Deployment of Object Detection Model

for NVidia GPUs
with use PyTorch & TensorRT

by

Alexey Gladyshev, Daniil Roman

Contacts:

Alexey Gladyshev
e-mail: *alexey.gladyshev@itlab.unn.ru*
telegram: *@back2debug*

Daniil Roman
e-mail: *danroman17397@gmail.com*
telegram: *@daniilroman17*

Введение.

Данная работа была выполнена в рамках участия в соревнованиях “Contest from Huawei”, которые проводились для студентов магистратуры 2 курса в ННГУ.

Task

- Take any modern object/keypoint detection model
- Deploy to the target device by using any IE discussed in the lecture
- Document and describe all the steps and challenges (if you faced any)
- Use all the possible selected IE knobs to get the best performance (for latency or throughput scenarios, or both if applicable)
- Document your performance findings
- Share your notes and code over github repo
- Small app demonstrating some use case is a plus

1. Постановка задачи.

Необходимо было рассмотреть задачу детектирования объектов или ключевых точек. Использовать для ее решения любую из общедоступных моделей машинного обучения, после чего развернуть ее на целевом устройстве с помощью одного из доступных Inference Engines.

2. Используемая модель и данные.

В качестве задачи была выбрана задача детектирования объектов. Для ее решения была выбрана сеть SSD: Single Shot MultiBox Detector[6].

Реализацию и веса для данной модели можно найти в открытом доступе[7][8].

Модель обучалась на датасете COCO 2017 Dataset. Для скачивания можно воспользоваться данной статьей[9].

Веса обученной модели предоставляются в формате FP16 (уже производился fine tuning после перехода от FP32). Поэтому нам было достаточно только валидационной выборки, т.к. дообучение производить не требовалось.

В качестве целевой платформы, для которой будет оптимизироваться данная модель, была выбрана платформа NVidia GPU. Целевая операционная система - Linux (Ubuntu 16.04).

3. Используемые инструменты.

Для выполнения поставленной задачи был использован следующий стэк инструментов:

- Язык программирования - Python
- Фреймворк машинного - PyTorch[1]
- Библиотека компьютерного зрения - OpenCV[2]
- Inference Engine - TensorRT[3]
- Инструмент для конвертации PyTorch в формат TensorRT - torch2trt[4][5]

Для деталей установки отдельных компонент обратитесь в приложение.

Так как целевой платформой была выбрана NVidia GPU, в качестве IE был выбран TensorRT.

К некоторым наиболее очевидным методам оптимизации, которые реализованы в TensorRT, можно отнести следующее:

- Удаление слоев, выходы которых не используются.
- Определенные операции (например, такие как Convolution, Bias и ReLU) объединяются в один слой. Это связано с тем, что Bias и ReLU имеют линейную сложность от размера данных и время выполнения тратится на считывание этих самых данных. Тогда, если прибавлять Bias и применять ReLU сразу к результату свертки, то не нужно будет тратить время на повторное считывание данных.
- Также сама операция свертки оптимизируется за счет применения различных алгоритмов ее вычисления. Разные алгоритмы могут автоматически подбираться в зависимости от размера батча.
- Также возможна работа с пониженной точностью FP16 и INT8. Для настройки весов в INT8 применяется процедура автоматической калибровки.

Так как выбранная нами модель SSD практически целиком состоит из сверточных слоев, использование в качестве IE TensorRT позволит получить хороший прирост в производительности.

4. Эксперименты.

Для экспериментов была использована следующая конфигурация аппаратного обеспечения:

- CPU - Intel Core i5
- GPU - MSI nVidia GeForce GTX 1050TI

Программное обеспечение:

- Ubuntu 16.04
- CUDA 10.2
- Nvidia Driver 440.33.01
- cuDNN 7.6.5.32
- TensorRT 6.0.1.8

Эксперименты проводились для следующих режимов запуска модели:

- int8_trt - представление весов модели в формате INT8 и использование оптимизаций TensoRT. Для калибровки использовалась выборка, состоящая из 200 сэмплов. Сэмплы были взяты случайным образом из валидационного множества COCO 2017.
- fp16_trt - представление весов модели в формате FP16 и использование оптимизаций TensorRT.
- fp32_trt - представление весов модели в формате FP32 и использование оптимизаций TensorRT.
- fp16 - представление весов модели в формате FP16 (использование только PyTorch)
- fp32 - представление весов модели в формате FP32 (использование только PyTorch)

Производительность каждого из режимов запуска тестировалась для различных размеров батча: 1, 4, 8, 16, 32, 64.

Это связано с тем, что TensorRT “под капотом” может использовать различные оптимизации для сверток в зависимости от размеров батча.

Результаты проведенных экспериментов приведены в таблицах ниже.

4.1. Model Precision.

Сначала проверялась точность полученных инференс-моделей, чтобы подтвердить, что после ускорения работы, не ухудшается качество предсказаний.

Таблица 1. Точность работы модели.

Model precision (mAP @[IoU=0.50:0.95])					
batch size	**int8_trt**	**fp16_trt**	**fp32_trt**	**fp16**	**fp32**
1	0,2479512305	0,2502465936	0,2501966288	0,2500865364	0,2501965606
4	0,2476422243	0,2502465977	0,25019664	0,2500865364	0,2501965606
8	0,2476422243	0,2502464747	0,2501967397	0,2500865302	0,2501965604
16	0,2476422243	0,2502466	0,250196634	0,2500864217	0,2501964825
32	0,2479280052	0,2502466004	0,250196634	0,2500864484	0,2501965506
64	0,2477379447	0,2502464747	0,2501967399	0,2500871185	0,2501965115

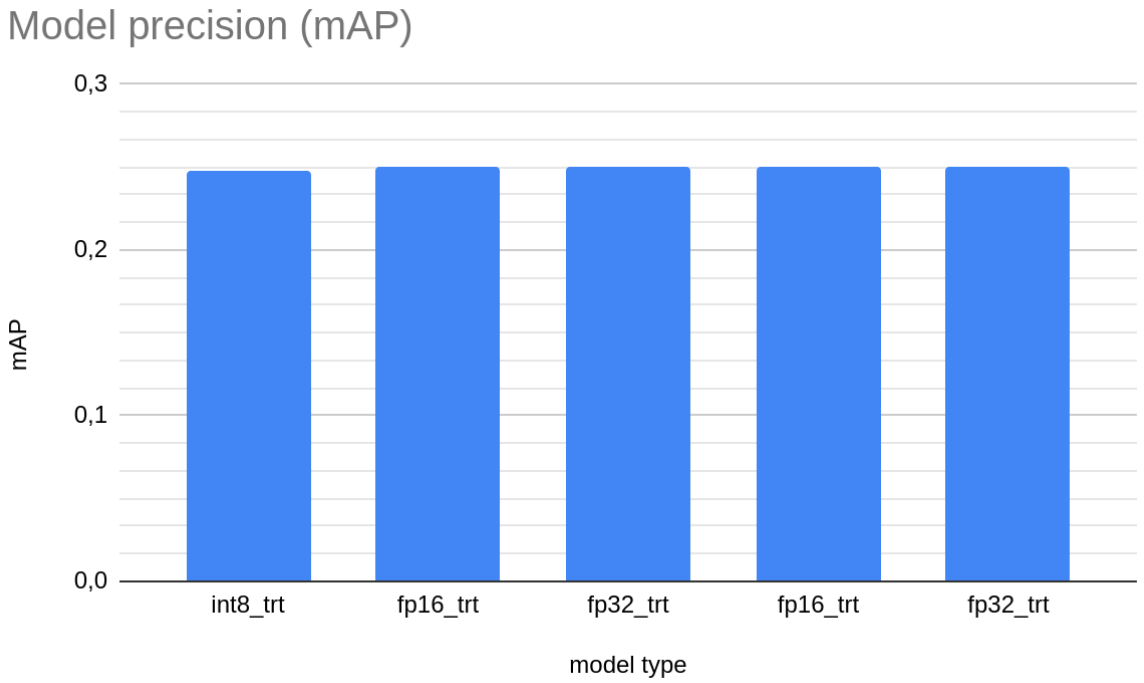

Рисунок 1. Средняя точность для каждого типа весов.

Видно, что при использовании формата INT8 точность модели немного ухудшается, но это было вполне ожидаемо (график специально сделан с таким масштабом, чтобы подчеркнуть, что ухудшение точности минимально). Разница в точности не слишком существенна. Это достигается благодаря используемым алгоритмам калибровки весов.

Также стоит обратить внимание на то, что при переходе от FP32 к FP16 точность модели никак не меняется, хотя она должна была бы ухудшится. Этого не происходит из-за того, что предоставляемые веса уже были в формате FP16. Авторы после тренировки исходной модели в FP32 перевели обученные веса в формат FP16 и дотренировали их до исходных значений точности. Если бы веса были в формате FP32, то после перехода к FP16 необходимо было бы проводить fine tuning, чтобы вернуть прежнюю точность.

4.2. Model Performance (time per sample).

Следующая таблица показывает время, затрачиваемое на инференс сети для обработки одного сэмпла.

Таблица 2. Затрачиваемое время на обработку одного сэмпла.

Model time per sample (sec)					
batch size	**int8_trt**	**fp16_trt**	**fp32_trt**	**fp16**	**fp32**
1	0,00056	0,00059	0,00059	0,00646	0,00677
4	0,00022	0,00025	0,00024	0,00159	0,00171
8	0,00011	0,00014	0,00014	0,00085	0,00088
16	0,00008	0,00012	0,00013	0,00044	0,00046
32	0,00008	0,00016	0,00016	0,00022	0,00026
64	0,0001	0,00025	0,00025	0,00056	0,00092

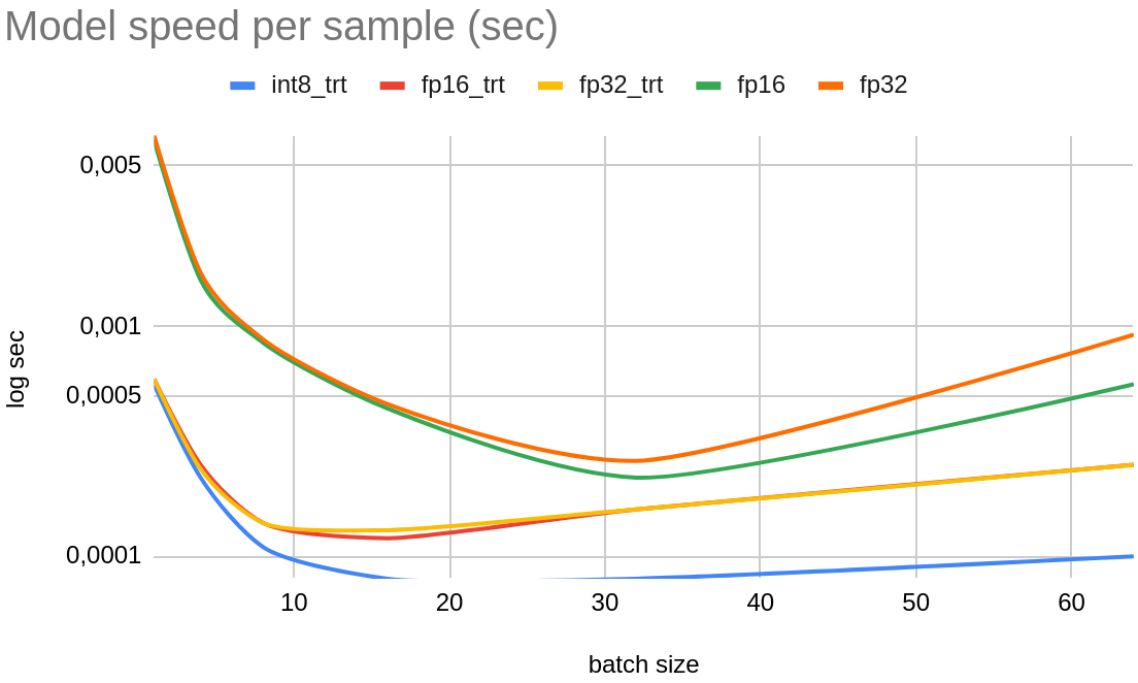

Рисунок 2. Затрачиваемое время на обработку одного сэмпла (логарифмическая шкала).

Из графиков видно, что скорость обработки соответствует предположению: чем “легче” веса, тем меньше нужно времени на обработку одного сэмпла.

Также можно заметить, что для каждого типа весов есть определенное значение размера батча, при котором на обработку одного сэмпла тратится наименьшее количество времени. Это полезно учитывать в тех случаях, когда запросы можно объединять в бачти, чтобы сделать процесс обработки более эффективным (например, когда данные от клиентов приходят на сервер).

4.3. Model Performance (“throughput”).

Таблица ниже основана на предыдущей таблице и также отражает производительность модели, но в качестве метрики используется FPS (когда размер батча отличен от единицы, значение fps считается, как отношение общего времени, затраченного на обработку батча, к количеству сэмплов в этом батче), т.е. значения таблицы отражают пропускную способность сети в секунду.

Таблица 3. Производительность модели (FPS)

Model throughput (fps)					
batch size	**int8_trt**	**fp16_trt**	**fp32_trt**	**fp16**	**fp32**
1	1785,714286	1694,915254	1694,915254	154,7987616	147,7104
4	4545,454545	4000	4166,666667	628,9308176	584,7953
8	9090,909091	7142,857143	7142,857143	1176,470588	1136,363
16	12500	8333,333333	7692,307692	2272,727273	2173,913
32	12500	6250	6250	4545,454545	3846,153
64	10000	4000	4000	1785,714286	1086,956

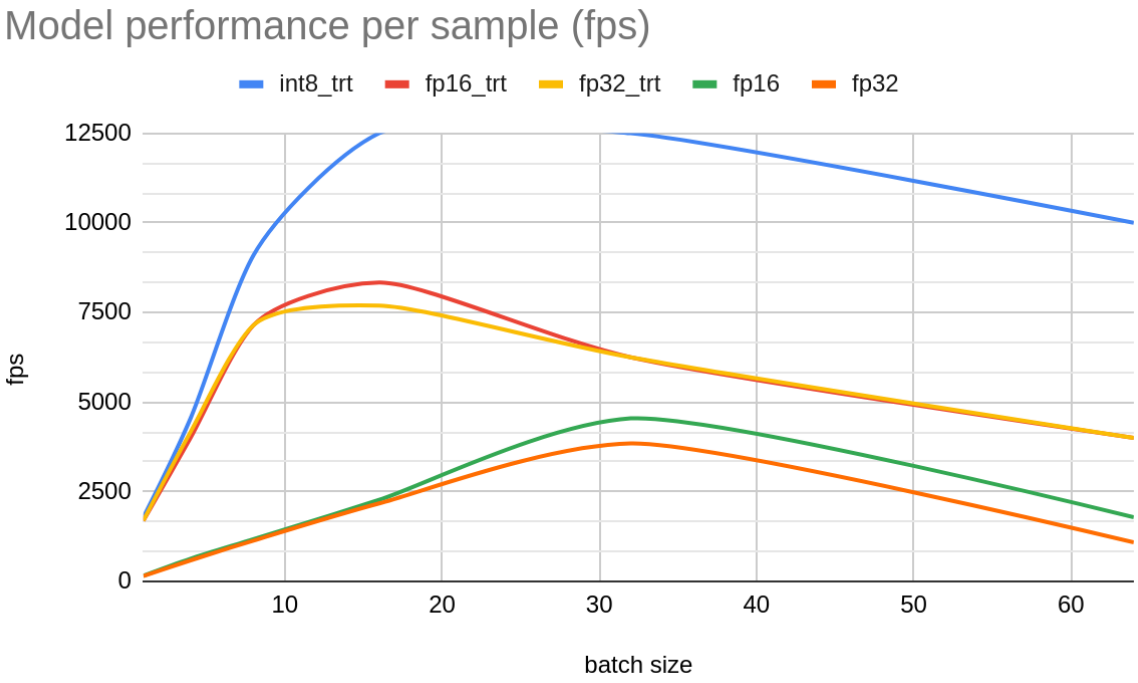

Рисунок 3. Производительность модели (FPS).

В целом ничего нового. Далее эти данные использовались для следующей таблицы, которая отражает ускорение пропускной способности модели, полученное после применения TensorRT.

4.4. Model performance (acceleration).

Предыдущая таблица оперирует абсолютными значениями производительности, но иногда бывает полезно взглянуть на относительный выигрыш, который получается за счет оптимизации инференса.

Ускорение считалось относительно значений скорости работы модели FP16, которая использовала чистый PyTorch для работы. Это обуславливается тем, что точность моделей FP16 и FP32 одинаковая (на валидационном множестве), а также тем, что переход от FP32 к FP16 уже дает небольшое ускорение.

Таким образом формула для вычисления значений таблицы выглядит следующим образом:

$$acceleration(batch\ size) = \frac{FPS_{TensorRT\ model}(batch\ size)}{FPS_{PyTorch\ model\ FP16}(batch\ size)}$$

Таблица 4. Достигнутое ускорение обработки.

Inference acceleration			
batch size	**int8_trt**	**fp16_trt**	**fp32_trt**
1	11,53571429	10,94915254	10,94915254
4	7,227272727	6,36	6,625
8	7,727272727	6,071428571	6,071428571
16	5,5	3,666666667	3,384615385
32	2,75	1,375	1,375
64	5,6	2,24	2,24

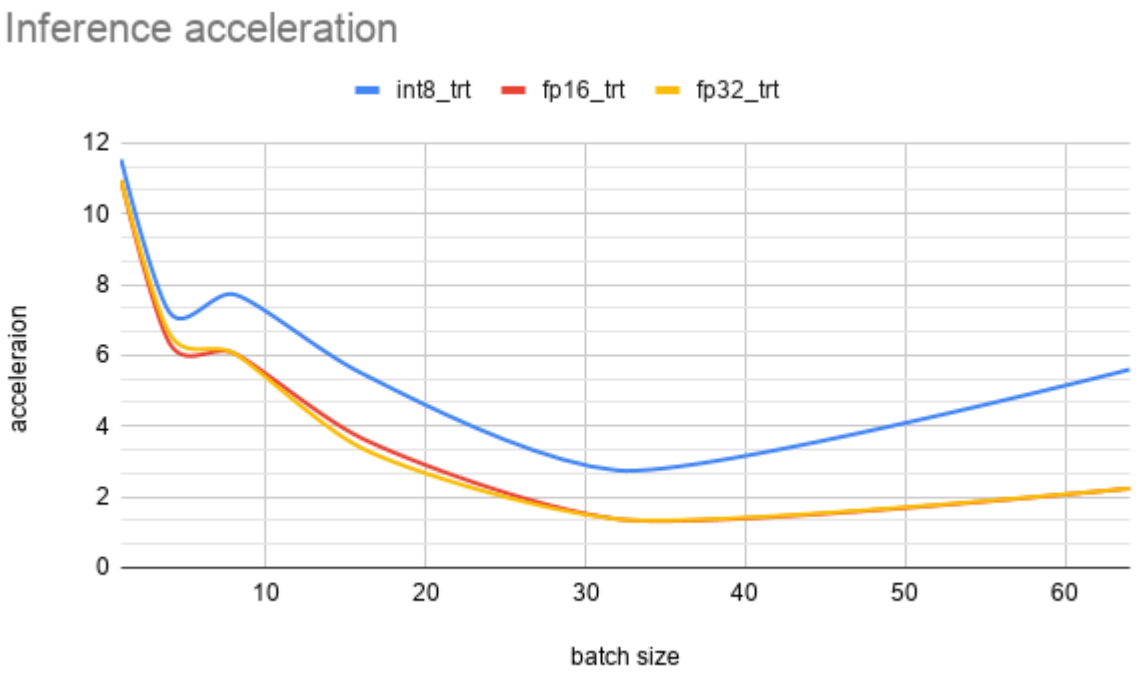

Рисунок 4. Достигнутое ускорение обработки.

Из графиков видно, что наибольшее ускорение достигается на небольших размерах батча. Когда размер равен единице - максимальное. Это особенно полезно для приложений реального времени, в которых данные поступают последовательно и требуют незамедлительной обработки.

Также это позволяет запускать относительно “тяжелые” модели на различных embedded модулях (nvidia jetson agx xavier, nvidia jetson nano) так, чтобы они отрабатывали за приемлемое время.

В целом результаты всех проведенных экспериментов соответствуют ожиданиям.

5. Демонстрационное приложение.

В качестве демонстрационного приложения, в котором могут быть применены результаты проделанной работы, была выбрана демонстрация возможной системы ADAS.

Одной из систем, в которых может найти свое применение задача детектирования объектов, является система экстренного торможения (AEB).

Это система, которая пытается предотвратить дорожно-транспортное происшествие путем включения в экстренной ситуации тормозов автомобиля независимо от водителя.

Сканируя пространство впереди движущегося автомобиля и используя данные о его скорости и траектории движения система оценивает вероятность столкновения.

С помощью камеры мы можем производить это самое сканирование и определять, есть ли перед нами объекты или нет.

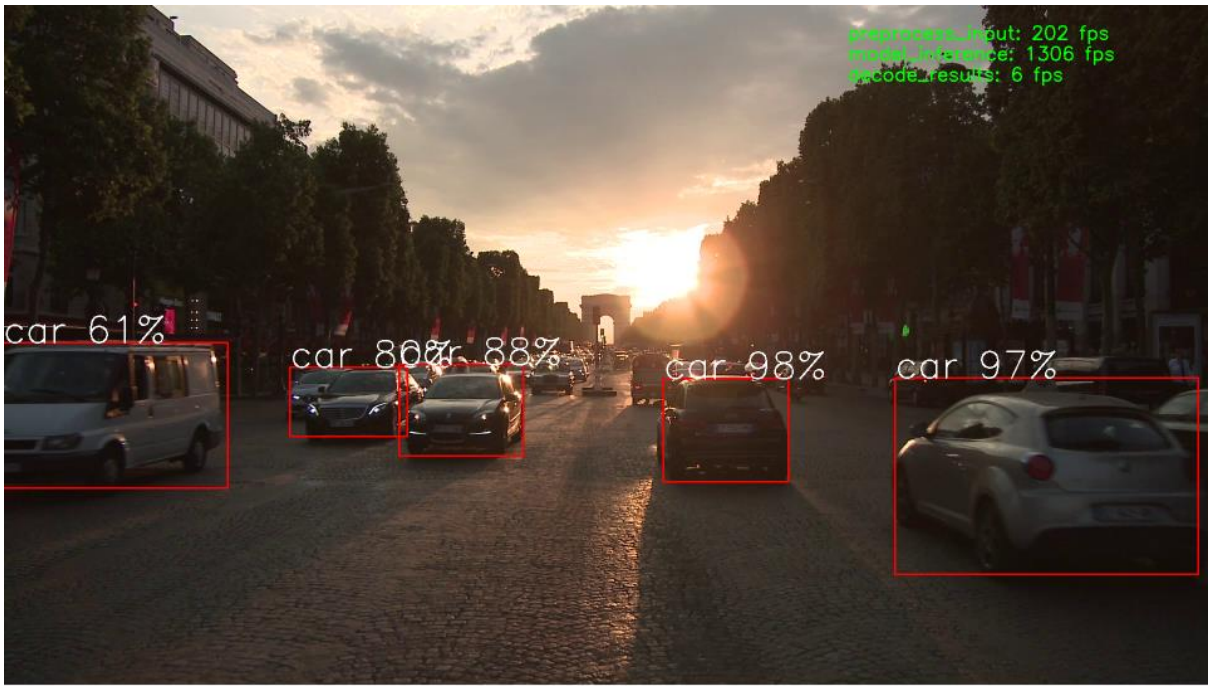

Рисунок 5. Детектирование объектов на кадре.

Но одного наличия объекта в кадре мало, необходимо также уметь оценивать расстояние до него.

В этом может помочь перспективная проекция изображения в так называемый bird’s eye view формат.

Матрицу для выполнения данного преобразования можно подобрать с помощью функции getPerspectiveTransform из библиотеки OpenCV. Однако, приведенный подход будет работать с некоторой степенью погрешности ввиду ручной настройки проекции. Демонстрация данного подхода приведена на рисунке ниже.

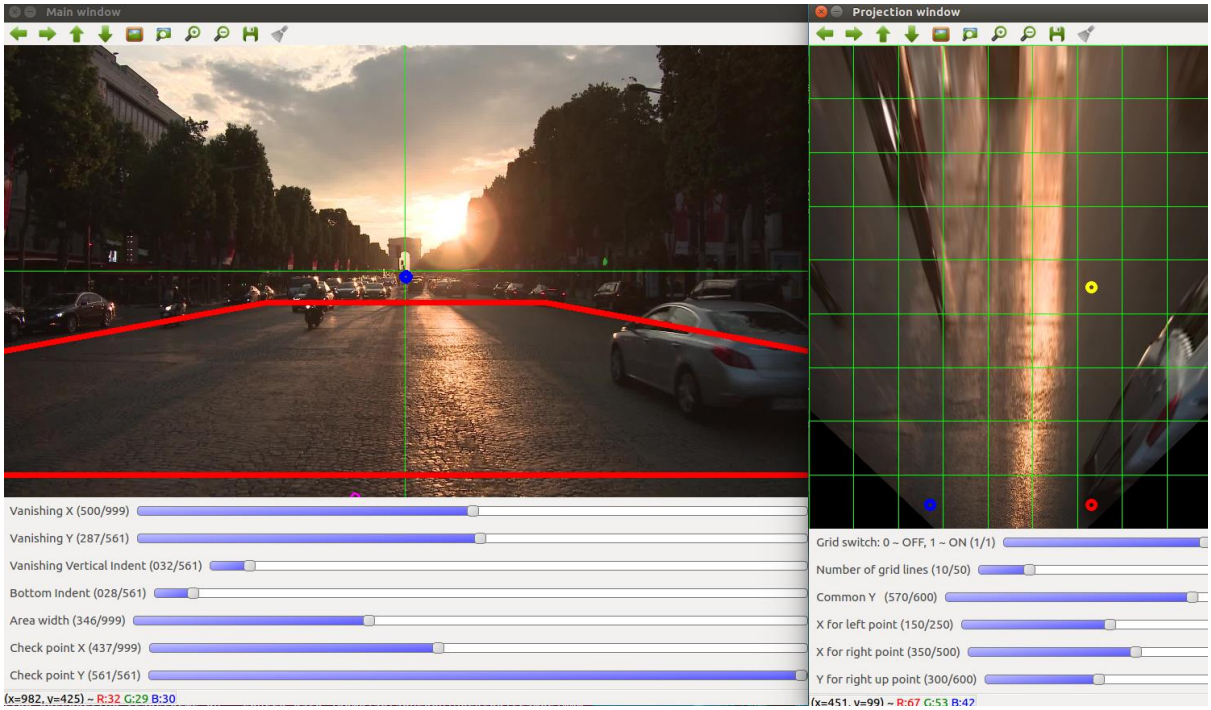

Рисунок 6. Настройка матрицы перспективной проекции.

Для того, чтобы с помощью данного преобразования определять расстояние до объектов, необходимо определить два коэффициента перевода пикселей в метры (k_x, k_y). Сделать это можно, если знать расстояние (в метрах) между объектами, которые располагаются в соответствии с тремя точками (см. правое изображение, синяя, красная, желтая точки).

После этого можно задавать зону (в соответствии с каким-либо нормативным документом для данной системы). Демонстрация этого приведена на рисунке ниже. Желтая зона означает область, в которой (условно) необходимо принимать решение о возможном применении торможения.

Рисунок 7. Демонстрация работы зоны предупреждения.

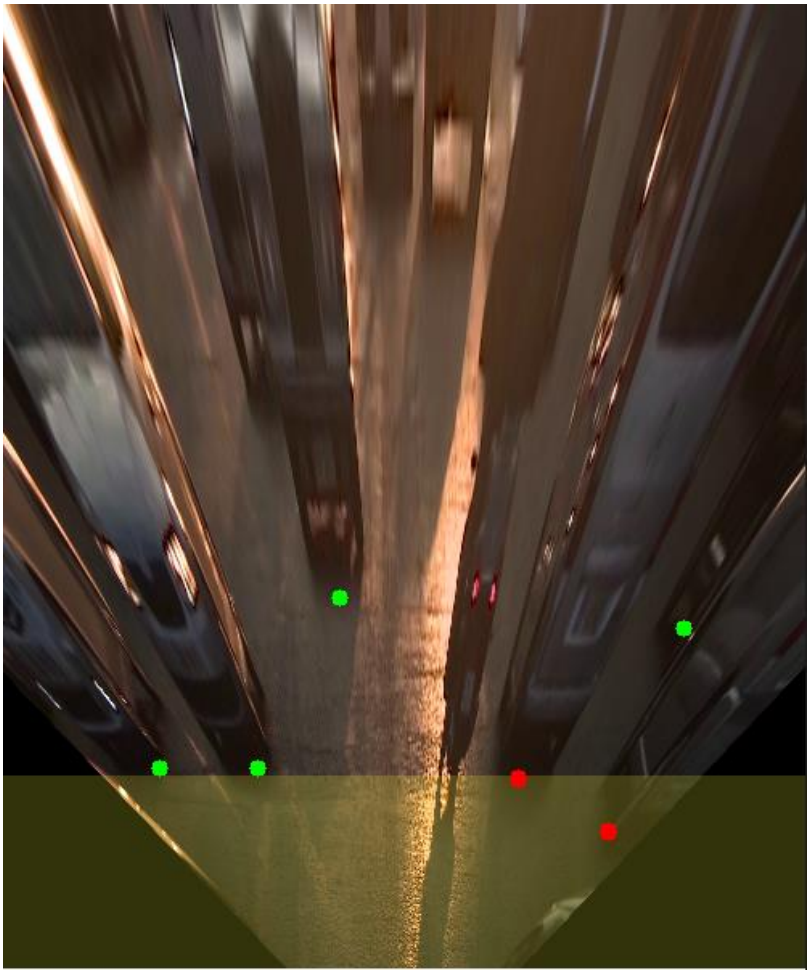

Рисунок 8. Зона предупреждения. Красные точки - объект находится в зоне, зеленые точки - объект вне зоны.

Ускорение инференса модели позволяет применять ее на embedded устройствах, например, nvidia jetson agx xavier. Это означает, что для того, чтобы сделать реальный прототип данной системы, не нужно располагать на конечном ТС высокопроизводительным, энергоемким и дорогим “железом”. А это в свою очередь означает удешевление стоимости конечной системы, что может повысить к ней интерес у потребителей на рынке.

6. Заключение.

В данной работе был изучен IE TensorRT для оптимизации инференса моделей машинного обучения для платформ, ориентированных на NVidia GPUs.

Изучение этого инструмента было рассмотрено на задаче детектирования объектов с помощью модели машинного обучения SSD.

Были проведены эксперименты по оценке качества и производительности оптимизированных моделей.

После чего полученные результаты были использованы в демонстрационном приложении, которое заключалось в написании условного прототипа возможной реализации части системы экстренного торможения.

Ссылки.

1. From Research To Production // URL: https://pytorch.org/
2. OpenCV // URL: https://opencv.org/
3. NVIDIA TensorRT Programmable Inference Accelerator // URL: https://developer.nvidia.com/tensorrt
4. torch2trt is a PyTorch to TensorRT converter which utilizes the TensorRT Python API // URL: https://github.com/NVIDIA-AI-IOT/torch2trt
5. torch2trt documentation // URL: https://nvidia-ai-iot.github.io/torch2trt/master/index.html
6. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y., & Berg, A. (2016). SSD: Single Shot MultiBox Detector*Lecture Notes in Computer Science*, 21–37.
7. Single Shot MultiBox Detector model for object detection // URL: https://pytorch.org/hub/nvidia_deeplearningexamples_ssd/
8. NVIDIA DeepLearningExamples // URL: https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Detection/SSD
9. PyTorch torchvision COCO Dataset // URL: https://medium.com/howtoai/pytorch-torchvision-coco-dataset-b7f5e8cad82

Приложение A (NVidia).

Для работы необходимо установить следующие зависимости:

- CUDA version: 10.2
- Nvidia Driver Version: 440
- cuDNN version: 7.6.5

1) Установить CUDA и Nvidia Driver можно с помощью .deb пакета, для этого необходимо следовать следующим инструкциям:

https://developer.nvidia.com/cuda-10.2-download-archive?target_os=Linux&target_arch=x86_64&target_distro=Ubuntu&target_version=1604&target_type=deblocal

2) Устанавить cuDNN через .tar архив, для этого необходимо перейти по ссылке:

https://developer.nvidia.com/rdp/cudnn-archive

Далее выбрать нужную версию (*Download cuDNN v7.6.5 (November 18th, 2019), for CUDA 10.2*), после чего скачать .tar архив (*cuDNN Library for Linux (x86)*)

Далее для установки следовать инструкциям:

(*https://docs.nvidia.com/deeplearning/cudnn/install-guide/index.html*, раздел tar file installation)

tar -xzvf cudnn-x.x-linux-x64-v8.x.x.x.tgz

sudo cp cuda/include/cudnn*.h /usr/local/cuda/include

sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64

sudo chmod a+r /usr/local/cuda/include/cudnn*.h /usr/local/cuda/lib64/libcudnn*

Приложение B (Python)

Для работы использовалась Anaconda (инструкции будут написаны для нее).

1) Сначала необходимо создать новую среду с python 3.7:

conda create -n python37-cuda102 python=3.7 anaconda

conda activate python37-cuda102

2) Необходимо установить в данную среду pip:

conda install -n python37-cuda102 -c anaconda pip

Для проверки выполнить: *pip --version*

В терминале должна быть указана версия и путь, который содержит активную анаконда среду.

3) Необходимо установить PyCuda:

pip install 'pycuda>=2019.1.1'

4) Необходимо установить ONNX parser:
pip install onnx==1.6.0

5) Далее установить PyTorch версии >= 1.5.0 под нужную версию CUDA.

Приложение С (TensorRT)

После того, как все шаги выше были проделаны, можно устанавливать TensorRT. Делать это необходимо через .tar файл, чтобы можно было установить его в созданную ранее анаконда среду.

В проекте использовалась версия 6.1.0.8.

Для установки необходимо перейти по ссылке: *https://developer.nvidia.com/nvidia-tensorrt-6x-download*. После чего скачать: *TensorRT 6.0.1.8 GA for Ubuntu 16.04 and CUDA 10.2 tar package.*

Далее следовать инструкциям из раздела Tar File Installation: *https://docs.nvidia.com/deeplearning/tensorrt/install-guide/index.html#installing-tar.*

1) Необходимо распаковать архив в любую удобную директорию:
tar xzvf TensorRT-7.2.1.6.Ubuntu-16.04.x86_64-gnu.cuda-10.2.cudnn8.0.tar.gz

2) Экспортировать в терминал абсолютный путь: <директория, в которую распаковали архив>/ TensorRT-6.0.1.8/lib

3) cd TensorRT-${version}/python

4) Проверить, активна ли сейчас анаконда среда

5) pip install tensorrt-6.0.1.8-cp37-none-linux_x86_64.whl

6) cd TensorRT-${version}/uff

pip install uff-0.6.9-py2.py3-none-any.whl

7) cd TensorRT-${version}/graphsurgeon
pip install graphsurgeon-0.4.5-py2.py3-none-any.whl

8) cd TensorRT-${version}/onnx_graphsurgeon
pip install onnx_graphsurgeon-0.2.6-py2.py3-none-any.whl

После того, как все пункты проделаны, ввести в терминале следуюущую команду: *python -c "import tensorrt as trt; print(trt.__version__)"*.

Если все прошло хорошо, то должна появиться версия TensorRT.

Приложение D (torch2trt)

После того, как все шаги выше были проделаны, установить конвертер в анаконда среду:

1) Клонировать реп: *https://github.com/NVIDIA-AI-IOT/torch2trt*

2) В файле setup.py заменить следующие строки:
2.1) В функции def trt_inc_dir():
return "/usr/include/aarch64-linux-gnu"
---->
return "<директория, в которую распаковали TensorRT>/TensorRT-6.0.1.8/include"
2.2) В функции def trt_lib_dir():
return "/usr/lib/aarch64-linux-gnu"
---->
return " <директория, в которую распаковали TensorRT> /TensorRT-6.0.1.8/lib"

3) python setup.py install --plugins