Should we use the DeepStream SDK or write a Python pipeline ourselves?

Lieu Zheng Hong

2nd November 2020

Introduction

The DeepStream SDK is a powerful end-to-end pipeline that claims to make development easier. Here I examine the pros and cons of DeepStream and conclude that we do not need it for our use case. We can get the performance

DeepStream will be helpful when dealing with complex multi-GPU, multi-stream pipelines

DeepStream is a powerful pipeline that can handle complex multi-GPU, multi-stream setups. Because DeepStream is based on GStreamer (written in C), I/O becomes blazing fast. However, this comes at a cost of **complexity, flexibility, and extensibility. DeepStream SDK locks us into its particular GStreamer-based pipeline, which leads to several disadvantages:

DeepStream has a steep learning curve

DeepStream is heavily based on GStreamer, and as such inherits many of its abstractions. I can attest that the GStreamer abstraction is not easy to learn because it is written in C and uses a lot of C idioms. This is further complicated by the fact that Python is a garbage-collected language and C is not, so even though there are Python bindings provided, one still has to write code to manually allocate and free memory. This will make subsequent handover to the TPs quite difficult.

We don't need to move to DeepStream to get performance benefits

The main reason for using DeepStream is better performance. The DeepStream SDK may give us performance benefits in two ways.

Firstly, the model will be converted to TensorRT, and TensorRT will almost certainly deliver better performance. This may be important if we want our model to run in real time.

Secondly, I/O processing using GStreamer will be much faster than I/O processing with Python.

However, **as the performance bottleneck is in the inference and not the I/O**, we can convert the model to TensorRT without having to use the DeepStream pipeline. This will give us the performance benefits without incurring a large complexity cost There is a TensorRT Python API we can use for that purpose.

DeepStream does not officially support the Intel RealSense camera

The Intel RealSense camera outputs depth data in the Z16 format, which is not supported by DeepStream (see link here and here).

While there is a third party library for RealSense to work with DeepStream, my research seemed to suggest that it did not work well on the Jetson NX. We are trying to develop software at the intersection of three niche technologies (JetsonNX + DeepStream + RealSense), which makes troubleshooting difficult.

DeepStream offers communication interfaces, but Python already has good support for RabbitMQ

As an end-to-end pipeline, DeepStream offers an IoT integration interface with Kafka, MQTT and AMQP and turnkey integration with AWS IoT and Microsoft Azure IoT.

This is useful, but Python already has good support for RabbitMQ. I have already written a sample RabbitMQ sender and receiver in Python available the repo

DeepStream allows us to deploy at scale, but we do not need to deploy at scale

The website writes:

> By using DeepStream, you can deploy at scale and manage containerized apps with Kubernetes and Helm Charts.

We do not need to deploy at scale, so the advantage of DeepStream is not relevant for us.

Conclusion

The most important consideration in any software engineering project is complexity. We should avoid unnecessary complexity at all costs because it acts as a drag on development, making reasoning about and adding to our codebase much slower and more difficult.

DeepStream is a powerful (and very complex) pipeline geared for high-throughput deployments. There seems to be little reason to use DeepStream for our use case, as it adds complexity and reduce our flexibility for not much return. Given the relatively small scope of our deployment and the tight time constraints, I would thus highly recommend that we build out the pipeline with Python.