WUST

学习总结

Mode by : 董 勇

目录
CONTENT

PART

01

S. Han et al., “ESE: Efficient Speech Recognition Eng-ine with Sparse LSTM on FPGA,” in Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey California USA, Feb. 2017, pp. 75–84, doi: 10.1145/3020078.3021745.

01 Paper

motivation

LSTM模型过大：
存储消耗
运算消耗

contribution

- 剪枝与量化应用于LSTM，剪枝时用到了负载平衡（load balance），将LSTM带来了大量的压缩20x（剪枝10x，量化2x）
- 提出了一个scheduler，用于编码和分组相应的复杂的压缩的LSTM，分配给PE

01 Paper

Pruning & Quatilization

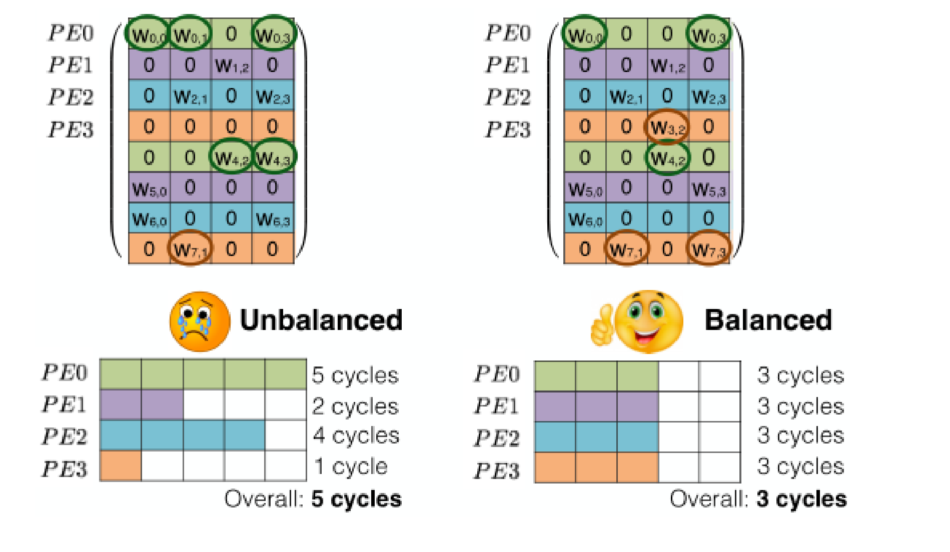

Figure 5: Load Balance Aware Pruning and its Benefit for Parallel Processing

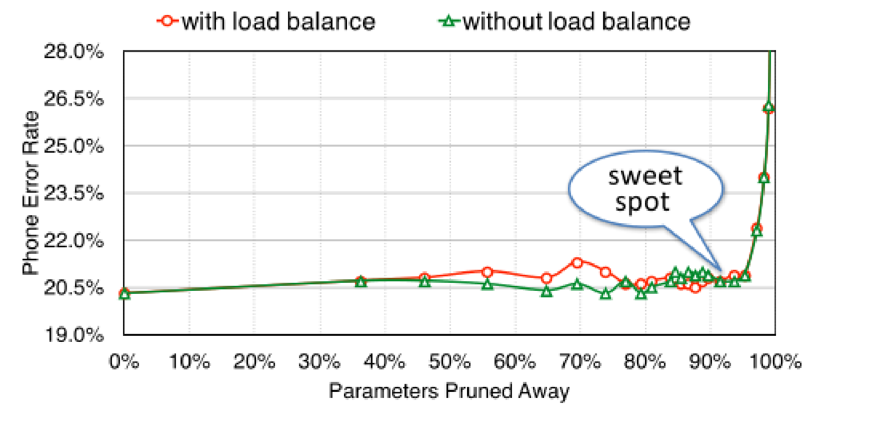

Figure 6: Accuracy curve of load-balance-aware pruning and original pruning.

将分组了的参数按照一致的比例去稀疏，而不是原来那样全局稀疏；
并通过retraining把损失的精度补回来。这样就做到了负载均衡的稀疏参数了。

量化：压缩32bit的浮点数到12bit定点，然后运用线性的量化来实现于weight和activation

01 Paper

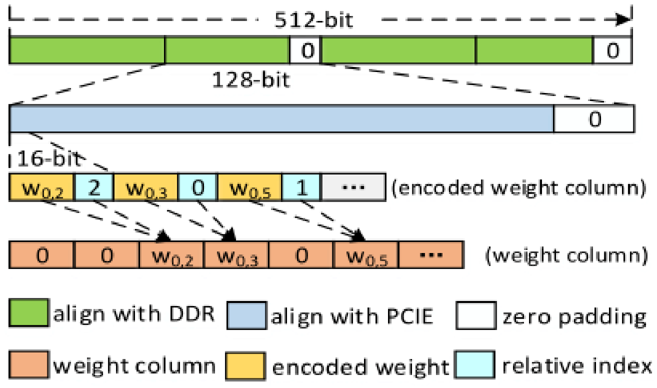

Figure 7: Encoding in CSC format and data align using zero-padding.

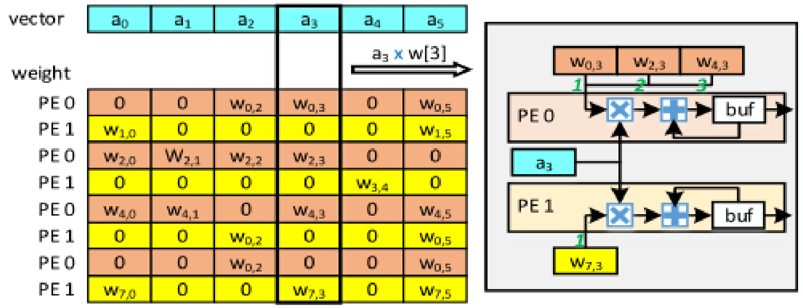

Figure 8: The computation pattern: non-zero weights in a column are assigned to 2 PEs, and every PE multiply-add their weights with the same element from the shared vector.

01 Paper

Architecture Overview

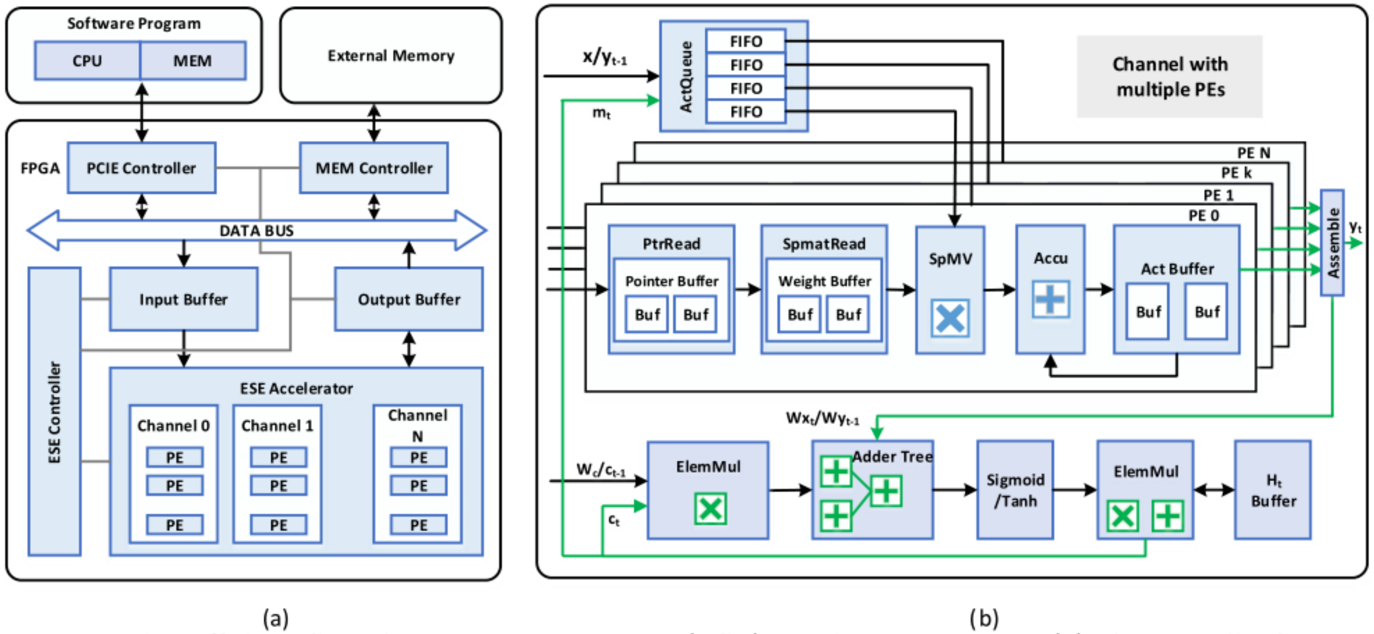

Figure 10: The Efficient Speech Recognition Engine (ESE) accelerator system: (a) the overall ESE system architecture; (b) one channel with multiple processing elements (PEs).

ESE Channel Architecture：

- ActQueue：每个PE之前设置一个队列，用于存储，这样PE之间不同同步，只用处理各自队列上的值。
- SpmatRead：稀疏矩阵读取单元，用于编码权重矩阵的存储和输出。
- SpMV：矩阵相乘单元。
- ElemMul：元素级别的乘法单元。
- Adder Tree：累加结果和偏置bias

01 Paper

Table 7: Power consumption of different platforms.

Platform	CPU Dense	CPU Sparse	GPU Dense	GPU Sparse	ESE
Power	111W	38W	202W	136W	**41W**

Table 8: Performance comparison of running LSTM on ESE, CPU and GPU

Plat.	ESE on FPGA (ours)									CPU		GPU	
Matrix	Matrix Size	Sparsity (%)[1]	Compres. Matrix (Bytes)[2]	Theoreti. Comput. Time (µs)	Real Comput. Time (µs)	Total Operat. (GOP)	Real Perform. (GOP/s)	Equ. Operat. (GOP)	Equ. Perform. (GOP/s)	Real Comput. Time (µs) Dense	Real Comput. Time (µs) Sparse	Real Comput. Time (µs) Dense	Real Comput. Time (µs) Sparse
W_{ix}	1024×153	11.7	36608	2.9	**5.36**	0.0012	218.6	0.010	1870.7	**1518.4[3]**	**670.4**	**34.2**	**58.0**
W_{fx}	1024×153	11.7	36544	2.9	**5.36**	0.0012	218.2	0.010	1870.7				
W_{cx}	1024×153	11.8	37120	2.9	**5.36**	0.0012	221.6	0.010	1870.7				
W_{ox}	1024×153	11.5	35968	2.8	**5.36**	0.0012	214.7	0.010	1870.7				
W_{ir}	1024×512	11.3	118720	9.3	**10.31**	0.0038	368.5	0.034	3254.6	**3225.0[4]**	**2288.0**	**81.3**	**166.0**
W_{fr}	1024×512	11.5	120832	9.4	**10.01**	0.0039	386.3	0.034	3352.1				
W_{cr}	1024×512	11.2	117760	9.2	**9.89**	0.0038	381.2	0.034	3394.5				
W_{or}	1024×512	11.5	120256	9.4	**10.04**	0.0038	383.5	0.034	3343.7				
W_{ym}	512×1024	10.0	104832	8.2	**15.66**	0.0034	214.2	0.034	2142.7	1273.9	611.5	124.8	63.4
Total	**3248128**	**11.2**	**728640**	**57.0**	**82.7**	**0.0233**	**282.2**	**0.208**	**2515.7**	**6017.3**	**3569.9**	**240.3**	**287.4**

CPU sparse比dense快，因为CPU擅长串行处理，而GPU sparse比dense慢，因为GPU注重吞吐量。对于稀疏的LSTM, CPU和GPU都更快，因为内存带宽的节省更加显著。

Conclusion:

ESE在sparse LSTM取得了282 GOPS处理速率，相当于非稀疏的2.52 TOPS的处理速率。

在语音识别的数据集上ESE取得了比 i7 5930和titan X GPU快43x的3x的速度提升

比CPU和GPU能耗降低40x和11.5x

PART 02

- transformer
- NLP(1-2)

02/ Video

1. CNN在文本中的编码能力弱于RNN，而RNN是序列模型，并行能力差，计算缓慢，而且只能考虑一个方向上的信息。Transformer可以综合的考虑两个方向的信息，而且有非常好的并行性质

2.Transformer也包含大量的矩阵运算，cnn中矩阵运算优化的方法也同样适用

PART

03

- RNN
-

id	label
0	0
1	0
2	0
3	1
4	1
5	1
6	0
7	1
8	1
9	1
10	1
11	1
12	1
13	0
14	1
15	1
16	0
17	0
18	1
19	0
20	1
21	1
22	0
23	0
24	1
25	0
26	0
27	1
28	0

03/ Experiment

文本情感分析，训练文件中的句子被标记为“1”或“0”，分别对应句子的情感色彩是“负面”与“正面”

```python
import os
import numpy as np
import pandas as pd
import argparse
from gensim.models import word2vec

def train_word2vec(x):
    # 訓練 word to vector 的 word embedding
    model = word2vec.Word2Vec(x, size=250, window=5, min_count=5, workers=12, iter=10, sg=1
    return model

if __name__ == "__main__":
    print("loading training data ...")
    train_x, y = load_training_data('training_label.txt')
    train_x_no_label = load_training_data('training_nolabel.txt')

    print("loading testing data ...")
    test_x = load_testing_data('testing_data.txt')

    #model = train_word2vec(train_x + train_x_no_label + test_x)
    model = train_word2vec(train_x + test_x)

    print("saving model ...")
    # model.save(os.path.join(path_prefix, 'model/w2v_all.model'))
    model.save(os.path.join(path_prefix, 'w2v_all.model'))
```

```python
from torch import nn
from gensim.models import Word2Vec

class Preprocess():
    def __init__(self, sentences, sen_len, w2v_path="./w2v.model"):
        self.w2v_path = w2v_path
        self.sentences = sentences
        self.sen_len = sen_len
        self.idx2word = []
        self.word2idx = {}
        self.embedding_matrix = []
    def get_w2v_model(self):
        # 把之前訓練好的 word to vec 模型讀進來
        self.embedding = Word2Vec.load(self.w2v_path)
        self.embedding_dim = self.embedding.vector_size
    def add_embedding(self, word):
        # 把 word 加進 embedding，並賦予他一個隨機生成的 representation vector
        # word 只會是 "<PAD>" 或 "<UNK>"
        vector = torch.empty(1, self.embedding_dim)
        torch.nn.init.uniform_(vector)
        self.word2idx[word] = len(self.word2idx)
        self.idx2word.append(word)
        self.embedding_matrix = torch.cat([self.embedding_matrix, vector], 0)
    def make_embedding(self, load=True):
        print("Get embedding ...")
        # 取得訓練好的 Word2vec word embedding
```

```python
import torch
from torch import nn
class LSTM_Net(nn.Module):
    def __init__(self, embedding, embedding_dim, hidden_dim, num_layers, dropout=0.5, fix_embedding=True):
        super(LSTM_Net, self).__init__()
        # 製作 embedding layer
        self.embedding = torch.nn.Embedding(embedding.size(0),embedding.size(1))
        self.embedding.weight = torch.nn.Parameter(embedding)
        # 是否將 embedding fix 住，如果 fix_embedding 為 False，在訓練過程中，embedding 也會跟著被訓練
        self.embedding.weight.requires_grad = False if fix_embedding else True
        self.embedding_dim = embedding.size(1)
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        self.dropout = dropout
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=num_layers, batch_first=True)
        self.classifier = nn.Sequential( nn.Dropout(dropout),
                                         nn.Linear(hidden_dim, 1),
                                         nn.Sigmoid() )

    def forward(self, inputs):
        inputs = self.embedding(inputs)
        x, _ = self.lstm(inputs, None)
        # x 的 dimension (batch, seq_len, hidden_size)
        # 取用 LSTM 最後一層的 hidden state
        x = x[:, -1, :]
        x = self.classifier(x)
        return x
```

要将给定的文本转换成词向量 → 从训练好的word2vec模型中提取出词向量 → 利用pytorch的lstm构建模型 → 训练