VGG16_Quant

December 6, 2024

[3]:
```
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 = "VGG16_quant"
model = VGG16_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]
```

```
        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…
VGG_quant(
  (features): Sequential(
    (0): QuantConv2d(
      3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): QuantConv2d(
      64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (7): QuantConv2d(
      64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): QuantConv2d(
      128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
```

```
      (weight_quant): weight_quantize_fn()
    )
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (14): QuantConv2d(
      128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): QuantConv2d(
      256, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (25): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (26): ReLU(inplace=True)
    (27): QuantConv2d(
      8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (28): ReLU(inplace=True)
    (29): QuantConv2d(
      8, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (30): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
    (31): ReLU(inplace=True)
    (32): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (33): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (34): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (35): ReLU(inplace=True)
    (36): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (37): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (38): ReLU(inplace=True)
    (39): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (40): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (41): ReLU(inplace=True)
    (42): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (43): AvgPool2d(kernel_size=1, stride=1, padding=0)
  )
  (classifier): Linear(in_features=512, out_features=10, bias=True)
)
Files already downloaded and verified
Files already downloaded and verified
```

Training Cell Please Ignore!:

[ ]:
```
lr = 1e-2
weight_decay = 4e-4
epochs = 100
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')
```

```python
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)
```

[4]:

```python
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 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)))
```

```
C:\Users\jacob\AppData\Local\Temp\ipykernel_18828\564645170.py:2: FutureWarning:
You are using `torch.load` with `weights_only=False` (the current default
value), which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  checkpoint = torch.load(PATH)


Test set: Accuracy: 9019/10000 (90%)
```

[5]:

```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()
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)
```

[4]:

```python
weight_q = model.features[27].weight_q
w_alpha = model.features[27].weight_quant.wgt_alpha
w_bit = 4
```

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

[5]:
```
act = save_output.outputs[8][0]
act_alpha  = model.features[27].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)
```

[6]:
```
## This cell is provided

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[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))
output_recovered = torch.relu(output_recovered)
#print(output_recovered)
```

[7]:
```
print("PSUM recovered error:")
print(abs((save_output.outputs[9][0] - output_recovered)).mean().item())
```

```
PSUM recovered error:
3.2451487186335726e-07
```

[8]:
```
# 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
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() = [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))
# a_pad.size() = [64, (32+2pad)*(32+2pad)]

w_int = torch.reshape(weight_int, (weight_int.size(0), weight_int.size(1), -1))␣
 ↪# merge ki, kj index to kij

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

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

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[oc_tile*array_size:
 ↪(oc_tile+1)*array_size, ic_tile*array_size:(ic_tile+1)*array_size, kij])
        m.weight = torch.nn.Parameter(w_int[:,:,kij])
        psum[:, nij, kij] = m(a_pad[:,nij]).cuda()
```

[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:
        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]
                ## 4th index = (int(o_nij/30)*32 + o_nij%30) + (int(kij/3)*32 +␣
↪kij%3)
```

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

```
tensor(-1.0729e-05, device='cuda:0', grad_fn=<SumBackward0>)
```

[11]:
```
### show this cell partially. The following cells should be printed by students␣
↪###

X = a_pad[:,:]  # [array row num, time_steps] only 36 values in an image at␣
↪this layer
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
```

[ ]:
```
kij = range(w_int.size(2))

bit_precision = 4
for k in kij:
    W = w_int[:,:,k]  # w_int[array col num, array row num, kij]
    file = open('weight_kij{}.txt'.format(k), '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
```

[13]:
```
W[0,:] # check this number with your 2nd line in weight.txt
```

[13]:
```
tensor([-1.,  5.,  1.,  2.,  2., -0.,  0., -4.], device='cuda:0',
       grad_fn=<SliceBackward0>)
```

0.1 PSUM writing

[23]:
```
### Complete this cell ###

bit_precision = 16

for k in range(psum.size(2)):
    file = open('psum_kij{}.txt'.format(k), '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 i in range(psum.size(1)):  # nijg #
        for j in range(psum.size(0)): # col #
            temp=round(psum[7-j,i,k].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
```

0.2 Output Writing

[22]:
```
bit_precision = 16
file = open('out.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')
print(torch.reshape(output_int[0,:,:,:], (output_int[0,:,:,:].size(0), -1)).
 ↪size())
out = torch.relu(torch.reshape(output_int[0,:,:,:], (output_int[0,:,:,:].
 ↪size(0), -1)))
for i in range(out.size(1)):  # nijg #
    for j in range(out.size(0)): # row #
        temp=round(out[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
```

```
torch.Size([8, 16])
```

1 Output Stationary Files:

Initial setup to save our files

[6]:
```
weight_q = model.features[0].weight_q
w_alpha = model.features[0].weight_quant.wgt_alpha
w_bit = 4

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

[7]:
```
act = save_output.outputs[0][0]
act_alpha  = model.features[0].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)
```

[8]:
```
conv_int = torch.nn.Conv2d(in_channels = 3, out_channels=64, kernel_size = 3,␣
 ↪padding=1)
conv_int.weight = torch.nn.parameter.Parameter(weight_int)
conv_int.bias = model.features[0].bias
output_int = conv_int(act_int)
output_recovered = output_int * (act_alpha / (2**act_bit-1)) * (w_alpha /␣
 ↪(2**(w_bit-1)-1))
```

```
output_recovered = torch.relu(output_recovered)
#print(output_recovered)

#print("PSUM recovered error:")
#print(abs((save_output.outputs[1][0] - output_recovered)).mean().item())
```

[9]:
```
# 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
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() = [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))
# a_pad.size() = [64, (32+2pad)*(32+2pad)]
#w_int = torch.reshape(weight_int, (weight_int.size(0), weight_int.size(1),␣
↪-1)) # merge ki, kj index to kij

#a_tile = torch.zeros(len(ic_tileg), array_size,    a_pad.size(1)).cuda()
w_tile = torch.zeros(len(oc_tileg), array_size, int(w_int.size(1)), len(kijg)).
↪cuda()
```

```
for oc_tile in oc_tileg:
    w_tile[oc_tile,:,:,:] = w_int[oc_tile*array_size:(oc_tile+1)*array_size, :,␣
 ↪:]


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

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

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

for kij in kijg:
        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, int(w_int.size(1)), bias=False)
                    #m.weight = torch.nn.Parameter(w_int[oc_tile*array_size:
 ↪(oc_tile+1)*array_size, ic_tile*array_size:(ic_tile+1)*array_size, kij])
                    m.weight = torch.nn.Parameter(w_tile[oc_tile,:,:,kij])
                    psum[ oc_tile, :, nij, kij] = m(a_pad[:,nij]).cuda()
```

[10]:
```
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 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[ 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 + kij%ki_dim, kij]
                ## 4th 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))
difference = (out_2D - output_int[0,:,:,:])
#print(difference.sum())
```

1.1 Activation File Creation

[11]:
```
#print(a_pad_2D.size())
#print(a_int.size(1))
#X = a_pad[:,0:8]  # [array row num, time_steps] only 36 values in an image at␣
 ↪this layer
bit_precision = 4
for ic in range(a_pad.size(0)):
    file = open('activation_os_ic{}.txt'.format(ic), '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 j in range(ki_dim):  # second kernal dim
        for i in range(ki_dim): # row in input map?
            for o_nij in range(8): #only eight outputs
                temp = round(a_pad[ic,(7-(o_nij)+i)+((a_int.size(1)+2)*j)].
 ↪item())
                if(temp < 0 ):
                    temp=temp+16
                X_bin = '{0:04b}'.format(temp)
                #print((7-(o_nij)+i)+(a_int.size(1)*j))
                #print("i=",i,"j=",j)
                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
#print(a_pad[2,0:32*3])
```

[ ]:
```
print(a_int[0])
```

1.1.1 Weight File Creation

[13]:
```
kij = range(w_int.size(2))

bit_precision = 4
for ic in range(a_pad.size(0)):
    W = w_int[0:8,ic,:]  # w_int[array col num, array row num, kij]
    file = open('weight_os_ic{}.txt'.format(ic), '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)): # kij #
        for j in range(W.size(0)): # col #
            temp=round(W[7-j,i].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
```

[ ]:
```
print(w_int[0:8,0,:])
```

1.1.2 Output File Creation

[16]:
```
bit_precision = 16
file = open('out_os.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')

out = out
for i in range(8):  # nijg #
    for j in range(8): # row #
        temp=round(out[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
```

[ ]:
```
print(out[0:8,0:8])
```