In [1]:
```
import argparse
import os
import time
import shutil
import sys

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

# Add parent directory to path to import models if needed, though we will define custom classes here
sys.path.append(os.path.abspath(os.path.join(os.getcwd(), '..')))

from models.quant_layer import weight_quantize_fn, act_quantization

global best_prec
use_gpu = torch.cuda.is_available()
print(f'Using GPU: {use_gpu}')
```

```
Using GPU: True
```

Model Definition

Here we define the custom VGG model with:

1. **2-bit Activation** and **4-bit Weight** quantization.
2. A specific layer (around 27th) squeezed to **16 channels** with **Batch Normalization removed**.

In [2]:
```
class QuantConv2d_Custom(nn.Conv2d):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=False, a_bit=2,
        super(QuantConv2d_Custom, self).__init__(in_channels, out_channels, kernel_size, stride, padding, dilation, groups, bi
        self.layer_type = 'QuantConv2d_Custom'
        self.w_bit = w_bit
```

```
        self.a_bit = a_bit
        self.weight_quant = weight_quantize_fn(w_bit=w_bit)
        self.act_alq = act_quantization(a_bit)
        self.act_alpha = torch.nn.Parameter(torch.tensor(8.0))
        self.weight_q  = torch.nn.Parameter(torch.zeros([out_channels, in_channels, kernel_size, kernel_size]))
        self.is_quantized = False # Flag to track if quantization has happened

    def forward(self, x):
        weight_q = self.weight_quant(self.weight)
        self.weight_q = torch.nn.Parameter(weight_q) # Store quantized weight for verification
        self.is_quantized = True
        x = self.act_alq(x, self.act_alpha)
        return F.conv2d(x, weight_q, self.bias, self.stride, self.padding, self.dilation, self.groups)

    def show_params(self):
        wgt_alpha = round(self.weight_quant.wgt_alpha.data.item(), 3)
        act_alpha = round(self.act_alpha.data.item(), 3)
        print('clipping threshold weight alpha: {:2f}, activation alpha: {:2f}'.format(wgt_alpha, act_alpha))
```

In [3]:

```
# Modified configuration for VGG16
# Original: [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']
# We modify the block after the 3rd 'M' (MaxPool). Input to this block is 256.
# Original Block: 512, 512, 512
# Modified Block: 16, '16_no_bn', 512
# 1. Conv(256 -> 16) -> BN -> ReLU
# 2. Conv(16 -> 16) -> ReLU (No BN) [Target Layer]
# 3. Conv(16 -> 512) -> BN -> ReLU

cfg_custom = {
    'VGG16_HW6': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 16, '16_no_bn', 512, 'M', 512, 512, 512, 'M'],
}

class VGG_quant_Custom(nn.Module):
    def __init__(self, vgg_name='VGG16_HW6', a_bit=2, w_bit=4):
        super(VGG_quant_Custom, self).__init__()
        self.a_bit = a_bit
        self.w_bit = w_bit
        self.features = self._make_layers(cfg_custom[vgg_name])
        self.classifier = nn.Linear(512, 10)
```

```
    def forward(self, x):
        out = self.features(x)
        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

    def _make_layers(self, cfg):
        layers = []
        in_channels = 3
        for x in cfg:
            if x == 'M':
                layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            elif x == 'F':
                # 1st layer usually high precision or standard, keeping standard Conv2d
                layers += [nn.Conv2d(in_channels, 64, kernel_size=3, padding=1, bias=False),
                           nn.BatchNorm2d(64),
                           nn.ReLU(inplace=True)]
                in_channels = 64
            elif isinstance(x, str) and '_no_bn' in x:
                # Special case: No BN layer
                out_ch = int(x.replace('_no_bn', ''))
                layers += [QuantConv2d_Custom(in_channels, out_ch, kernel_size=3, padding=1, a_bit=self.a_bit, w_bit=self.w_bi
                           nn.ReLU(inplace=True)]
                in_channels = out_ch
            else:
                # Standard Quantized Layer
                layers += [QuantConv2d_Custom(in_channels, x, kernel_size=3, padding=1, a_bit=self.a_bit, w_bit=self.w_bit),
                           nn.BatchNorm2d(x),
                           nn.ReLU(inplace=True)]
                in_channels = x
        layers += [nn.AvgPool2d(kernel_size=1, stride=1)]
        return nn.Sequential(*layers)

    def show_params(self):
        for m in self.modules():
            if isinstance(m, QuantConv2d_Custom):
                m.show_params()
```

In [4]:
```
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 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

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
```

In [5]:

```
def train(trainloader, model, criterion, optimizer, epoch, print_freq=100):
    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, print_freq=100):
    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:
                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
```

In [6]:

```
# Hyperparameters
batch_size = 128
lr = 0.1
weight_decay = 1e-4
epochs = 250
print_freq = 100
model_name = "VGG16_HW6_2bit_act_4bit_wgt"

print('=> Building model...')
model = VGG_quant_Custom(vgg_name='VGG16_HW6', a_bit=2, w_bit=4)

if use_gpu:
    model.cuda()
    criterion = nn.CrossEntropyLoss().cuda()
else:
    criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9, weight_decay=weight_decay)

# scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)

print('=> Loading Data...')
normalize = transforms.Normalize(mean=[0.491, 0.482, 0.447], std=[0.247, 0.243, 0.262])

# Ensure data path is correct. If running in software/hw/, data might be in ../data or ./data
# Adjust root as necessary
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)

# Results Directory
if not os.path.exists('../result'):
    os.makedirs('../result')
fdir = './result/'+str(model_name)
print(fdir)
if not os.path.exists(fdir):
    os.makedirs(fdir)
```

```
=> Building model...
=> Loading Data...
Files already downloaded and verified
Files already downloaded and verified
./result/VGG16_HW6_2bit_act_4bit_wgt
```

In [ ]:

```
best_prec = 0
for epoch in range(0, epochs):
    adjust_learning_rate(optimizer, epoch)
    print(f"Epoch {epoch} started...")
    train(trainloader, model, criterion, optimizer, epoch, print_freq)
    # scheduler.step()

    print("Validation starts")
    prec = validate(testloader, model, criterion, print_freq)

    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 [7]:
```
# Test set accuracy
PATH = "./result/VGG16_HW6_2bit_act_4bit_wgt/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: 8903/10000 (89%)
```

PSUM Recovery and Verification

Here we verify that the `psum_recovered` matches the pre-hooked input for the next layer.

1. We hook the input of the **Target Layer** (16->16).
2. We hook the input of the **Next Layer** (16->512). This serves as our reference output (post-ReLU).

3. We verify that `Conv(quantized_input) -> ReLU` matches the reference.

In [37]:
```python
# Load Best Model (If available)
model_path = os.path.join(fdir, 'model_best.pth.tar')
if os.path.exists(model_path):
    print(f"=> Loading checkpoint {model_path}")
    checkpoint = torch.load(model_path)
    model.load_state_dict(checkpoint['state_dict'])
else:
    print("No checkpoint found. Using random weights for demonstration.")

model.eval()

# Indices based on cfg_custom['VGG16_HW6']
# [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 16, '16_no_bn', 512, ...]
# Blocks before:
# 64 block: 2 layers (idx 0-6 with M)
# 128 block: 2 layers (idx 7-13 with M)
# 256 block: 3 layers (idx 14-23 with M)
# 512 block start:
#   24: Conv(256->16)
#   25: BN
#   26: ReLU
#   27: Conv(16->16) [Target, No BN]
#   28: ReLU
#   29: Conv(16->512)

target_layer_idx = 27
next_layer_idx = 29

print(f"Target Layer: {model.features[target_layer_idx]}")
print(f"Next Layer: {model.features[next_layer_idx]}")

# Hook Setup
class SaveInput:
    def __init__(self):
        self.inputs = []
    def __call__(self, module, module_in):
        self.inputs.append(module_in)
    def clear(self):
```

```
        self.inputs = []

save_input_target = SaveInput()
save_input_next = SaveInput()

hook_target = model.features[target_layer_idx].register_forward_pre_hook(save_input_target)
hook_next = model.features[next_layer_idx].register_forward_pre_hook(save_input_next)

# Run Inference
dataiter = iter(testloader)
images, labels = next(dataiter)
images = images.cuda() if use_gpu else images

with torch.no_grad():
    model(images)

# Get Captured Inputs
x_input = save_input_target.inputs[0][0] # Input to 16->16 layer
x_ref_next = save_input_next.inputs[0][0] # Input to 16->512 layer (Reference Output of 16->16)

print(f"Input shape: {x_input.shape}")
print(f"Reference Output shape: {x_ref_next.shape}")

# Check if quantization was actually performed
# If the model is new and un-trained/un-forwarded, weight_q might be all zeros if initialized that way in __init__
# But in our QuantConv2d_Custom, weight_q is a Parameter initialized to zeros, BUT it is updated in forward().
# Since we just ran model(images), forward() was called, so weight_q should be populated with quantized values.

target_layer = model.features[target_layer_idx]

if not getattr(target_layer, 'is_quantized', False):
    print("WARNING: Layer reports it has not been quantized! Check forward pass logic.")

# PSUM Recovery Calculation
act_q = target_layer.act_alq(x_input, target_layer.act_alpha)
act_bit = 4
act_alpha = target_layer.act_alpha
act_delta = act_alpha/(2**act_bit-1)
act_int = act_q/act_delta
weight_q = target_layer.weight_q
w_bit = 4
```

```
w_alpha = target_layer.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
output_int_sim = F.conv2d(act_int, weight_int, target_layer.bias, target_layer.stride, target_layer.padding, target_layer.dila
output_recovered = F.relu(output_int_sim)*w_delta*act_delta

diff = (output_recovered - x_ref_next).abs().mean()
print(f"Mean Difference: {diff.item()}")

if diff.item() < 1e-3:
    print("SUCCESS: PSUM Recovery Verified!")
else:
    print("WARNING: Difference is high. Check quantization or layer logic.")

hook_target.remove()
hook_next.remove()
```

```
=> Loading checkpoint ./result/VGG16_HW6_2bit_act_4bit_wgt/model_best.pth.tar
Target Layer: QuantConv2d_Custom(
  16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
)
Next Layer: QuantConv2d_Custom(
  16, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
)
Input shape: torch.Size([128, 16, 4, 4])
Reference Output shape: torch.Size([128, 16, 4, 4])
Mean Difference: 2.2487314765839983e-07
SUCCESS: PSUM Recovery Verified!
```

In [38]:

```
import math
# Write to activation.txt, psum.txt, weight.txt, output.txt
a_int = x_quantized[0,:,:,:]  # pick only one input out of batch
o_int = output_int_sim[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 [39]:
```
a_pad_ni_dim = int(math.sqrt(a_pad.size(1)))

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 [41]:
```
tile_id = 0
nij = 0
X = a_tile[tile_id,:,nij:nij+64]  # [tile_num, array row num, time_steps]

bit_precision = 2
file = open('activation_2b.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 = '{0:02b}'.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 [42]:
```
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}{'_v2.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 [43]:
```
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_v2.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 [44]:

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

bit_precision = 16
file = open('output_v2.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 [ ]: