VGG16_Quantization_Aware_Training

December 11, 2025

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

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
print('=> Building model...')


batch_size = 128
model_name = "VGG16_quant"
model = VGG16_quant_part1()

# 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
        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 VGGNet, the lr starts from 0.01, and is divided by 10 at 50 and 100␣
 ↪epochs"""
    adjust_list = [80, 120, 160]
    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
```

[2]:
```
# Training Cell

# # PATH = "result/VGG_quant_part2_4bit/model_best.pth.tar"
# # checkpoint = torch.load(PATH)
# # model.load_state_dict(checkpoint['state_dict'])

# lr = 1e-2
# weight_decay = 1e-4
# epochs = 200
# best_prec = 0

# #model = nn.DataParallel(model).cuda()
# model.cuda()
# criterion = nn.CrossEntropyLoss().cuda()
# optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,␣
 ↪weight_decay=weight_decay)
# #cudnn.benchmark = True

# 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)
```

[3]:
```
# Load trained model
PATH = "result/VGG16_quant_part1/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model.cuda()
model.eval()

# Evaluate model accuracy
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()

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

```
Test set: Accuracy: 9033/10000 (90%)
```

[4]:
```
# Capture layer inputs using forward pre-hooks
class SaveOutput:
    def __init__(self):
        self.outputs = []
    def __call__(self, module, module_in):
        self.outputs.append(module_in[0])
    def clear(self):
        self.outputs = []

# Register hooks for both layers we need to capture
save_output = SaveOutput()
save_output_next = SaveOutput()
target_layer = model.features[27]  # Current conv layer
target_layer_next = model.features[29]  # Next conv layer (after ReLU)
target_layer.register_forward_pre_hook(save_output)
target_layer_next.register_forward_pre_hook(save_output_next)

# Get a sample batch
dataiter = iter(testloader)
images, labels = next(dataiter)
images = images.cuda()

# Single forward pass to capture both inputs
model.eval()
with torch.no_grad():
    _ = model(images)

x = save_output.outputs[0]  # Input to features[27]
x_next_ref = save_output_next.outputs[0]  # Input to features[29] (after ReLU)
print(f"Captured input shape (features[27]): {x.shape}")
print(f"Captured next layer input shape (features[29]): {x_next_ref.shape}")
```

```
Captured input shape (features[27]): torch.Size([128, 8, 4, 4])
Captured next layer input shape (features[29]): torch.Size([128, 8, 4, 4])
```

[5]:
```
# Convert quantized weights to integers
w_bit = 4
target_conv = model.features[27]
weight_q = target_conv.weight_q.data
w_alpha = target_conv.weight_quant.wgt_alpha.data.item()
w_delta = w_alpha / (2 ** (w_bit-1) - 1)
# divide by alpha first to get normalized values and multiply by (2^(w_bit - 1)
 ↪- 1) to get integers
weight_int = torch.round(weight_q / w_delta).int()
print(f"Weight alpha: {w_alpha:.4f}, Weight delta: {w_delta:.4f}")
print(f"Weight int shape: {weight_int.shape}")
print(f"Weight int sample (first few values): {weight_int.flatten()[:10]}")
```

```
Weight alpha: 2.1499, Weight delta: 0.3071
Weight int shape: torch.Size([8, 8, 3, 3])
Weight int sample (first few values): tensor([ 1,  0, -2,  4,  3, -2, -3, -2,
-5, -4], device='cuda:0',
       dtype=torch.int32)
```

[6]:
```python
# Convert quantized activations to integers
x_bit = 4
target_conv = model.features[27]
x_alpha = target_conv.act_alpha.data.item()
x_delta = x_alpha / (2 ** x_bit - 1)

act_quant_fn = act_quantization(x_bit)
x_q = act_quant_fn(x, x_alpha)
x_int = torch.round(x_q / x_delta).int()
print(f"Activation alpha: {x_alpha:.4f}, Activation delta: {x_delta:.4f}")
print(f"x_int shape: {x_int.shape}")
print(f"x_int sample (first few values): {x_int.flatten()[:10]}")
```

```
Activation alpha: 5.8155, Activation delta: 0.3877
x_int shape: torch.Size([128, 8, 4, 4])
x_int sample (first few values): tensor([0, 0, 0, 0, 3, 1, 0, 0, 2, 1],
device='cuda:0', dtype=torch.int32)
```

[7]:
```python
# Perform integer convolution and recover floating-point output
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.float())

output_int = conv_int(x_int.float())
output_recovered = output_int * x_delta * w_delta
print(f"Output recovered shape: {output_recovered.shape}")
print(f"Output recovered sample: {output_recovered[0, 0, :5, :5]}")
```

```
Output recovered shape: torch.Size([128, 8, 4, 4])
Output recovered sample: tensor([[-0.9526, -2.1434, -0.9526, -0.4763],
        [ 0.8335, -0.4763, -6.4301, -7.9781],
        [ 3.2150,  6.7873,  3.8104, -1.1908],
        [ 7.2636, 10.5977, 12.5030,  7.5018]], device='cuda:0',
       grad_fn=<SliceBackward0>)
```

[8]:
```python
# Apply ReLU to the recovered output and compare with reference
output_recovered_relu = F.relu(output_recovered)
```

```
difference = abs(x_next_ref - output_recovered_relu)
print(f"Difference mean: {difference.mean():.10f}")
print(f"Difference max: {difference.max():.10f}")
print(f"Difference should be < 10^-3: {difference.mean() < 1e-3}")
```

```
Difference mean: 0.0000003776
Difference max: 0.0000078678
Difference should be < 10^-3: True
```

[9]:
```
# Prepare data for systolic array computation (no tiling - 8x8 array handles 8 input and 8 output channels)
# Extract single sample from batch
a_int = x_int[0,:,:,:]  # Shape: [8, 4, 4] - input channels, height, width

# Reshape weights: [out_ch, in_ch, ki, kj] -> [out_ch, in_ch, kij]
w_int = torch.reshape(weight_int, (weight_int.size(0), weight_int.size(1), -1)) # [8, 8, 9]

padding = 1
stride = 1
array_size = 8  # 8x8 systolic array (matches 8 input and 8 output channels)

# Define ranges
nig = range(a_int.size(1))  # ni (height)
njg = range(a_int.size(2))   # nj (width)
icg = range(int(w_int.size(1)))  # input channels (8)
ocg = range(int(w_int.size(0)))  # output channels (8)
kijg = range(w_int.size(2))  # kernel elements (3x3 = 9)
ki_dim = int(math.sqrt(w_int.size(2)))  # kernel dimension (3)

# Pad activations
a_pad = torch.zeros(len(icg), len(nig)+padding*2, len(njg)+padding*2).cuda()
a_pad[:, padding:padding+len(nig), padding:padding+len(njg)] = a_int.cuda()
a_pad = torch.reshape(a_pad, (a_pad.size(0), -1))  # [8, 36] - flattened spatial dimensions (8 channels, 36 nij positions)

# No tiling needed - 8x8 array handles all 8 input and 8 output channels
# Keep integer versions for file writing, and float versions for computation
# Ensure integer type for file writing (a_pad might be float from torch.zeros)
a_full_int = a_pad.int()  # [8, 36] - all input channels, all nij positions (int for file writing)
a_full = a_pad.float()  # [8, 36] - float version for computation
w_full_int = w_int  # [8, 8, 9] - all output channels, all input channels, all kernel elements (int for file writing)
w_full = w_int.float()  # [8, 8, 9] - float version for computation

# Compute partial sums using nn.Linear (no tiling - direct 8x8 computation)
```

```
# psum shape: [8 output channels, 36 nij positions, 9 kernel elements]
p_nijg = range(a_pad.size(1))  # 36 nij positions
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)
        # Use full 8x8 weight matrix
        m.weight = torch.nn.Parameter(w_full[:, :, kij])
        psum[:, nij, kij] = m(a_full[:, nij]).cuda()
```

[10]:
```
# Accumulate partial sums across kernel elements (no tiling needed)
a_pad_ni_dim = int(math.sqrt(a_pad.size(1)))  # 6 (4+2*padding)
o_ni_dim = int((a_pad_ni_dim - (ki_dim- 1) - 1)/stride + 1)  # 4
o_nijg = range(o_ni_dim**2)  # 16 output spatial positions

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

# Accumulate across all kernel elements
# psum shape: [8 output channels, 36 nij positions, 9 kernel elements]
for kij in kijg:
    ky = kij // ki_dim     # kernel row
    kx = kij %  ki_dim     # kernel col

    for o_nij in o_nijg:
        oy = o_nij // o_ni_dim
        ox = o_nij %  o_ni_dim
        nij_src = (oy + ky) * a_pad_ni_dim + (ox + kx)

        # Accumulate: psum[:, nij_src, kij] gives [8] output channels
        out[:, o_nij] += psum[:, nij_src, kij]

# Reshape output to match original shape [8, 4, 4]
out_reshaped = out.view(len(ocg), o_ni_dim, o_ni_dim)

# Recover the output and apply ReLU
out_recovered = out_reshaped * x_delta * w_delta
out_recovered_relu = F.relu(out_recovered)

print(f"Output recovered shape: {out_recovered_relu.shape}")
print(f"Output recovered sample: {out_recovered_relu[0, :5, :5]}")

# Compare with reference
difference_tiled = abs(x_next_ref[0] - out_recovered_relu)
print(f"\nComputation difference mean: {difference_tiled.mean():.6f}")
print(f"Computation difference max: {difference_tiled.max():.6f}")
```

```
print(f"Computation difference should be < 10^-3: {difference_tiled.mean() < 1e-3}")
```

```
Output recovered shape: torch.Size([8, 4, 4])
Output recovered sample: tensor([[ 0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.8335,  0.0000,  0.0000,  0.0000],
        [ 3.2150,  6.7873,  3.8104,  0.0000],
        [ 7.2636, 10.5977, 12.5030,  7.5018]], device='cuda:0',
       grad_fn=<SliceBackward0>)

Computation difference mean: 0.000000
Computation difference max: 0.000002
Computation difference should be < 10^-3: True
```

[11]:
```
# Generate activation data file for hardware testing
# No tiling - 8 channels fit directly in 8x8 array
# Write ALL nij positions (all spatial positions after padding)
# Format: Each row is one nij position, with 8 channels (8 columns × 4 bits = 32 bits per row)

nij_start = 0
nij_end = a_full_int.size(1)  # All nij positions (36 for 4x4 input -> 6x6 after padding)
X = a_full_int[:, nij_start:nij_end]  # [8 channels, 36 nij positions]

bit_precision = 4
file = open('activation_tile0.txt', 'w')
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 (nij positions)
    for j in range(X.size(0)):  # row # (channels)
        val = int(round(X[7-j, i].item()))  # Write row7 first, then row6, ...,row0
        if val < 0:
            val = val + (2**bit_precision)  # Convert negative to unsigned
        X_bin = '{0:04b}'.format(val)
        for k in range(bit_precision):
            file.write(X_bin[k])
    file.write('\n')
file.close()
print(f"Written activation data: shape {X.shape}, nij range [{nij_start}:{nij_end}]")
```

```
Written activation data: shape torch.Size([8, 36]), nij range [0:36]
```

[12]:
```
# Generate weight data files for hardware testing
# No tiling - 8x8 weights fit directly in 8x8 array
# Generate files for all kij values (0-8) as expected by testbench
# w_full_int shape: [8 output channels, 8 input channels, 9 kernel elements]

bit_precision = 4
len_kij = 9  # Number of kernel iterations (0-8)

for kij in range(len_kij):
    # Extract weight matrix for this kij: [8 output channels, 8 input channels]
    W = w_full_int[:, :, kij]  # Shape: [8, 8] (int for file writing)

    # Generate filename matching testbench expectation: weight_itile0_otile0_kij{kij}.txt
    filename = f'weight_itile0_otile0_kij{kij}.txt'
    file = open(filename, 'w')

    # Write 3 comment lines (testbench skips first 3 lines)
    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')

    # Write 8 lines (one per output channel/column), each with 32 bits (8 input channels × 4 bits)
    for i in range(W.size(0)):  # columns (8 output channels)
        for j in range(W.size(1)):  # rows (8 input channels)
            weight_val = int(round(W[7-j, i].item()))  # Write row7 first, then row6, ..., row0
            if weight_val < 0:
                weight_val = weight_val + (2**bit_precision)  # Convert to unsigned
            W_bin = '{0:04b}'.format(weight_val)
            for k in range(bit_precision):
                file.write(W_bin[k])
        file.write('\n')
    file.close()
    print(f"Written weight data: shape {W.shape}, kij={kij}, file={filename}")

print(f"\nGenerated {len_kij} weight files for hardware testbench")
```

```
Written weight data: shape torch.Size([8, 8]), kij=0,
file=weight_itile0_otile0_kij0.txt
Written weight data: shape torch.Size([8, 8]), kij=1,
file=weight_itile0_otile0_kij1.txt
Written weight data: shape torch.Size([8, 8]), kij=2,
file=weight_itile0_otile0_kij2.txt
Written weight data: shape torch.Size([8, 8]), kij=3,
```

```
file=weight_itile0_otile0_kij3.txt
Written weight data: shape torch.Size([8, 8]), kij=4,
file=weight_itile0_otile0_kij4.txt
Written weight data: shape torch.Size([8, 8]), kij=5,
file=weight_itile0_otile0_kij5.txt
Written weight data: shape torch.Size([8, 8]), kij=6,
file=weight_itile0_otile0_kij6.txt
Written weight data: shape torch.Size([8, 8]), kij=7,
file=weight_itile0_otile0_kij7.txt
Written weight data: shape torch.Size([8, 8]), kij=8,
file=weight_itile0_otile0_kij8.txt

Generated 9 weight files for hardware testbench
```

[13]:
```python
# Generate partial sum data file for hardware testing
# No tiling - 8 output channels fit directly in 8x8 array
# Write all nij positions for a specific kij (typically kij=0 for initial psum)

kij = 0
nij_start = 0
nij_end = psum.size(1)  # All nij positions (36)

# Extract psum: psum[:, nij_start:nij_end, kij]
# psum shape: [8 output channels, 36 nij positions, 9 kernel elements]
psum_tile = psum[:, nij_start:nij_end, kij]  # Shape: [8, 36]

bit_precision = 16
file = open('psum.txt', 'w')
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')

# Write all nij positions (all time steps)
for i in range(psum_tile.size(1)):  # time (nij positions)
    for j in range(psum_tile.size(0)):  # column (PE/output channel)
        psum_val = int(round(psum_tile[7-j, i].item()))  # Write col7 first,␣
 ↪then col6, ..., col0
        if psum_val < 0:
            psum_val = psum_val + (2**bit_precision)  # Convert to unsigned
        psum_bin = '{0:16b}'.format(psum_val)
        for k in range(bit_precision):
            file.write(psum_bin[k])
    file.write('\n')
file.close()
print(f"Written psum data: shape {psum_tile.shape}, nij range [{nij_start}:
 ↪{nij_end}], kij={kij}")
```

```
Written psum data: shape torch.Size([8, 36]), nij range [0:36], kij=0
```

[14]:
```
# Generate expected output file for hardware verification (out.txt)
# This file contains the final output after accumulation and ReLU
# Uses out_recovered_relu from Cell 9 computation (no tiling - all 8 channels)

# out_recovered_relu shape: [8, 4, 4] (8 channels, 4x4 spatial)
out_final = out_recovered_relu.cpu()  # Shape: [8, 4, 4]

# Reshape to [8 channels, 16 spatial positions]
o_ni_dim = out_final.size(1)  # 4
out_final_flat = out_final.view(8, -1)  # [8, 16]

bit_precision = 16
file = open('out.txt', 'w')
file.write('#out0col7[msb-lsb],out0col6[msb-lst],....,out0col0[msb-lst]#\n')
file.write('#out1col7[msb-lsb],out1col6[msb-lst],....,out1col0[msb-lst]#\n')
file.write('#................#\n')

# Write 16 lines (one per output spatial position)
# Each line: 128 bits = 8 columns × 16 bits
for i in range(out_final_flat.size(1)):  # 16 output positions
    for j in range(out_final_flat.size(0)):  # 8 columns (output channels)
        out_val = round(out_final_flat[7-j, i].item())
        if out_val < 0:
            out_val = 0  # ReLU already applied, but ensure no negatives
        out_bin = '{0:016b}'.format(out_val & ((2**bit_precision)-1))  # Ensure␣
 ↪16 bits
        file.write(out_bin)
    file.write('\n')
file.close()
print(f"Written output data: shape {out_final_flat.shape}, file=out.txt")
```

```
Written output data: shape torch.Size([8, 16]), file=out.txt
```

[15]:
```
# Generate accumulation address file (acc.txt)
# This file contains memory addresses where partial sums are stored
# Format: 11-bit binary addresses, total of len_onij × len_kij = 16 × 9 = 144␣
 ↪addresses

len_onij = 16  # Number of output spatial positions
len_kij = 9    # Number of kernel iterations

file = open('acc.txt', 'w')

# Generate addresses for each output position and each kij iteration
# Address calculation: kij * len_onij + output_position
for output_pos in range(len_onij):
    for kij in range(len_kij):
```

```
        address = kij * len_onij + output_pos
        # Convert to 11-bit binary
        addr_bin = '{0:011b}'.format(address & ((2**11)-1))  # Ensure 11 bits
        file.write(addr_bin + '\n')

file.close()
print(f"Written accumulation address file: {len_onij * len_kij} addresses,␣
 ↪file=acc.txt")
```

```
Written accumulation address file: 144 addresses, file=acc.txt
```

[ ]: