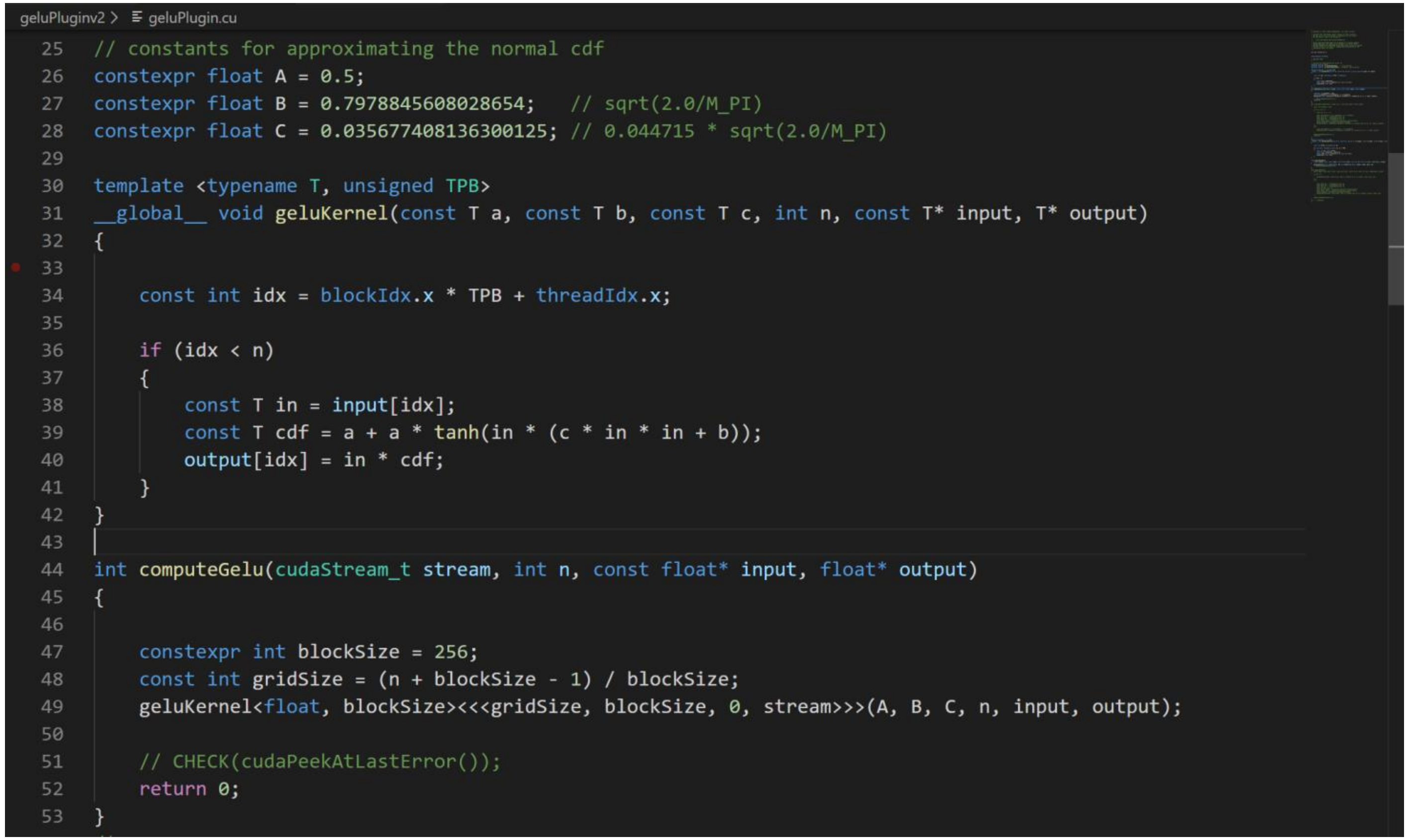
# EXTENDING TENSORRT WITH CUSTOM LAYERS

Dr. Pallab Maji, Senior Solutions Architect – Deep Learning

# CPP & CUDA SETUP

# Prepare the CUDA Kernel

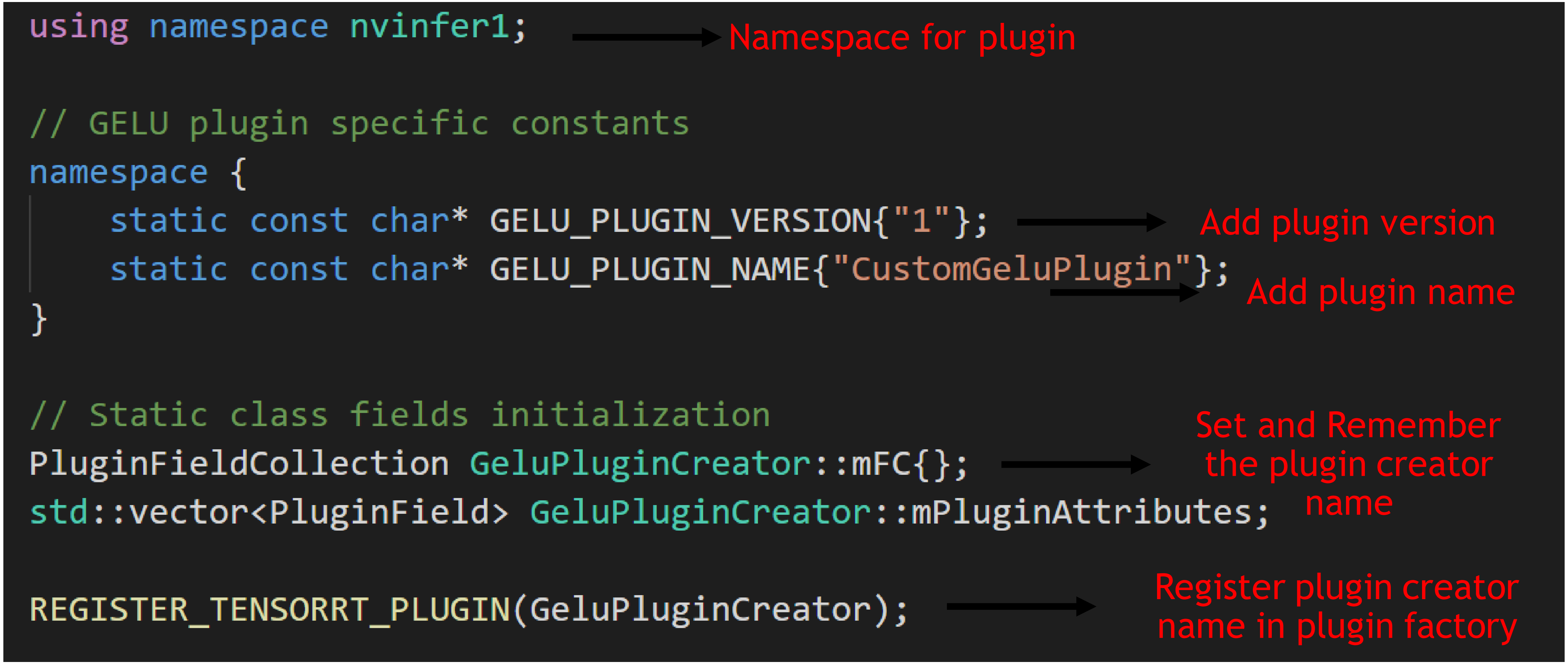
Prepare and Verify the CUDA Kernel for the Layer to be Replaced



```
// constants for approximating the normal cdf
constexpr float A = 0.5;
constexpr float B = 0.7978845608028654;    // sqrt(2.0/M_PI)
constexpr float C = 0.035677408136300125; // 0.044715 * sqrt(2.0/M_PI)

template <typename T, unsigned TPB>
__global__ void geluKernel(const T a, const T b, const T c, int n, const T* input, T* output)
{

    const int idx = blockIdx.x * TPB + threadIdx.x;

    if (idx < n)
    {
        const T in = input[idx];
        const T cdf = a + a * tanh(in * (c * in * in + b));
        output[idx] = in * cdf;
    }
}

int computeGelu(cudaStream_t stream, int n, const float* input, float* output)
{

    constexpr int blockSize = 256;
    const int gridSize = (n + blockSize - 1) / blockSize;
    geluKernel<float, blockSize><<<gridSize, blockSize, 0, stream>>>(A, B, C, n, input, output);

    // CHECK(cudaPeekAtLastError());
    return 0;
}
```

# custom plugin – Setup

Look into the CPP code of the sample plugin



```
using namespace nvinfer1;

// GELU plugin specific constants
namespace {
    static const char* GELU_PLUGIN_VERSION{"1"};
    static const char* GELU_PLUGIN_NAME{"CustomGeluPlugin"};
}

// Static class fields initialization
PluginFieldCollection GeluPluginCreator::mFC{};
std::vector<PluginField> GeluPluginCreator::mPluginAttributes;

REGISTER_TENSORRT_PLUGIN(GeluPluginCreator);
```

Namespace for plugin

Add plugin version

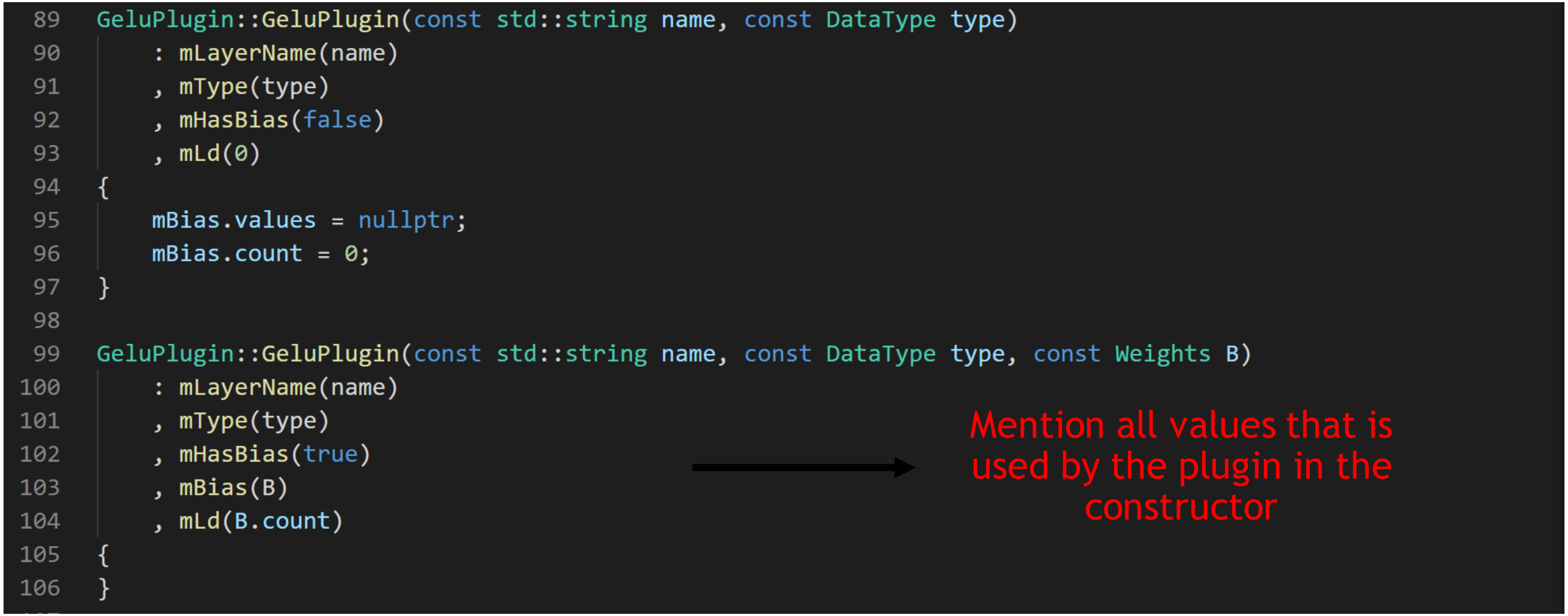
Add plugin name

Set and Remember the plugin creator name

Register plugin creator name in plugin factory

# custom plugin - Constructors

## Look into the CPP code of the sample plugin (contd.)

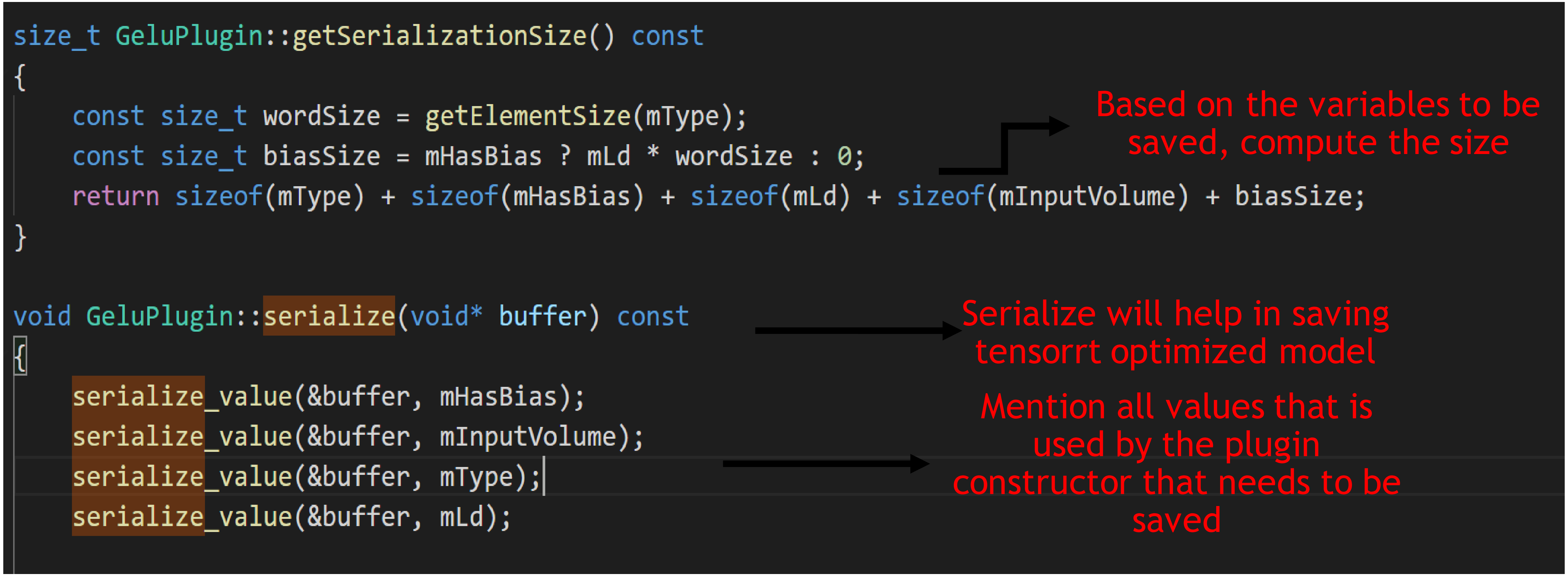


```cpp
GeluPlugin::GeluPlugin(const std::string name, const DataType type)
    : mLayerName(name)
    , mType(type)
    , mHasBias(false)
    , mLd(0)
{
    mBias.values = nullptr;
    mBias.count = 0;
}

GeluPlugin::GeluPlugin(const std::string name, const DataType type, const Weights B)
    : mLayerName(name)
    , mType(type)
    , mHasBias(true)
    , mBias(B)
    , mLd(B.count)
{
}
```

Mention all values that is used by the plugin in the constructor

# custom plugin - Serialize

## Look into the CPP code of the sample plugin (contd.)



```cpp
size_t GeluPlugin::getSerializationSize() const
{
    const size_t wordSize = getElementSize(mType);
    const size_t biasSize = mHasBias ? mLd * wordSize : 0;
    return sizeof(mType) + sizeof(mHasBias) + sizeof(mLd) + sizeof(mInputVolume) + biasSize;
}

void GeluPlugin::serialize(void* buffer) const
{
    serialize_value(&buffer, mHasBias);
    serialize_value(&buffer, mInputVolume);
    serialize_value(&buffer, mType);
    serialize_value(&buffer, mLd);
```

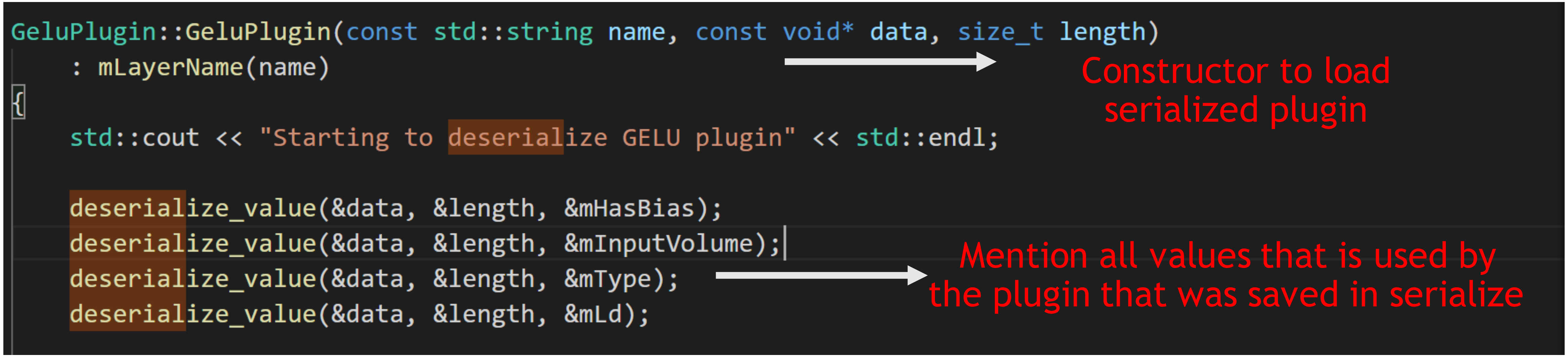
Based on the variables to be saved, compute the size

Serialize will help in saving tensorrt optimized model

Mention all values that is used by the plugin constructor that needs to be saved

# custom plugin – Constructor to DE-SerialiZE

Look into the CPP code of the sample plugin (contd.)



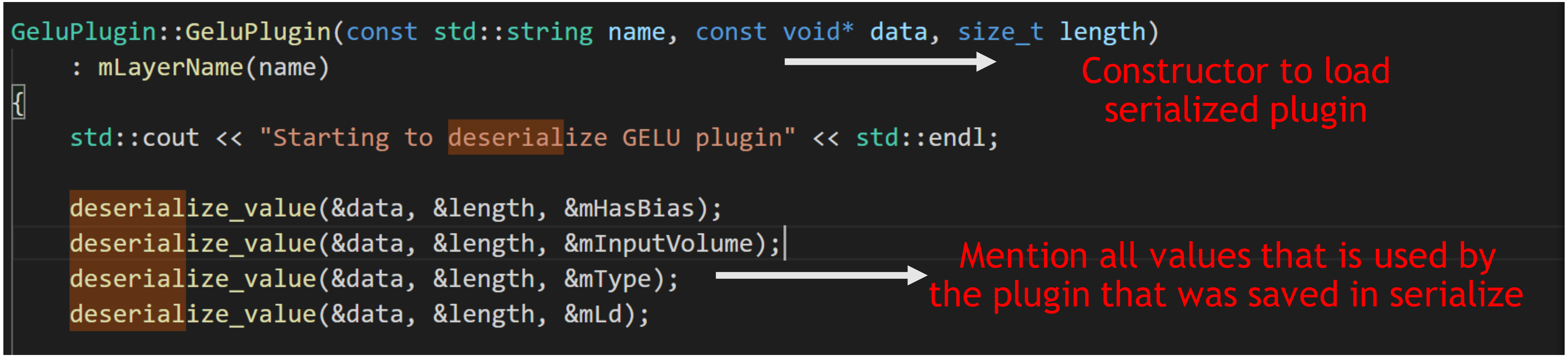
```
GeluPlugin::GeluPlugin(const std::string name, const void* data, size_t length)
    : mLayerName(name)
{
    std::cout << "Starting to deserialize GELU plugin" << std::endl;

    deserialize_value(&data, &length, &mHasBias);
    deserialize_value(&data, &length, &mInputVolume);
    deserialize_value(&data, &length, &mType);
    deserialize_value(&data, &length, &mLd);
```

Constructor to load serialized plugin

Mention all values that is used by the plugin that was saved in serialize

# custom plugin – Constructor to DE-SerialiZE

Look into the CPP code of the sample plugin (contd.)



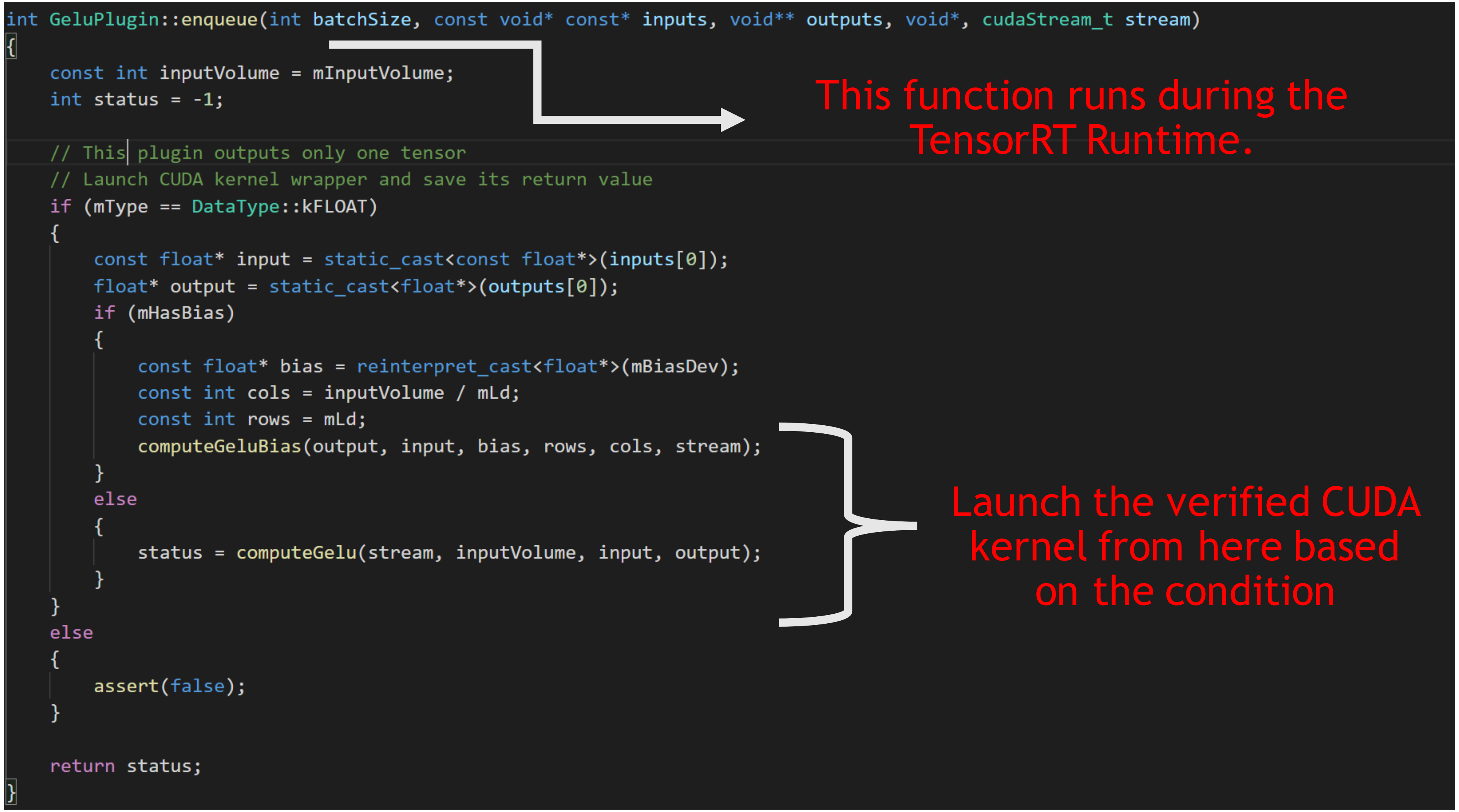
```cpp
GeluPlugin::GeluPlugin(const std::string name, const void* data, size_t length)
    : mLayerName(name)
{
    std::cout << "Starting to deserialize GELU plugin" << std::endl;

    deserialize_value(&data, &length, &mHasBias);
    deserialize_value(&data, &length, &mInputVolume);
    deserialize_value(&data, &length, &mType);
    deserialize_value(&data, &length, &mLd);
```

Constructor to load serialized plugin

Mention all values that is used by the plugin that was saved in serialize

# custom plugin – ENQUEUE Function

## Look into the CPP code of the sample plugin (contd.)



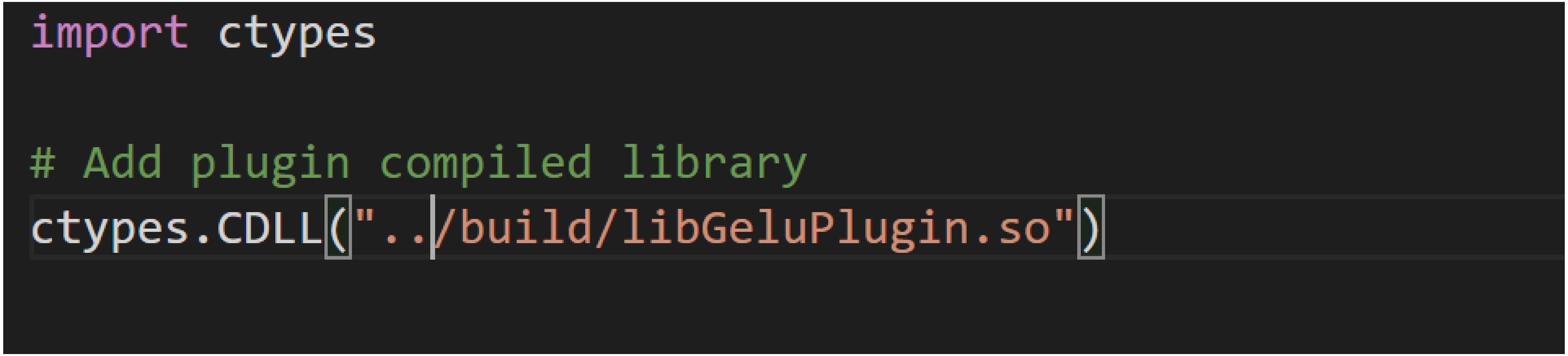
```cpp
int GeluPlugin::enqueue(int batchSize, const void* const* inputs, void** outputs, void*, cudaStream_t stream)
{
    const int inputVolume = mInputVolume;
    int status = -1;

    // This plugin outputs only one tensor
    // Launch CUDA kernel wrapper and save its return value
    if (mType == DataType::kFLOAT)
    {
        const float* input = static_cast<const float*>(inputs[0]);
        float* output = static_cast<float*>(outputs[0]);
        if (mHasBias)
        {
            const float* bias = reinterpret_cast<float*>(mBiasDev);
            const int cols = inputVolume / mLd;
            const int rows = mLd;
            computeGeluBias(output, input, bias, rows, cols, stream);
        }
        else
        {
            status = computeGelu(stream, inputVolume, input, output);
        }
    }
    else
    {
        assert(false);
    }

    return status;
}
```

This function runs during the TensorRT Runtime.

Launch the verified CUDA kernel from here based on the condition

# BUILD Tensorrt Plugin with CMAKE

Build the plugin to generate a library then verify in Python

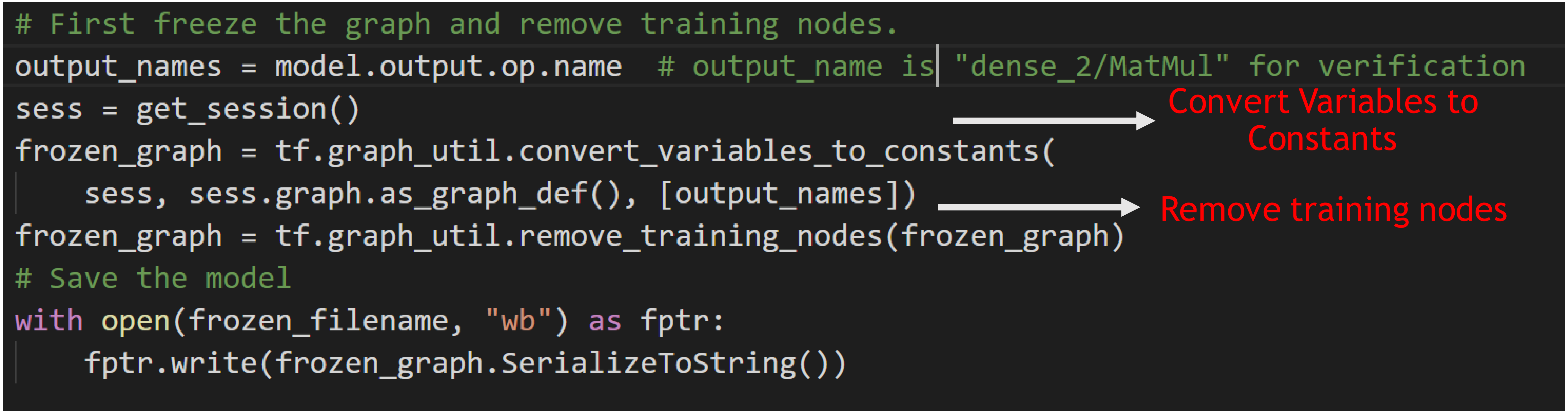


```python
import ctypes

# Add plugin compiled library
ctypes.CDLL("../build/libGeluPlugin.so")
```

# PYTHON SETUP

# Freeze the Graph

Load saved model, Remove Training Nodes, Convert Variables to Constants & Save to Disk

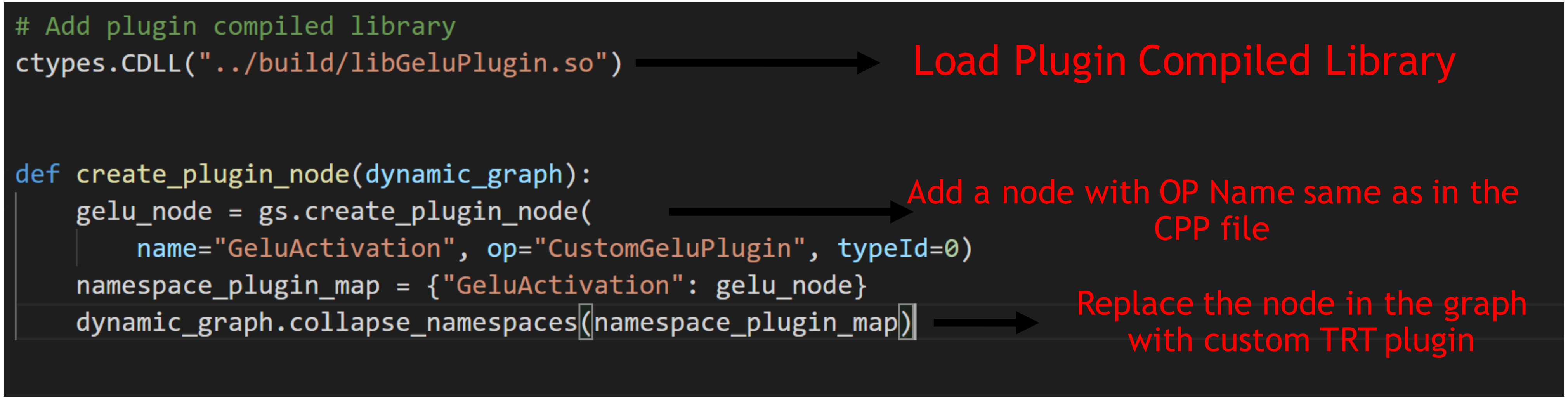


```
# First freeze the graph and remove training nodes.
output_names = model.output.op.name  # output_name is "dense_2/MatMul" for verification
sess = get_session()
frozen_graph = tf.graph_util.convert_variables_to_constants(
    sess, sess.graph.as_graph_def(), [output_names])
frozen_graph = tf.graph_util.remove_training_nodes(frozen_graph)
# Save the model
with open(frozen_filename, "wb") as fptr:
    fptr.write(frozen_graph.SerializeToString())
```

Convert Variables to Constants

Remove training nodes

# Convert FROZEN File to UFF

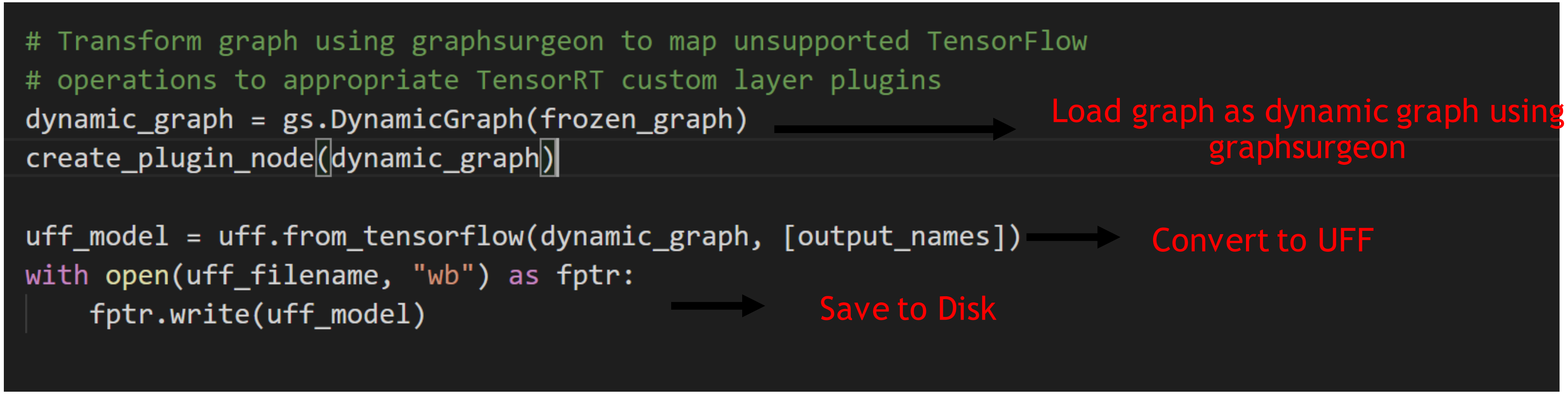


```python
# Add plugin compiled library
ctypes.CDLL("../build/libGeluPlugin.so")


def create_plugin_node(dynamic_graph):
    gelu_node = gs.create_plugin_node(
        name="GeluActivation", op="CustomGeluPlugin", typeId=0)
    namespace_plugin_map = {"GeluActivation": gelu_node}
    dynamic_graph.collapse_namespaces(namespace_plugin_map)
```

Load Plugin Compiled Library

Add a node with OP Name same as in the CPP file

Replace the node in the graph with custom TRT plugin



```python
# Transform graph using graphsurgeon to map unsupported TensorFlow
# operations to appropriate TensorRT custom layer plugins
dynamic_graph = gs.DynamicGraph(frozen_graph)
create_plugin_node(dynamic_graph)

uff_model = uff.from_tensorflow(dynamic_graph, [output_names])
with open(uff_filename, "wb") as fptr:
    fptr.write(uff_model)
```

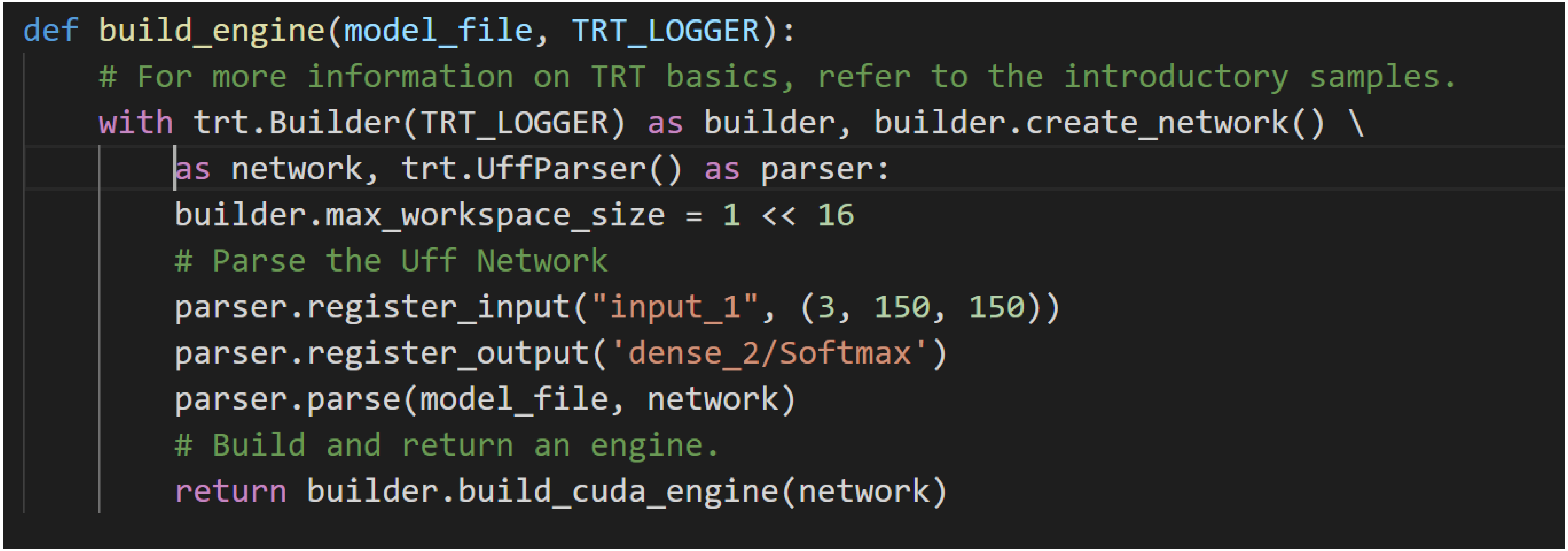
Load graph as dynamic graph using graphsurgeon

Convert to UFF

Save to Disk

# Build & Infer

Build the TensorRT engine as usual and do Inference



```python
def build_engine(model_file, TRT_LOGGER):
    # For more information on TRT basics, refer to the introductory samples.
    with trt.Builder(TRT_LOGGER) as builder, builder.create_network() \
        as network, trt.UffParser() as parser:
        builder.max_workspace_size = 1 << 16
        # Parse the Uff Network
        parser.register_input("input_1", (3, 150, 150))
        parser.register_output('dense_2/Softmax')
        parser.parse(model_file, network)
        # Build and return an engine.
        return builder.build_cuda_engine(network)
```