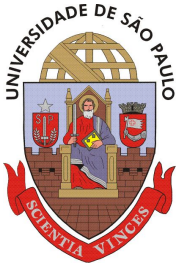


UNIVERSIDADE DE SÃO PAULO

SSC0740

Sistemas Embarcados

YoloV4 TensorFlow Lite

Alunos:

Fernando Akio Tutume de Salles Pucci (8957197)
Lucas Nobuyuki Takahashi (10295670)
Vitor Kodhi Teruya (10284441)

São Carlos, 21 de Dezembro de 2020

1. Introdução

Neste trabalho é feita a conversão de uma rede neural para o formato do *TensorFlow Lite* para ser usado em sistemas embarcados ou *mobiles*, testando as otimizações de quantização.

Para implementação do trabalho foi utilizado a rede neural *yoloV4* já treinada com um *dataset* com 80 classes. A conversão da rede para *tflite* foi feita utilizando a própria biblioteca do *TensorFlow Lite* e o ambiente em que a rede foi executado foi um celular *android*.

1.1. *YoloV4*

Yolo é um modelo de rede neural convolucional para detecção de objetos em tempo real. O modelo utiliza o *framework "You Only Look Once"*, e é baseado na rede *Darknet53*. O Yolo consegue detectar até 9000 objetos diferentes com uma acurácia de 70% em uma velocidade de mais de *60FPS*. A versão utilizada é a quatro, contendo diversas otimizações do algoritmo.

1.2 *TensorFlow Lite*

O *TensorFlow Lite* faz parte da biblioteca *TensorFlow* que é uma biblioteca para treinar e executar redes neurais implementada pelo Google. A ferramenta do *TensorFlow Lite* é composta por duas partes, a que converte a rede neural para o modelo do *TensorFlow Lite (tflite)* e o interpretador que faz a leitura desse arquivo e a execução dele.

O próprio *TensorFlow Lite* disponibiliza implementações do interpretador em *Java*, *Swift*, *Object-C*, *C++* e *Python*, sendo que a implementação do *Java* é para dispositivos *android* e as de *Swift* e *Object-C* para dispositivos *iOS*.

O conversor do *TensorFlow Lite* contém até o momento 3 otimizações de quantização:

- Dynamic Quantization;
- Float16 Quantization;
- Full integer Quantization;

A primeira otimização apenas diminui o tamanho da rede. De acordo com a documentação do *TensorFlow Lite,* ela pode diminuir a rede em até ¼ do seu tamanho, trocando os valores dos pesos para *int*. Porém, a execução da rede ainda

ocorre com o uso de ponto flutuante de 32 *bits*.

Na segunda otimização é feita a conversão de todos os pesos para *float16*, diminuindo o tamanho da rede para metade do tamanho original. Porém, para a execução ser efetiva, é necessário que o *hardware* do dispositivo tenha suporte a operações de *float16*. Caso o hardware não tenha esse suporte, é feita a conversão dos pesos para *float32* na hora da execução.

A última otimização é a conversão da rede como um todo, ou partes dela, para inteiros de 8 *bits* e permitir que a rede seja executada sem a necessidade de converter os pesos para *float*, o que ajuda no desempenho da rede em dispositivos móveis com *hardware* de ponto flutuante limitado. Porém, para fazer essa quantização, é necessário que se tenha uma parte do conjunto de treino.

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

Tabela retirada dos guias do *TensorFlow Lite*, comentando que as otimizações são boas para CPU, GPU e TPU. E também lista os benefícios de cada quantização. (https://www.tensorflow.org/lite/performance/post_training_quantization)

2. Implementação

A implementação foi feita utilizando os pesos da rede *yoloV4* já treinados e para conversão foi utilizado um *script* disponibilizado no github: https://github.com/hunglc007/tensorflow-yolov4-tflite. Ele contém um *script* que implementa operações que o *yolo* utiliza, mas o *TersorFlow* não tem implementado. Ele também disponibiliza uma aplicação *android*, que foi utilizada como base para criar o programa e executar a aplicação.

3. Modelos Gerados

Os modelos gerados para o trabalho foram dois, um com *Dynamic quantization* e outro com *float16 quantization*, não foi utilizado o *Full Integer Quantization,* pois não tínhamos em mão o conjunto de dados usados para treinar a rede.

4. Resultados

O modelo pré-treinado utilizado tinha o arquivo *.weights* com 256MB.

Os modelos foram executados em um celular Motorola Moto G4 Play (2016) e um Samsung Galaxy S9+ (2018).

Especificações do Motorola Moto G4 Play (2016):

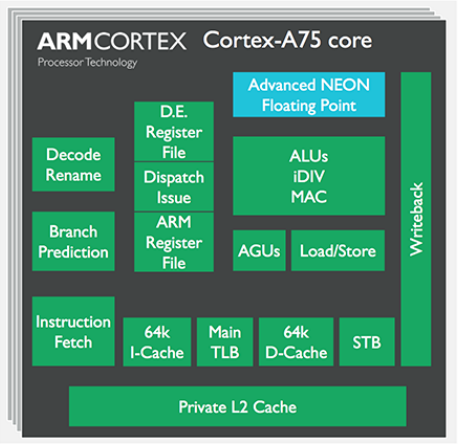
- SoC: Qualcomm MSM8916 Snapdragon 410 (SMIC 28nm);
- CPU: 4x Cortex-A53 @ 1.2GHz;
- GPU: Adreno 306 @ 450MHz;
- RAM: 2GB LPDDR3 Single Channel 32bit @ 533MHz (4.2GB/s);
- Armazenamento: 16GB de eMMC 4.5;
- Sistema Operacional: Android 7.1.1 (API 25).

Especificações do Samsung Galaxy S9+ (2018):

- SoC: Qualcomm SDM845 Snapdragon 845 (Samsung 10nm FinFET LPP);
- CPU: 4x Kyro 385 Gold (Cortex-A75 based) @ 2.8GHz + 4x Kyro 385 Silver (Cortex-A55 based) @ 1.8GHz;
- GPU: Adreno 630 @ 710MHz (727 GFLOPs);
- RAM: 6GB LPDDR4X Quad Channel 16bit (64bit) 1866MHz (29.9GB/s);
- Armazenamento: 128GB de UFS 2.1;
- *Advanced NEON Floating Point* (*FP16* com dobro de vazão em relação ao *FP32*);
- Sistema Operacional: Android 10 (API 29).

Additional capabilities for NEON/FPU

- Dedicated renaming engine for NEON/FPU
- Support for FP16 half-precision processing
  - Double throughput compared to single precision
  - Significant performance uplift for image processing
- Support for Int8 dot product
  - Increased performance on neural network algorithms
- Enhanced floating-point MAC throughput
- Dedicated data store queue



	Tempo		Tamanho do Modelo *tflite*
Dispositivo	Motorola Moto G4 play	Samsung Galaxy S9+	
Dynamic Quantization	15502 ms	1870 ms	62 MB
Float16 Quantization	18503 ms	1990 ms	124 MB

5. Conclusão

Aprendemos que converter a rede para *tflite* não é tão linear assim, já que nem todas as operações estão implementadas nele, necessitando a criação dessas operações para poder executar no interpretador. O resultado da rede *yoloV4* foi bom, mas a demora da execução é um problema. Apesar de não termos conseguido aplicar as otimizações que diminuem a latência, podemos ver que o uso da *Dynamic Quantization* pode diminuir o tamanho da rede de forma considerável.

Também vimos que a rede consegue ser executada muito mais rapidamente quando utilizado um *hardware* melhor, além do suporte nativo para operações de *float16*.