Part 1 GPU Acceleration

1. GPU Accelerated Computing

A graphics processing unit (GPU) is **a specialized micro processor used to process image in personal computers, workstations, game consoles and mobile devices (phone and tablet).** Similar to CPU, but CPU is designed to implement complex mathematical and geometric calculations which are essential for graphics rendering.

GPU-accelerated computing is the employment of a graphics processing unit (GPU) along with a computer processing unit (CPU) in order to accelerate science, analytics, engineering, consumer and cooperation applications. Moreover, GPU can facilitate the applications on various platforms, including vehicles, phones, tablets, drones and robots.

2.Comparison between GPU and CPU

The **main difference between CPU and GPU is how they handle the tasks.** CPU consists of several cores optimized for sequential processing. While GPU owns a large parallel computing architecture composed of thousands of smaller and more effective cores tailored for multitasking simultaneously.

GPU stands out for thousands of cores and large amount of high-speed memory, and is initially intended for processing game and computer image. **It is adept at parallel computing which is ideal for image processing**, because the pixels are relatively independent. And the GPU provides a large number of cores to perform parallel processing on multiple pixels at the same time, but it only improves throughput without alleviating the delay. For example, when receives one message, it will use only one core to tackle this message although it has thousands of cores. GPU cores are usually employed to complete operations related to image processing, which is not universal as CPU.

3. Advantage of GPU

GPU is excellent in massive parallel operations, hence it has an important role in deep learning. Deep learning relies on neural network that is utilized to analyze massive data at high speed.

For example, if you want to let this network recognize the cat, you need to show it lots of the pictures of cats. And that is the forte of GPU. Besides, GPU consumes less resources than CPU.

Part 2 TensorRT Acceleration

1. TensorRT Acceleration Description

TensorRT is a high-performance deep learning inference, includes a deep learning inference optimizer and runtime that delivers low latency and high throughput for inference applications. **It is deployed to hyperscale data centers, embedded platforms, or automotive product platforms to accelerate the inference.**

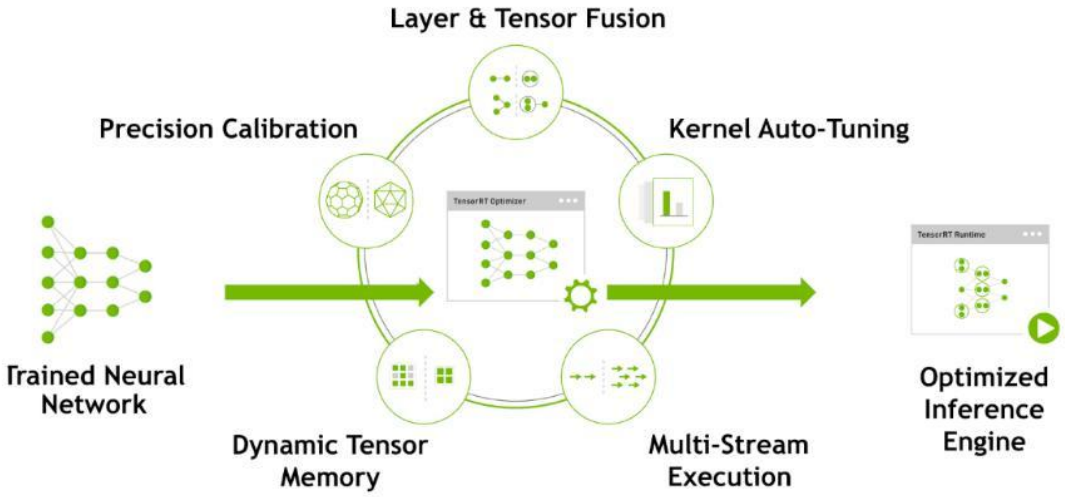
TensoRT supports almost all deep learning frameworks, such as TensorFlow, Caffe, Mxnet and Pytorch. Combing with new NVIDIA GPU, TensorRT can realize swift and effective deployment and inference on almost all frameworks.

To accelerate deployment inference, multiple methods to optimize the models are proposed, such as model compression, pruning, quantization and knowledge distillation. And we can use the above methods to optimize the models during training, however **TensorRT optimize**

the trained models. It improves the model efficiency through optimizing the network computation graph.

After the network is trained, you can directly put the model training file into tensorRT without relying on deep learning framework.

2. Optimization Methods



♦Precision Calibration

Most deep learning frameworks train neural networks with a network tensor of 32-bit floating point precision (FP32). After the network is trained, it is possible to reduce the precision of the data to FP16 or INT8, as backpropagation is not required during the deployment inference process.**Reducing the precision of the data reduces the memory footprint and latency, as well as the size of the model.**

Precision	Dynamic Range
FP32	−3.4×1038 ~+3.4×1038
FP16	−65504 ~+65504
INT8	−128 ~+127

There are only 256 different values in INT8, and if INT8 is used to represent FP32 accuracy values, there is bound to be a loss of information, leading to performance degradation. However, TensorRT is able to provide a fully automated calibration process to reduce FP32 precision data to INT8 precision with optimal matching performance, minimizing the performance loss.

♦Layer & Tensor Fusion

CUDA cores compute tensors very quickly, but still spend a lot of time on CUDA core startup and read/write operations on each level of the input/output tensor, which results in wasted resources on the GPU and bottlenecks in memory bandwidth.

TensorRT optimizes the model structure by reducing the number of layers by merging them horizontally or vertically, which in turn reduces the number of occupied CUDA cores.

Horizontal merging can combine convolution, bias, and activation layers into a single CBR structure that occupies only one CUDA core. A vertical merge can combine layers with the same structure but different weights into a wider layer that also occupies only one CUDA core.

In addition, for the case of multi-branch merging, TensorRT can eliminate the concat layer by directing the layer output to the correct memory address in a non-copy manner, thus reducing the number of memory accesses.

♦Kernel Auto-Tuning

Network model recalls CUDA core of GPU to infer and compute. According to different algorithms, network models and GPU platform, **TensorRT can implement kernel-level optimization to enable the model to compute on the specific platform with best performance.**

♦Dynamic Tensor Memory

When using the tensor, TensorRT will designate its memory to avoid repetitive application, reduce storage occupation and improve the reuse efficiency.

♦Multi-Stream Execution

TensorRT employs stream technology of CUDA to perform parallel operation on multiple branches with the same input, and can optimize based on different batchsize.

Part 3 MediaPipe Development

1. MediaPipe Description

MediaPipe is an open-source framework of multi-media machine learning models. Cross-platform MediaPipe can run on mobile devices, workspace and servers, as well as support mobile GPU acceleration. It is also compatible with **TensorFlow and TF Lite Inference Engine,** and all kinds of TensorFlow and TF Lite models can be applied on it. Besides, MediaPipe supports GPU acceleration of mobile and embedded platform.

2. Pros and Cons

2.1 Pros

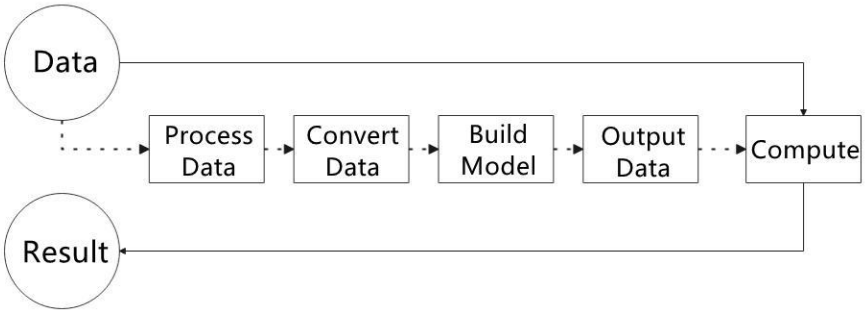
1) MediaPipe supports various platforms and languages, including iOS, Android, C++, Python, JAVAScript, Coral, etc.
2) Swift running. Models can run in real-time.
3) Models and codes are with high reuse rate.

2.2 Cons

1) For mobile devices, MediaPipe will occupy 10M or above.
2) As it greatly depends on Tensorflow, you need to alter large amount of codes if you want to change it to other machine learning frameworks, which is not friendly to machine learning developer.
3) It adopts static image which can improve efficiency, but make it difficult to find out the errors.

3. How to use MediaPipe

The figure below shows how to use MediaPipe. The solid line represents the part to coded, and the dotted line indicates the part not to coded. MediaPipe can offer the result and the function realization framework quickly .



3.1 Dependencies

MediaPipe relies on OpenCV for video and FFMPEG for audio data. It also has other dependencies such as OpenGL/Metal, Tensorflow, Eigen, and so on.

4. Use MediaPipe

MediaPipe can be employed to realize **human body tracking, 3D face detection, face detection, gesture recognition**, etc.

4.1 Face Mesh

1） Start JetAuto, then connect to ubuntu desktop through NoMachine.

2） Press "**Ctrl+Alt+T**" to open command line terminal.

3） Next, input command "**systemctl stop start_app_node.service**" to stop auto-start program.

4） Input command "**python3 ./jetauto_ws/src/jetauto_example/scripts/mediapipe_example/face_mesh.py**" to start 3D face detection game.

5） Input command "**sudo vim jetauto_ws/src/jetauto_example/scripts/mediapipe_example/face_mesh.py**" to open the corresponding **.py** file.

```
#!/usr/bin/env python3
# encoding: utf-8
# Real-time face mesh detection is implemented using the MediaPipe library,
which can be performed via webcam or video input.
import cv2
import mediapipe as mp
import jetauto_sdk.fps as fps


# Initializing the MediaPipe Component
mp_drawing = mp.solutions.drawing_utils # Initialize MediaPipe's Drawing Tools
module
mp_face_mesh = mp.solutions.face_mesh # Initializes MediaPipe's face mesh
detection module.

# Set up webcam input:
```

```
drawing_spec = mp_drawing.DrawingSpec(thickness=1, circle_radius=1) # Configure
face keypoint drawing parameters
cap = cv2.VideoCapture("/dev/astrapro") # Initialize a video capture object
#cap = cv2.VideoCapture("/dev/usb_cam")
print('\n******Press any key to exit!******')


fps = fps.FPS()

# Used to initialize the face mesh detector.
with mp_face_mesh.FaceMesh(
    max_num_faces=1,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5) as face_mesh:
  while cap.isOpened():
    success, image = cap.read() # Reads frames from the webcam.
    if not success:
      print("Ignoring empty camera frame.")
      # If loading a video, use 'break' instead of 'continue'.
      continue

    # To improve performance, optionally mark the image as not writeable to
pass by reference.
    image.flags.writeable = False
    # Converts the image from BGR format to RGB format because MediaPipe needs
the input image in RGB format.
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Perform face mesh detection on the image and return the detection result.
    results = face_mesh.process(image)
    # Converts an image from RGB format back to BGR format for processing by
subsequent OpenCV functions.
    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)


    # Set the image to writable to draw detected face keypoints on the image.
    image.flags.writeable = True

    # Check if the face keypoint is detected.
    if results.multi_face_landmarks:
      # Traverse each detected face keypoint.
      for face_landmarks in results.multi_face_landmarks:
         # Plot the detected face keypoints on the image.
         mp_drawing.draw_landmarks(
            image=image,
            landmark_list=face_landmarks,
            landmark_drawing_spec=drawing_spec)
    # Update frame rate information.
    fps.update()
```

```
    # Display the frame rate information on the image and also flip the image
horizontally using the cv2.flip() function to give it a selfie view.
    result_image = fps.show_fps(cv2.flip(image, 1))
    # Flip the image horizontally for a selfie-view display.

    # Displays the processed image with a window titled "MediaPipe Face Mesh".
    cv2.imshow('MediaPipe Face Mesh', result_image)
    key = cv2.waitKey(1)
    if key != -1:
        break

cap.release()
cv2.destroyAllWindows()
```

4.2 Self Segmentation

4.3 3D Object Detection

4.4 Face Detection

4.5 Hand Keypoint Detection

4.6 Pose Estimation

4.7 Whole body holistic testing

Part 4 Yolov5 Model Training