# 利用TENSORRT自由搭建高性能推理模型

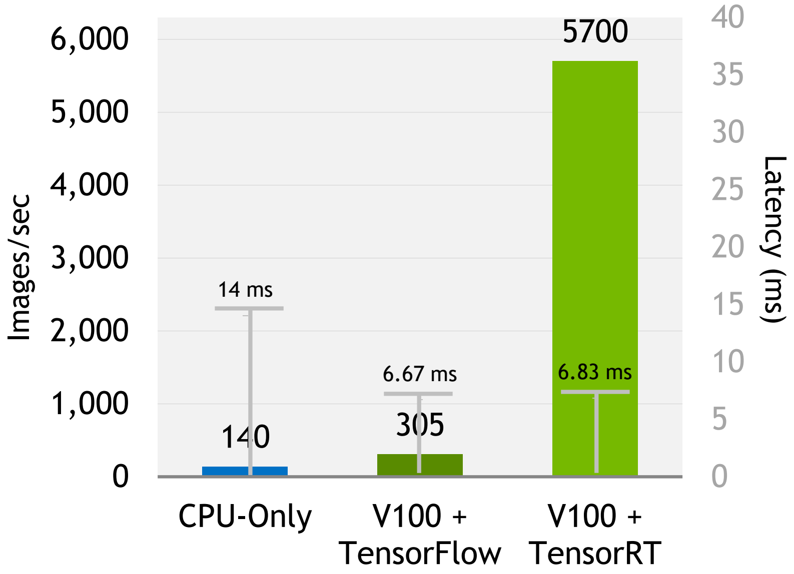
Gary Ji (季光)

# TENSORRT：深度学习推理加速

- 深度学习应用开发的两个阶段
  - 训练：利用训练数据生成和优化网络模型
  - 推理：把网络模型集成到应用程序，输入现实数据，得到推理结果
- TensorRT深度优化了推理的运行效率
  - 自动选取最优kernel
    - 矩阵乘法、卷积有多种CUDA实现方式，根据数据大小和形状自动选取最优实现
  - 计算图优化
    - 通过kernel融合、减少数据拷贝等手段，生成网络的优化计算图
  - 支持fp16/int8
    - 对数值进行精度转换与缩放，充分利用硬件的低精度高通量计算能力

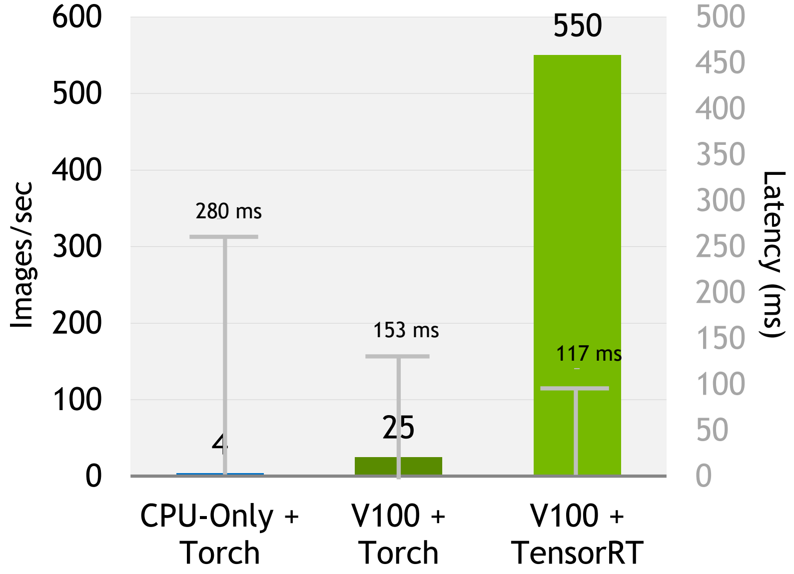
# TENSORRT的加速效果

40x Faster CNNs on V100 vs. CPU-Only
Under 7ms Latency (ResNet50)




Inference throughput (images/sec) on ResNet50. **V100 + TensorRT**: NVIDIA TensorRT (FP16), batch size 39, Tesla V100-SXM2-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On **V100 + TensorFlow**: Preview of volta optimized TensorFlow (FP16), batch size 2, Tesla V100-PCIE-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On. **CPU-Only:** Intel Xeon-D 1587 Broadwell-E CPU and Intel DL SDK. Score doubled to comprehend Intel's stated claim of 2x performance improvement on Skylake with AVX512.

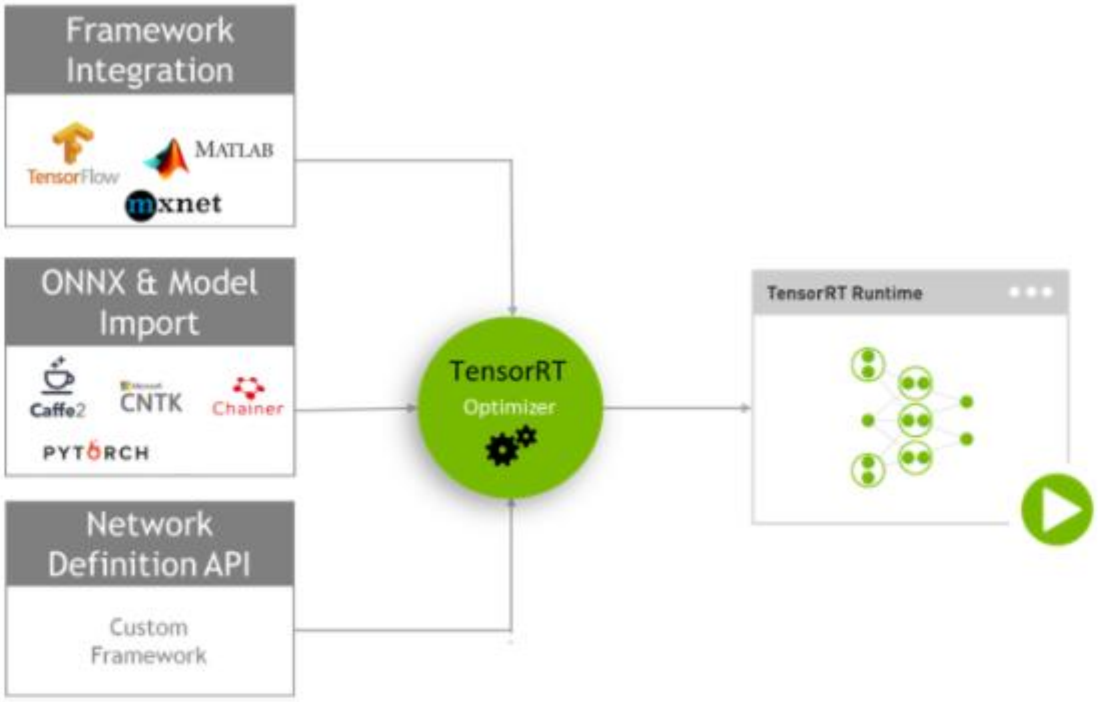
140x Faster Language Translation RNNs on
V100 vs. CPU-Only Inference (OpenNMT)




Inference throughput (sentences/sec) on OpenNMT 692M. **V100 + TensorRT**: NVIDIA TensorRT (FP32), batch size 64, Tesla V100-PCIE-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On. **V100 + Torch**: Torch (FP32), batch size 4, Tesla V100-PCIE-16GB, E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On. **CPU-Only:** Torch (FP32), batch size 1, Intel E5-2690 v4@2.60GHz 3.5GHz Turbo (Broadwell) HT On

# 快速上手TENSORRT

- 通过TensorFlow或MXNet内部集成的TRT
  - 易于使用
  - 未达到最佳效率
- 从现有框架导出模型(ONNX)，再导入TRT
  - 难度适中
  - 兼容性不佳
- 使用TRT C++/Python API自行构造网络
  - 兼容性最强，效率最高
  - 难度最高



# 基本框架

```
import tensorrt as trt

logger = trt.Logger(trt.Logger.WARNING)
builder = trt.Builder(logger)
builder.max_batch_size = 32
builder.max_workspace_size = 10 << 20

network = builder.create_network()
data = network.add_input("data", trt.DataType.FLOAT, (c, h, w))
# ...
# Add network layers
# ...
network.mark_output(outputLayer.get_output(0))

engine = builder.build_cuda_engine(network)
context = engine.create_execution_context()
context.execute_async(bindings=[d_input, d_output])
```

# 卷积网络

- 卷积网络常用于提取特征，作为更复杂网络的前端结构，常被称作backbone
  - 如GoogLeNet, VGG, ResNet
- 卷积网络一般由如下层构成
  - convolution, fully connected (FC)
  - activation, pooling, softmax
  - batch norm (BN), layer norm
  - element wise add/multiply
  - split, concat, padding

# 网络不难搭，参数不好搞

- 以卷积层为例
  - 知道out channel, kernel, padding & stride，就可以调用TRT network definition API，造出相应的层

```
conv0 = network.add_convolution(data, 32, (3,3), trt.Weights(w0), trt.Weights(b0))

conv0.stride = (1, 1)

conv0.padding = (1, 1)
```

- 如何设置weight?
  - 怎样从TF得到?
  - 需不需要做预处理?

# 从TF获取卷积参数，设置在TRT上

- 获取

```
tf.train.Saver(tf.all_variables()).restore(sess, model_checkpoint_path)
tf_args = {}
for i in tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES):
    tf_args[i.name] = sess.run(i)
np.savez('tf_args.npz', **tf_args)
```

- 设置

```
params = np.load('tf_args.npz’)
bag = []
w = params['conv_conv1/W:0'].transpose((3, 2, 0, 1)).reshape(-1)
b = np.zeros(64, dtype=np.float32)
bag += [w, b]
conv1 = network.add_convolution(data, 64, (3,3), w, b)
```

# 谜之3, 2, 0, 1

**virtual void nvinfer1::IConvolutionLayer::setKernelWeights ( Weights weights )** pure virtual

Set the kernel weights for the convolution.

The weights are specified as a contiguous array in `GKCRS` order, where `G` is the number of groups, `K` the number of output feature maps, `C` the number of input channels, and `R` and `S` are the height and width of the filter.

**See Also**

getKernelWeights()

conv_conv1/W:0 (3, 3, 1, 64)
conv_conv2/W:0 (3, 3, 64, 128)
conv_conv3/W:0 (3, 3, 128, 256)

TF的卷积参数：h, w, in_c, out_c
0 1 2 3

TRT的卷积参数：out_c, in_c, h, w
3, 2, 0, 1

# 全连接层

- 设置

```
w = params['dense1/weights:0'].transpose((1, 0)).reshape(-1)
b = params['dense1/biases:0'].reshape(-1)
bag += [w b]
fc1 = network.add_fully_connected(layer.get_output(0), 256, w, b)
```

# 批量归一化层

- TRT未提供BatchNorm层，但提供了更通用的Scale层
- BatchNorm层的定义

$$BN[i,:] = \frac{(in[i,:] - mean[i])}{\sqrt{var[i] + \epsilon}} * gamma[i] + beta[i]$$

- TRT Scale层的定义

$$TRT\_Scale = (in * scale + shift)^{power}$$

- 套用TRT Scale实现BatchNorm

$$scale = \frac{gamma}{\sqrt{var + \epsilon}} \qquad shift = -\frac{mean}{\sqrt{var + \epsilon}} * gamma + beta \qquad power = 1$$

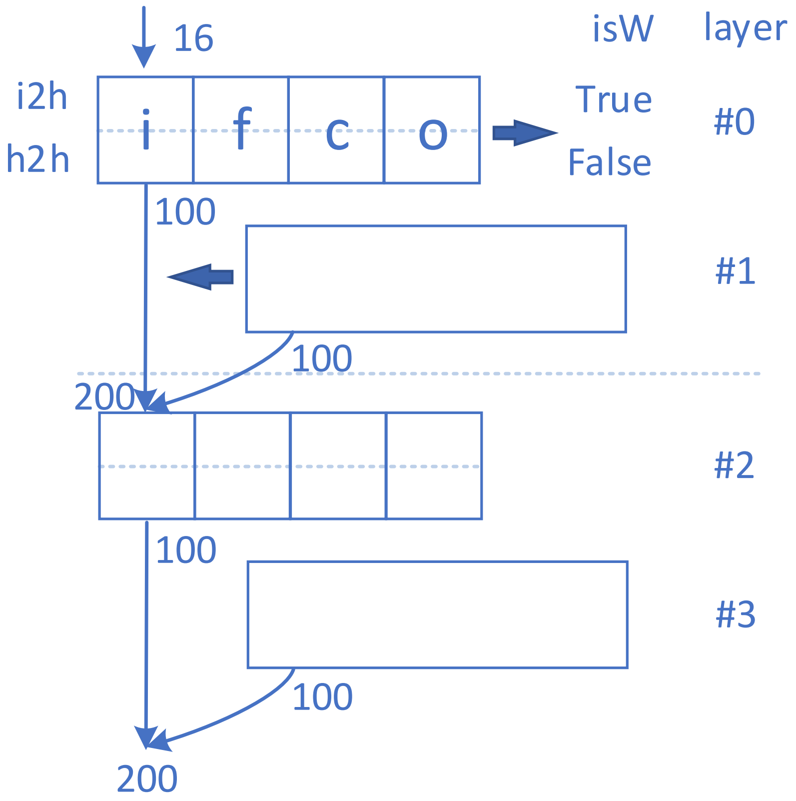
# 批量归一化层：代码示例

```
g0 = params['batchnorm0_gamma'].asnumpy().reshape(-1)
m0 = extra_params['batchnorm0_moving_mean'].asnumpy().reshape(-1)
v0 = extra_params['batchnorm0_moving_var'].asnumpy().reshape(-1)

scale0 = g0 / np.sqrt(v0 + 1e-3)
shift0 = -m0 / np.sqrt(v0 + 1e-3) * g0 +
     params['batchnorm0_beta'].asnumpy().reshape(-1)
power0 = np.ones(len(g0), dtype=np.float32)
bag += [scale0 shift0 power0]

batchNormLayer = network.add_scale(layer.get_output(0), trt.ScaleMode.CHANNEL,
     shift0, scale0, power0)
```

# LSTM层




$$I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i)$$

$$\tilde{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c)$$

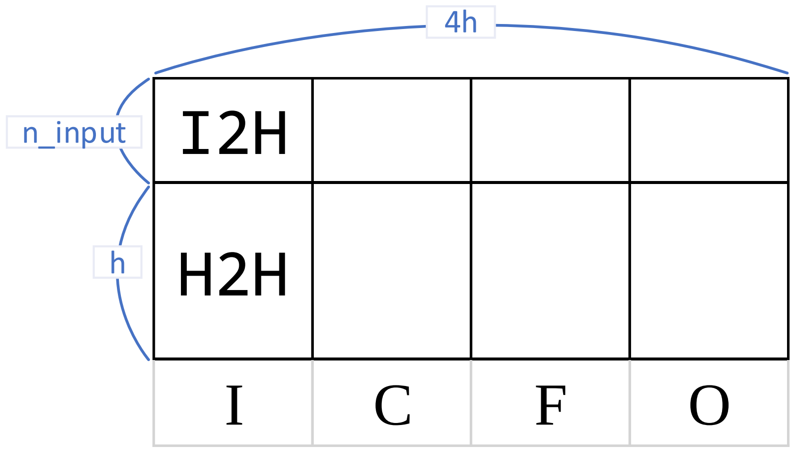
$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f)$$

$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o)$$

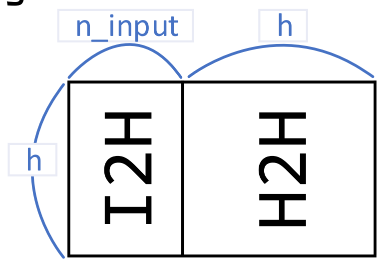
```
lstmLayer.set_weights_for_gate(layer, trt.RNNGateType.INPUT, isW, w_i)
lstmLayer.set_bias_for_gate(layer, trt.RNNGateType.INPUT, isW, b_i)
```

# LSTM层

TF的参数布局




TRT的参数布局



# LSTM层：代码示例

```python
n_input = 512
n_hidden = 512
lstm_encode = network.add_rnn_v2(shuf1.get_output(0), 1, n_hidden, 80, trt.RNNOperation.LSTM)
lstm_encode.direction = trt.RNNDirection.BIDIRECTION
zero_bias = np.zeros(n_hidden, dtype=np.float32)
for i in range(4):
    layer = i // 2
    iW = i % 2
    isW = True if iW == 0 else False

    param_name = 'bidirectional_rnn/{}/basic_lstm_cell/'.format("fw" if layer % 2 == 0 else "bw")
    all_w = [w.transpose((1, 0)).reshape(-1) for w in
        np.split(np.split(params[param_name + 'weights:0'], [n_input])[iW], 4, axis=1)]
    all_b = [w if isW else zero_bias for w in np.split(params[param_name + 'biases:0'], 4)]

    for t,w,b in zip([trt.RNNGateType.INPUT, trt.RNNGateType.CELL, trt.RNNGateType.FORGET, trt.RNNGateType.OUTPUT],
            all_w, all_b):
        lstm_encode.set_weights_for_gate(layer, t, isW, w)
        lstm_encode.set_bias_for_gate(layer, t, isW, b)
```

# 自定义操作：构建PLUGIN

**TRT的扩展机制**

对于TRT不支持的层，可自定义Plugin

**开发步骤**

用cuBLAS/cuDNN/NPP/CUDA C编写定义层上的计算

实现Plugin需要的接口

```
class CustomLayerPlugin: public IPluginExt {
public:
    int getNbOutputs() const override {return 1;}
    Dims getOutputDimensions(int index, const Dims* pInputDim,
            int nInputDim) override {
        return Dims4(pInputDim[0].d[0], 16, 1, 1);
    }
    bool supportsFormat(DataType type, PluginFormat format)const override {
        return type == DataType::kFLOAT && format == PluginFormat::kNCHW;
    }

    void configureWithFormat(const Dims* pInputDim, int nInputDim,
            const Dims* pOutputDim, int nOutputDim,  DataType dataType,
            PluginFormat pluginFormat, int maxBatchSize) override {
        inputDim = *pInputDim;
    }
    ......
};
```

# 构建PLUGIN

```
virtual size_t getWorkspaceSize(int nBatchSize) const override {
    return nBatchSize * 1024;
}

virtual int enqueue(int nBatchSize, const void * const *inputs, void **outputs, void* workspace,
        cudaStream_t stream) override {
   custom_kernel<<<(nBatchSize * inputDim[0] / nBlockSize, nBlockSize>>>(...);
    return 0;
}
...
```

# 使用动态大小的输入

- TRT 6开始支持动态大小的输入：输入数据的大小可变（输出数据的大小也可能随之变化）

```
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
data = network.add_input("data", trt.DataType.FLOAT, (batch_size, 3, 32, -1))
shuf = network.add_shuffle(data)
shuf.first_transpose = (0, 3, 1, 2)
# ...
network.mark_output(outputLayer.get_output(0))

op = builder.create_optimization_profile()
op.set_shape('data', (batch_size, 3, 32, 100), (batch_size, 3, 32, 200), (batch_size, 3, 32, 2000))
config = builder.create_builder_config()
config.add_optimization_profile(op)
engine = builder.build_engine(network, config)
context = engine.create_execution_context()

context.set_binding_shape(0, (batch_size, 3, 32, 100))
context.execute_async(bindings=[d_input0, d_output0])

context.set_binding_shape(0, (batch_size, 3, 32, 200))
context.execute_async(bindings=[d_input1, d_output1])
```

# 使用动态大小的输入

- 要点
  - 填好模板
  - 某些层可以用tensor设置参数
- 特点与限制
  - weight的大小必须固定：否则无法构建网络
  - 一旦启用动态大小，就不能用RNN；反过来一样（即，动态大小与RNN互斥）
- 问题：如何构建动态大小的CNN+静态大小的RNN网络？

# 使用FP16/INT8加速计算

- TRT后端的计算模式
  - 默认使用fp32
  - 可在Volta以及更新的GPU上设置使用fp16/int8，输入数据保持不变
- 使用fp16较为简单
  - builder.fp16_mode = True
- 使用int8需要校正数据集
  - float向int8转换需要量化，校正数据集会让量化尽可能准确

# 总结与建议

- 对于DNN推理应用，TRT能充分释放GPU的计算潜力
- 对于初级用户
    - 推荐试用框架集成的TRT，或者parser导入模型，尝试加速效果
- 对于高级用户
    - 推荐使用网络定义API，实现完全迁移
    - 需要了解参数格式
    - 可能需要自定义Plugin
- 推荐使用fp16/int8计算模式
    - fp16只需略微修改代码，明显提高速度，基本不影响精度
    - int8有更高的计算性能，可能会有精度下降

# NVIDIA GTC

2019 年 12 月 16 日 – 19 日
苏州, 中国


## 沟通

与来自 NVIDIA 和其他业界领先组织的技术专家互动。

## 学习

通过百余场讲座、动手实验和研究海报获取宝贵见解和实践培训。

## 发现

了解 GPU 技术如何为深度学习等重要领域带来重大突破，描绘最新 AI 世界观。


## 创新

共同探索改变世界的颠覆性创新，定义未来。

立即注册， 扫码立享 75 折邀请优惠购票
或使用我的优惠邀请码：XXXXXX
前往 www.nvidia.cn/gtc/ 完成报名

NVIDIA