)

act_quant_fn = act_quantization(act_bit) # define the quantization function
act_q = act_quant_fn(act, act_alpha)         # create the quantized value for x

act_int = act_q/act_delta
```

In [7]:
```
conv_int = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3, padding=1, bias = False)
conv_int.weight = torch.nn.parameter.Parameter(weight_int)

output_int = F.relu(conv_int(act_int))     # output_int can be calculated with conv_int and x_int
output_recovered = output_int*w_delta*act_delta

output_ref = save_output.outputs[9][0] # get input of next quantconv
```

```
difference = abs(output_ref - output_recovered)
print(difference.mean())
```

```
tensor(3.2691e-07, device='cuda:0', grad_fn=<MeanBackward0>)
```

In [8]:
```
# Write to activation.txt, psum.txt, weight.txt, output.txt
a_int = act_int[0,:,:,:]  # pick only one input out of batch
o_int = output_int[0,:,:,:]
# a_int.size() = [8, 32, 32]

# conv_int.weight.size() = torch.Size([8, 8, 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([8, 8, 9])

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

nig = range(a_int.size(1))  ## ni group
njg = range(a_int.size(2))  ## nj group

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

ic_tileg = range(int(len(icg)/array_size))
oc_tileg = range(int(len(ocg)/array_size))

kijg = range(w_int.size(2))
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(nig)+padding*2).cuda()
# a_pad.size() = [8, 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))
# a_pad.size() = [8, (32+2pad)*(32+2pad)]
o_int = torch.reshape(o_int, (o_int.size(0), -1))

a_tile = torch.zeros(len(ic_tileg), array_size,    a_pad.size(1)).cuda()
w_tile = torch.zeros(len(oc_tileg)*len(ic_tileg), array_size, array_size, len(kijg)).cuda()
```

```
for ic_tile in ic_tileg:
    a_tile[ic_tile,:,:] = a_pad[ic_tile*array_size:(ic_tile+1)*array_size,:]

for ic_tile in ic_tileg:
    for oc_tile in oc_tileg:
        w_tile[oc_tile*len(oc_tileg) + ic_tile,:,:,:] = w_int[oc_tile*array_size:(oc_tile+1)*array_size, ic_tile*array_size:(i



#############################################

p_nijg = range(a_pad.size(1)) ## psum nij group

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

for kij in kijg:
    for ic_tile in ic_tileg:       # Tiling into array_sizeXarray_size array
        for oc_tile in oc_tileg:   # Tiling into array_sizeXarray_size array
            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[oc_tile*array_size:(oc_tile+1)*array_size, ic_tile*array_size:(ic_til
                    m.weight = torch.nn.Parameter(w_tile[len(oc_tileg)*oc_tile+ic_tile,:,:,kij])
                    psum[ic_tile, oc_tile, :, nij, kij] = m(a_tile[ic_tile,:,nij]).cuda()
```

In [9]:
```
import math

a_pad_ni_dim = int(math.sqrt(a_pad.size(1))) # 32

o_ni_dim = int((a_pad_ni_dim - (ki_dim- 1) - 1)/stride + 1)
o_nijg = range(o_ni_dim**2)

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


### SFP accumulation ###
for o_nij in o_nijg:
    for kij in kijg:
        for ic_tile in ic_tileg:
            for oc_tile in oc_tileg:
                out[oc_tile*array_size:(oc_tile+1)*array_size, o_nij] = out[oc_tile*array_size:(oc_tile+1)*array_size, o_nij]
                psum[ic_tile, oc_tile, :, int(o_nij/o_ni_dim)*a_pad_ni_dim + o_nij%o_ni_dim + int(kij/ki_dim)*a_pad_ni_dim + k
                ## 4th index = (int(o_nij/30)*32 + o_nij%30) + (int(kij/3)*32 + kij%3)

out = F.relu(out)
```

In [10]:
```
out_2D = torch.reshape(out, (out.size(0), o_ni_dim, -1))
difference = (out_2D - output_int[0,:,:,:])
print(difference.sum())
```

```
tensor(-2.2888e-05, device='cuda:0', grad_fn=<SumBackward0>)
```

In [11]:
```
tile_id = 0
nij = 0
X = a_tile[tile_id,:,nij:nij+64]  # [tile_num, array row num, time_steps]

bit_precision = 4
file = open('activation.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
```

In [12]:
```
tile_id = 0
for kij in range(9):
    W = w_tile[tile_id,:,:,kij]  # w_tile[tile_num, array col num, array row num, kij]


    bit_precision = 4
    filename = f"{'weight'}_{kij}{'.txt'}"
    file = open(filename, '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(1)):  # col
        for j in range(W.size(0)):  # row
            val = round(W[7-j,i].item())
            if val < 0:
                val = val + 2**bit_precision
            W_bin = '{0:04b}'.format(val)
            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
```

In [15]:

```
ic_tile_id = 0
oc_tile_id = 0

kij = 0
nij = 0
psum_tile = psum[ic_tile_id,oc_tile_id,:,nij:nij+64,:]
# psum[len(ic_tileg), len(oc_tileg), array_size, len(p_nijg), len(kijg)]

bit_precision = 16
file = open('psum.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')
file.write('#...............#\n')

for y in range(9): # kij
    for i in range(psum_tile.size(1)): # time
        for j in range(psum_tile.size(0)): # col
            val = round(psum_tile[7-j,i,y].item())
            if val < 0:
                val = val + 2**bit_precision
            psum_bin = '{0:016b}'.format(val)
            for k in range(bit_precision):
                file.write(psum_bin[k])
        #file.write(' ')  # for visibility with blank between words, you can use
    file.write('\n')
file.close() #close file
```

In [14]:
```
tile_id = 0
nij = 0
X = o_int[:,nij:nij+64]  # [array row num, time_steps]

bit_precision = 16
file = open('output.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 #
        val = round(X[7-j,i].item())
        if val < 0:
                val = val + 2**bit_precision
        X_bin = '{0:016b}'.format(val)
        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
```

In [ ]:
```

```