



лекция

GPU

Graphics processing unit

спикер



Игорь
Калгин



enot.ai

Содержание

Введение →

Содержание

Введение → Замеры latency →

Содержание

Введение → Замеры latency → **CUDA graph** →

Содержание

Введение → Замеры latency → CUDA
graph → **TensorRT** →

Содержание

Введение → Замеры latency → CUDA
graph → TensorRT → **Квантование**
в TensorRT →

Содержание

Введение → Замеры latency → CUDA
graph → TensorRT → Квантование
в TensorRT → **TensorRT LLM** →

Введение

Как устроен GPU и чем он отличается от CPU?

CPU

- спроектирован для последовательного выполнения команд
- мало ядер, но они более эффективны
- каждое ядро выполняет независимый набор команд

GPU

- спроектирован для параллельных вычислений
- много ядер, отдельные ядра менее эффективны чем у CPU
- множество ядер параллельно выполняют один и тот же набор команд

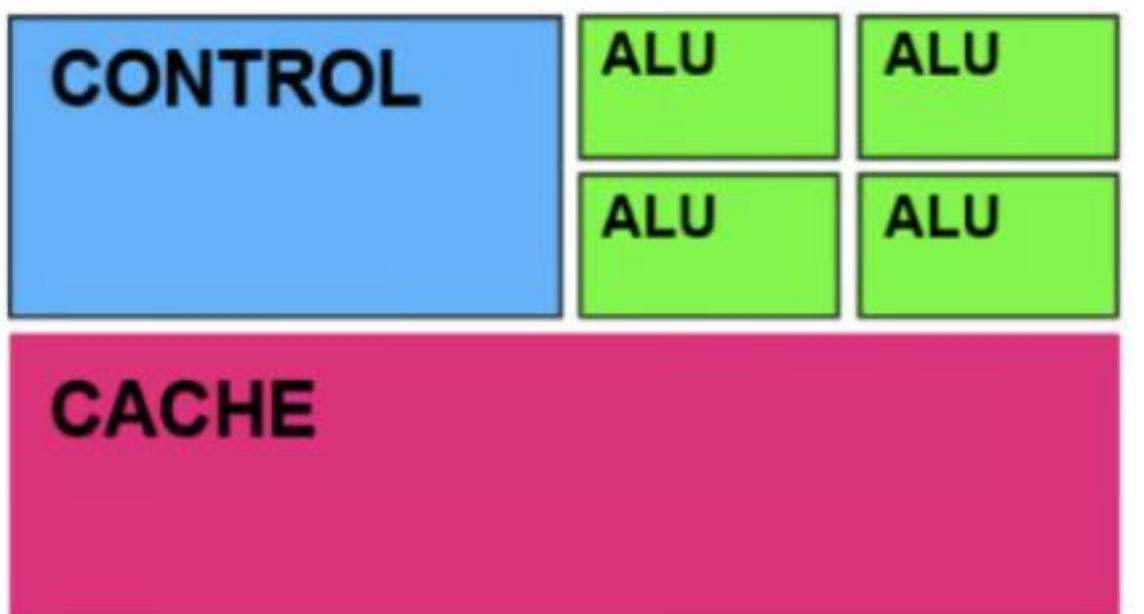
CPU

- большой набор инструкций для решения любых задач
- работа с потоками происходит на уровне OS
Много потоков, много накладных расходов
- SIMD — инструкции как основной инструмент ускорения алгоритмов
Single instruction, multiple data

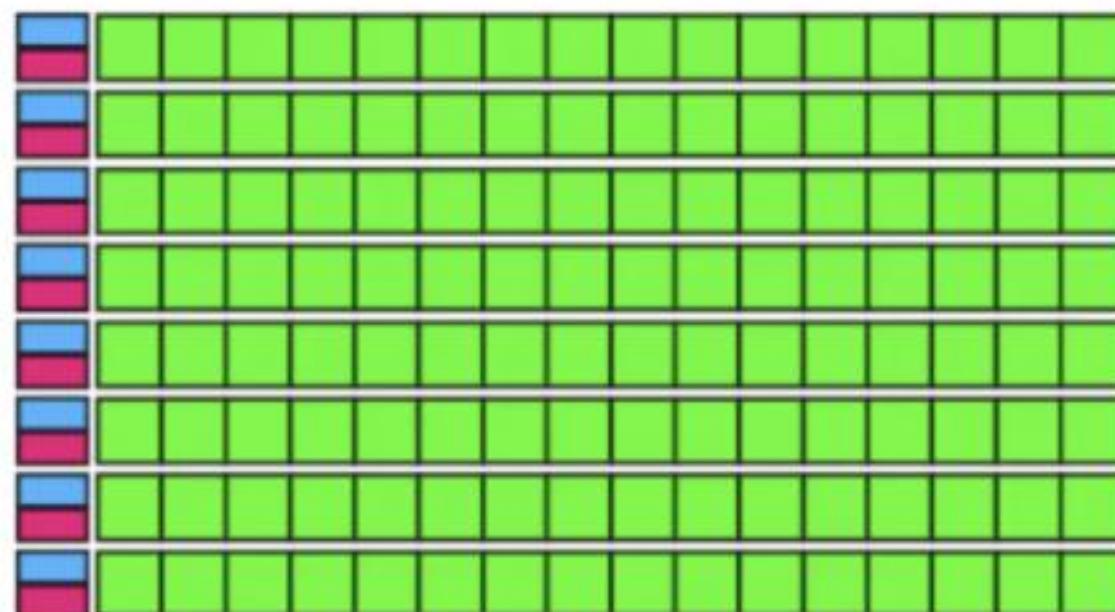
GPU

- малый набор инструкций для выполнения специализированных задач
- работа с потоками реализована на уровне железа
Накладные расходы минимальны
- SIMD — основной инструмент ускорения алгоритмов
Single instruction, multiple threads

CPU



GPU



Особенности GPU памяти

CPU DRAM/cache

- в приоритете время доступа
Latency
- сложная логика синхронизации
кэша разного уровня

GPU VRAM/cache

- в приоритете пропускная способность
Bandwidth
- синхронизация кэша происходит
по явному запросу

Замеры latency

CUDA libs

1

базовые математические библиотеки (cuBLAS, cuFFT, cuSPARSE, cuRAND). Практически никогда не используются пользователем напрямую

2

cuDNN (cuda Deep Neural Network library) — имплементация базовых операций deep neural network (matmul, conv, attention ...). Основа таких фреймворков, как PyTorch и TensorFlow

3

TensorRT — инференс оптимизатор и runtime. Как правило показывает лучшие результаты среди инференс движков

CUDA stream

Что такое CUDA stream?

- CUDA stream — это линейная последовательность операций принадлежащая определенному device (GPU)

Зачем используется CUDA stream?

- CUDA stream служит очередью в которую мы можем асинхронно (не дожидаясь фактического исполнения) добавлять операции для запуска на GPU
- Несколько разных CUDA stream могут использоваться для параллельного исполнения задач на одном device (GPU)

Вы можете представлять себе CUDA stream, как аналог потоков OS, только в контексте исполнения на GPU

Где используются CUDA stream?

→ Практически
везде :-)

В большинстве случаев вы просто не видите как тот
или иной фреймворк использует CUDA stream

Если пользователь не указал явно желаемый CUDA stream,
то используется default CUDA stream с номером 0, который
не надо создавать вручную, он существует всегда

Синхронизация

Так как операции на device (GPU) исполняются асинхронно относительно host (CPU), нам необходим механизм который позволит дождаться результата исполнения всех операции отправленных в очередь — механизм синхронизации device и host

**Когда происходит
синхронизация
в PyTorch?**

- При передаче данных с device на host (`Tensor.cpu()`,
`float(Tensor...)`)
- Явный вызов `torch.cuda.synchronize(device=None)`

Latency модели

Как правильно измерить время исполнения модели?

- Warmup — обязательный шаг перед настоящим замером
- Синхронизация после каждого запуска

Без синхронизации значение latency намного ниже, так как мы измерили не время исполнения, а время требуемое на отправку команд в CUDA stream

```
model = resnet18().cuda()
test_input = torch.rand(1, 3, 1920, 1080).cuda()

# warmup
for _ in range(10):
    model(test_input)

torch.cuda.synchronize()

# no sync
stats_no_sync = []
for _ in range(10):
    t_0 = perf_counter()
    model(test_input)
    t_1 = perf_counter()
    stats_no_sync.append(t_1 - t_0)

stats_no_sync = np.asarray(stats_no_sync) * 1000

torch.cuda.synchronize()

# sync
stats_sync = []
for _ in range(10):
    t_0 = perf_counter()
    model(test_input)
    torch.cuda.synchronize()
    t_1 = perf_counter()
    stats_sync.append(t_1 - t_0)

stats_sync = np.asarray(stats_sync) * 1000

print(f'{stats_no_sync.mean():.2f}ms +- {stats_no_sync.std():.3f}')
print(f'{stats_sync.mean():.2f}ms +- {stats_sync.std():.3f}')

1.59ms +- 0.045
19.75ms +- 0.099
```

Timeit

→ В то же время timeit дал правильный ответ, как с синхронизацией, так и без. Почему?

```
[2]: %timeit model(test_input)
20.4 ms ± 29.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

[3]: %timeit model(test_input);torch.cuda.synchronize()
20.3 ms ± 41.3 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Происходит неявная синхронизация. Очередь команд в CUDA stream конечна. Если заполнить очередь, то это работает как synchronize(), host ждет пока очередь освободиться

Не стоит надеяться на подобное в своих замерах :-).

Timeit

Правильный результат,
но чтобы его получить
потребовалось 10 минут
вместо 6 секунд!!!

Timeit увеличивает
количество повторений
пока не получит приемле-
мое среднеквадратичное
отклонение

```
import time
import torch

test_data = torch.ones(8, 3, 24, 24, device="cuda")
test_model = torch.nn.Sequential(
    torch.nn.Conv2d(in_channels=3, out_channels=10000, kernel_size=3, device="cuda"),
    torch.nn.Conv2d(in_channels=10000, out_channels=1000, kernel_size=3, device="cuda"),
)

with torch.no_grad():
    t_0 = time.perf_counter()
    %timeit test_model(test_data);torch.cuda.synchronize()
    t_1 = time.perf_counter()
    print(f"{t_1 - t_0:.1f}s")

81 ms ± 326 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
6.6s

with torch.no_grad():
    t_0 = time.perf_counter()
    %timeit test_model(test_data)
    t_1 = time.perf_counter()
    print(f"{t_1 - t_0:.1f}s")

81.4 ms ± 63.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
643.5s
```

Главное

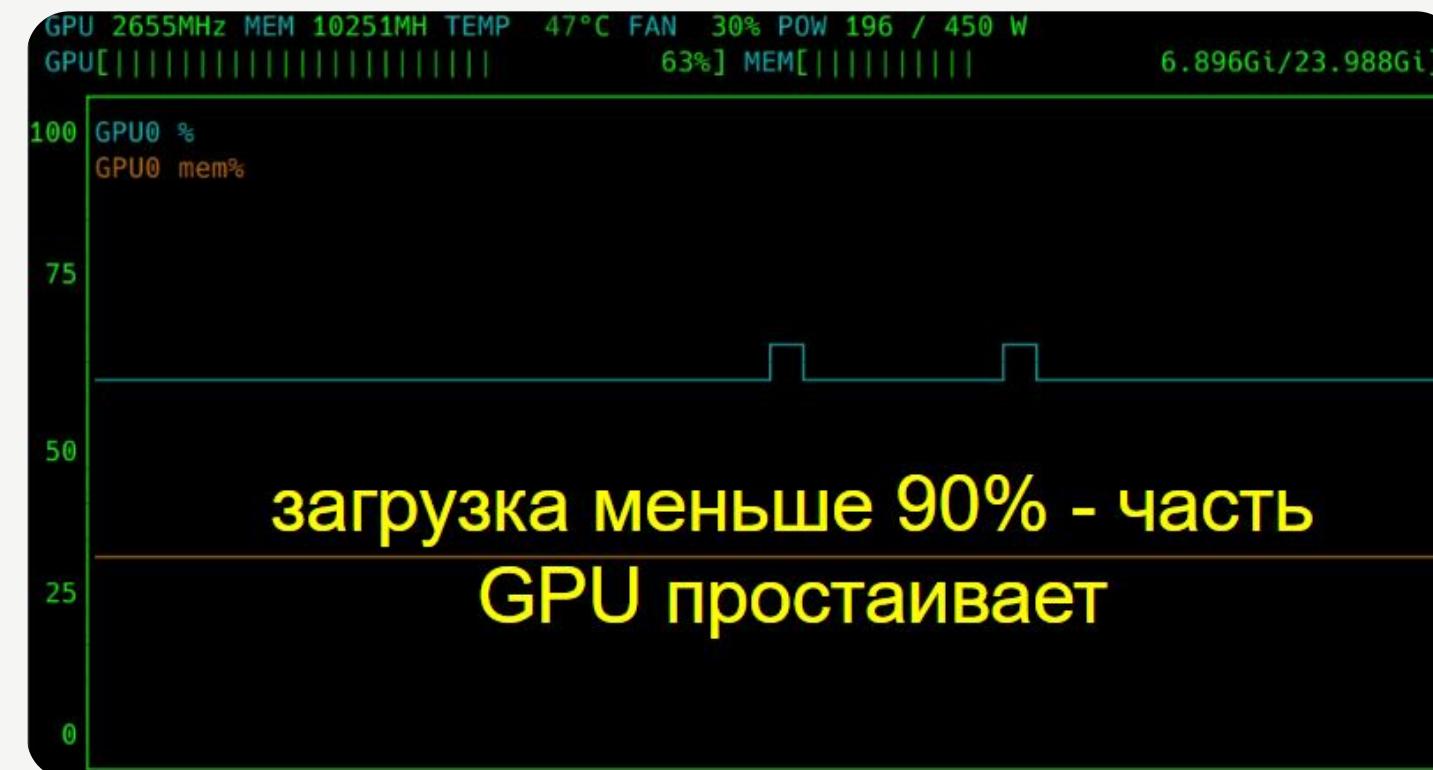
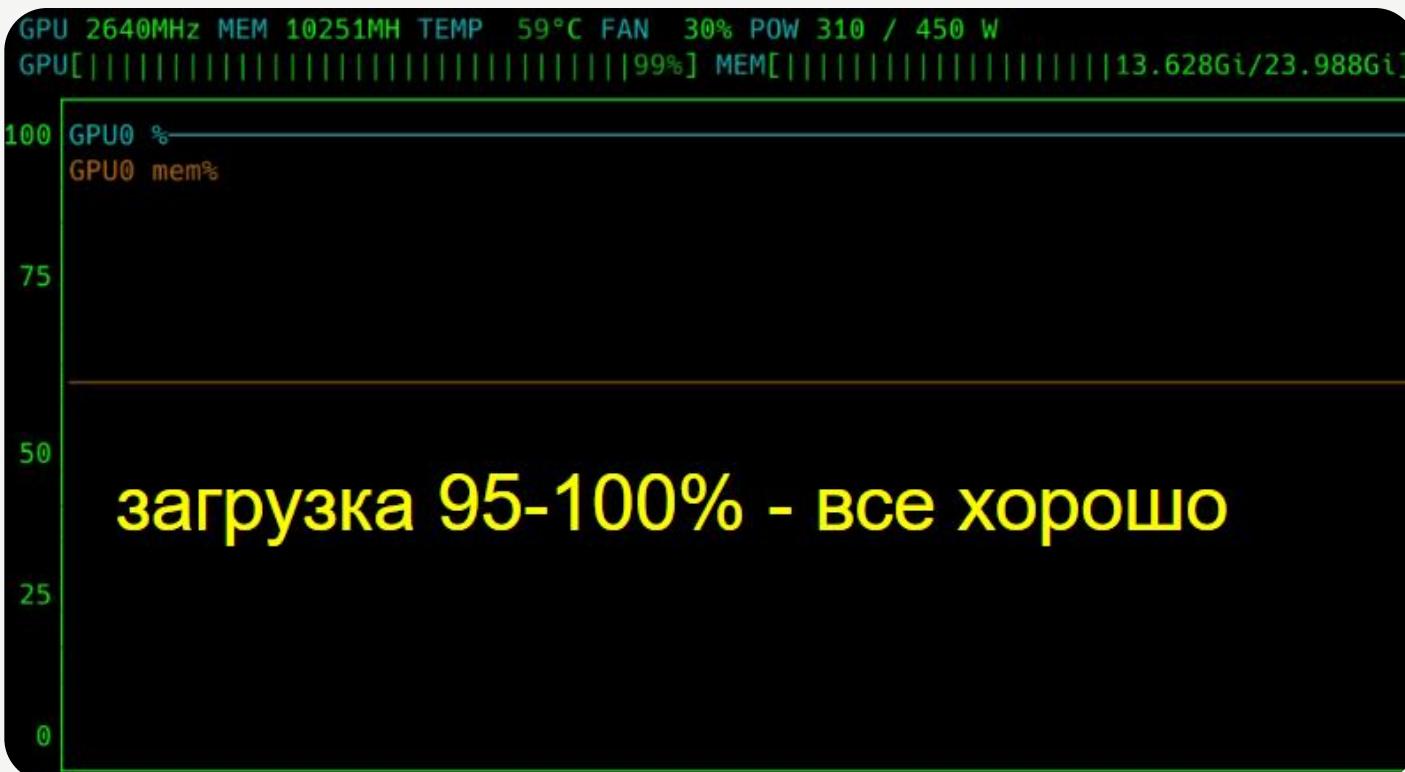
-
- 1 перед началом замеров всегда делайте warmup запуск
(5-10 итераций)

 - 2 после каждого запуска вызывайте
`torch.cuda.synchronize()`

CUDA graph

Следите за загрузкой GPU

- Уровень загрузки GPU самый простой и доступный инструмент анализа модели



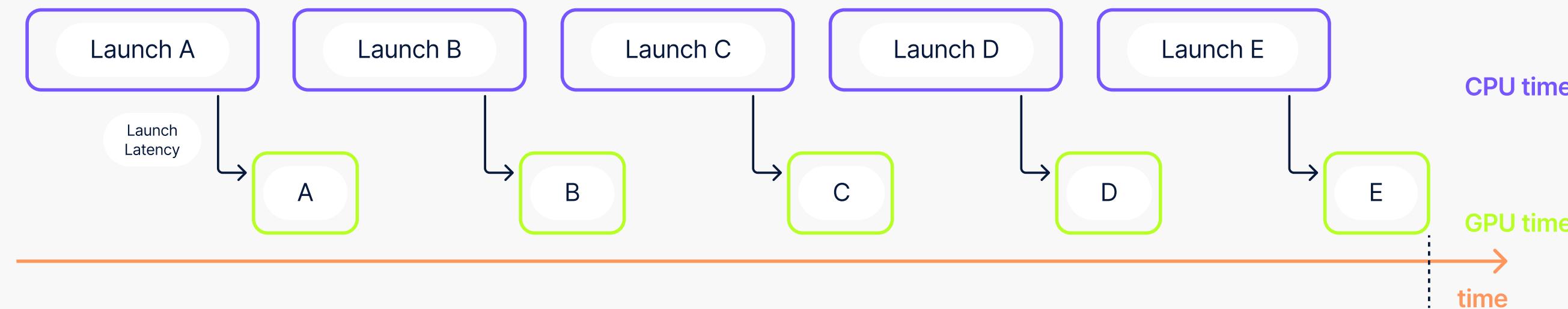
Простой способ увеличить загрузку GPU

Как правило достаточно поднять размер batch

Но почему это работает?
И что делать если нет возможности поменять размер batch

Задачи для GPU формируют на CPU

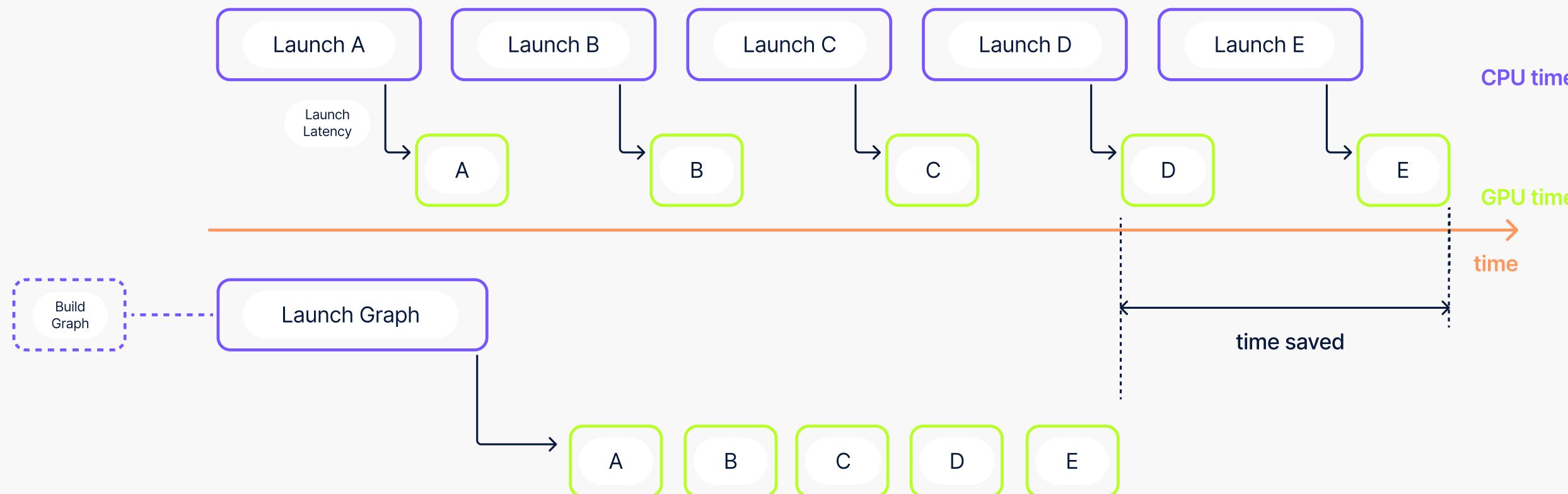
→ Если время составления и отправки задачи для GPU превышает время её исполнения, то GPU пристаивает в ожидании новой задачи



Увеличивая batch size в N раз вы также увеличиваете время исполнения задач в N раз. Очередь задач растет быстрее чем задачи выполняются на GPU

CUDA graph

→ CUDA graph — инструмент позволяющий записывать последовательность команд для GPU и отправлять всю записанную последовательность как один объект. Таким образом мы избавляемся от накладных расходов на формирование очереди и связанных с этим простоев GPU



Разбираем пример CUDA graph

```
import torch
from torchvision.models import mobilenet_v2

model = mobilenet_v2().cuda().eval()
test_data = torch.ones(1, 3, 128, 128, device="cuda")

with torch.no_grad():
    # Warmup
    for _ in range(10):
        model(test_data)

    torch.cuda.synchronize()

    # Capture
    graph = torch.cuda.CUDAGraph()
    input_placeholder = test_data
    with torch.cuda.graph(graph):
        output_placeholder = model(input_placeholder)

    torch.cuda.synchronize()

    # Fill the graph's input memory with new data
    new_data = torch.zeros_like(input_placeholder)
    input_placeholder.copy_(new_data)

    # Replay
    graph.replay()
    output_data = output_placeholder.clone()

%timeit model(test_data);torch.cuda.synchronize()
3.25 ms ± 1.62 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit graph.replay();torch.cuda.synchronize()
1.11 ms ± 221 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Ограничения CUDA graph

- 1 CUDA graph работает со статическими значениями shape входа/выхода/промежуточных тензоров
- 2 CUDA graph записывает только последовательность команд для GPU, если часть операций происходит на CPU, то вы не сможете их записать
- 3 CUDA graph всегда проигрывает одну и ту же последовательность команд

Если ваша модель имеет динамический граф исполнения, то можно разбить её на части. Части с постоянным shape и статическим графиком можно оптимизировать с помощью CUDA graph

Главное

-
- 1 Следите за загрузкой GPU
 - 2 Простейший способ увеличить утилизацию GPU — это поднять размер batch
 - 3 Если нет возможности увеличить batch, то можно воспользоваться CUDA graph
-

GPU

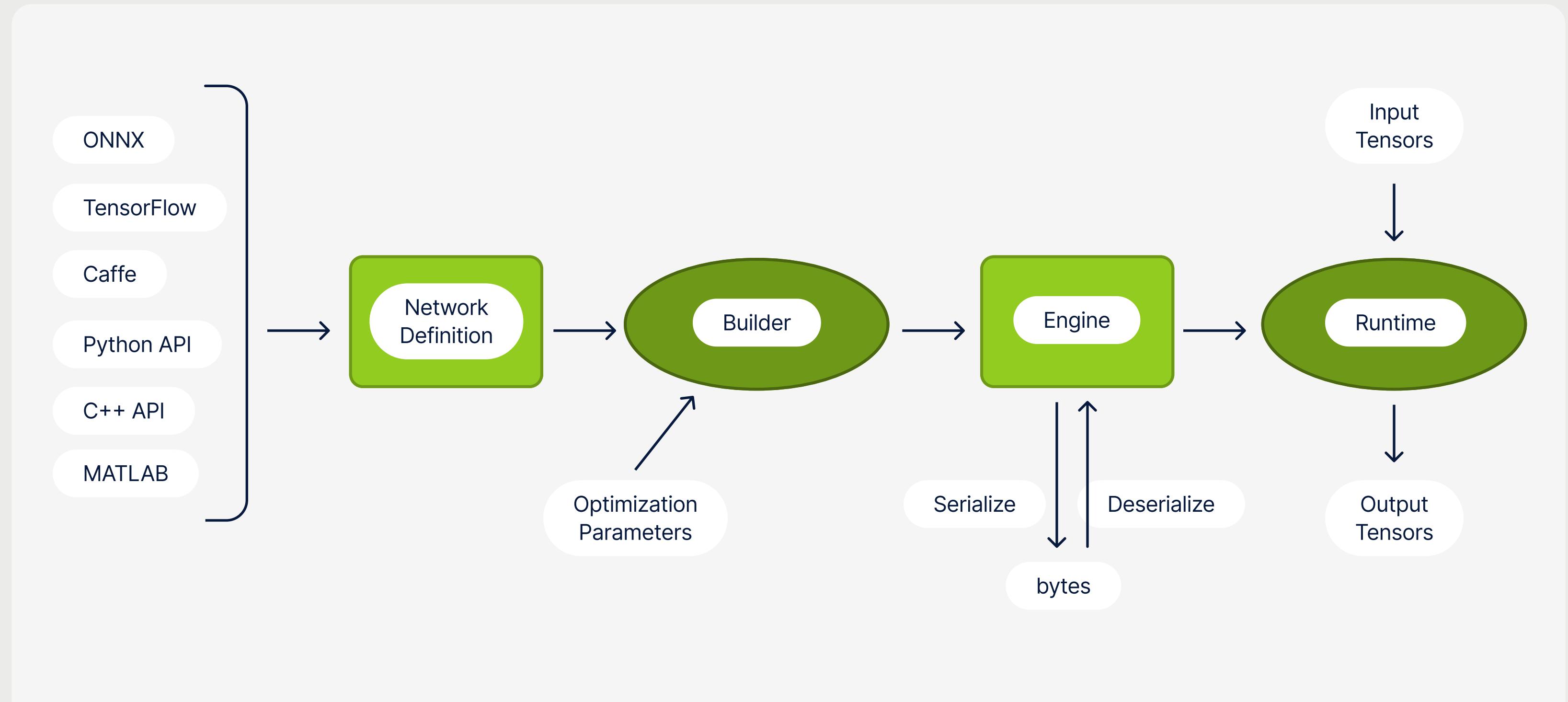
30

TensorRT

Что такое TensorRT?

- Это высокопроизводительный инференс фреймворк, который включает в себя оптимизатор инференса и рантайм для запуска оптимизированной модели
- TensorRT не может участвовать в обучении модели, он предназначен для запуска обученных моделей

TensorRT Workflow



Конвертация модели

В качестве входа Builder принимает описание модели в виде класса NetworkDefinition.

На данный момент TensorRT предоставляет возможность автоматической конвертации ONNX в NetworkDefinition. Любой другой формат потребует промежуточной конвертации в ONNX, либо вам придется вручную через API составить объект NetworkDefinition

PyTorch

→ Хорошая поддержка экспорта моделей в ONNX

TensorFlow

→ Потребуется промежуточная конвертация с помощью tf2onnx

Подготовка ONNX

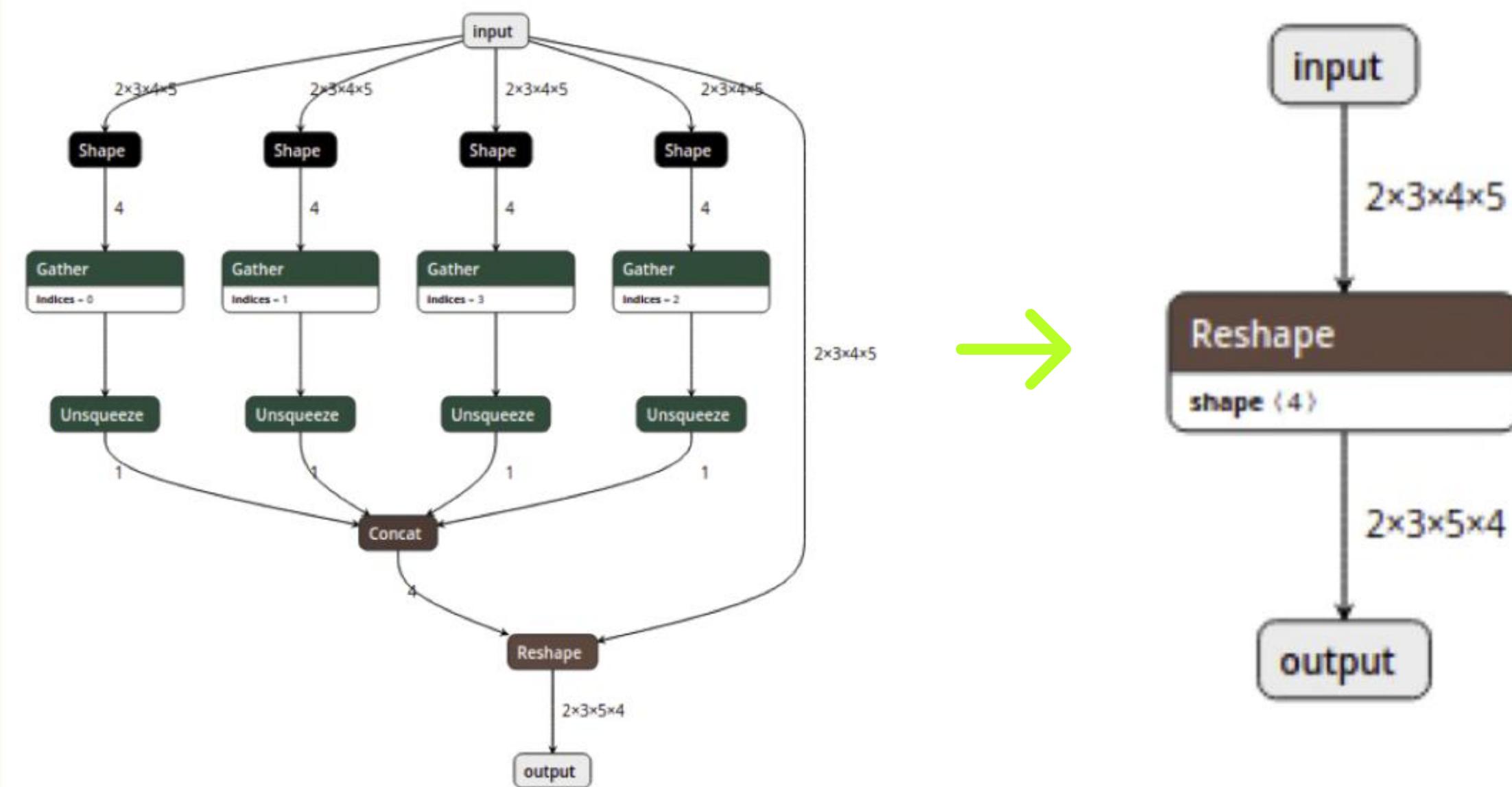
Даже если у вас есть готовый валидный ONNX — это не значит, что он без проблем будет работать с TensorRT

Чтобы минимизировать проблемы на этапе конвертации пользуйтесь onnx-graphsurgeon, или onnx-simplifier

```
$ onnxsim my-model.onnx my-model-opt.onnx
Simplifying...
Finish! Here is the difference:
```

	Original Model	Simplified Model
Add	142	42
Clip	1	1
Concat	82	42
Constant	693	184
Conv	117	117
Div	70	50
Gather	80	0
LeakyRelu	47	47
Mul	50	50
Relu	2	2
Reshape	240	40
Shape	80	0
Slice	376	319
Sub	50	0
Transpose	20	20
Unsqueeze	180	0
Upsample	2	2
Model Size	3.4MiB	2.7MiB

Подготовка ONNX



Подготовка ONNX

Список поддерживаемых операций и их ограничения
можно найти в репозитории TensorRT на github

github.com/onnx/onnx-tensorrt/blob/main/docs/operators.md ↗

Параметры оптимизации модели

1 Перед компиляцией нужно задать
желаемые параметры оптимизации

- Допустимые типы данных тензоров
tf32, fp16, int8
- Границы динамических осей
и размер для оптимизации
min, max, opt

2 Почему вообще надо задавать границы
динамических осей?

- TensorRT должен знать максимальный
допустимый размер тензоров, так как
это влияет на выбор оптимального
алгоритма конкретной операции
экономить ли память в ущерб производительности?
- Оптимальный размер — это длина
оси которая будет использоваться
для оптимизации

Какие параметры выбрать?

- ✓ Чем меньше динамических осей тем лучше оптимизируется модель, как правило лучшая производительность если все оси фиксированы. Исключение из правил batch size. Динамический batch size практически не влияет на latency
- ✓ Если fp16 не достаточно, то придется заняться квантованием
- ✓ Простейший способ удвоить производительность — перейти от fp32 к fp16
- ✓ Если все таки динамические оси необходимы, то в качестве оптимального значения указываем размер который будет чаще всего использоваться

Как оптимизируется модель?

Последовательные операции
объединяются в одну где это возможно

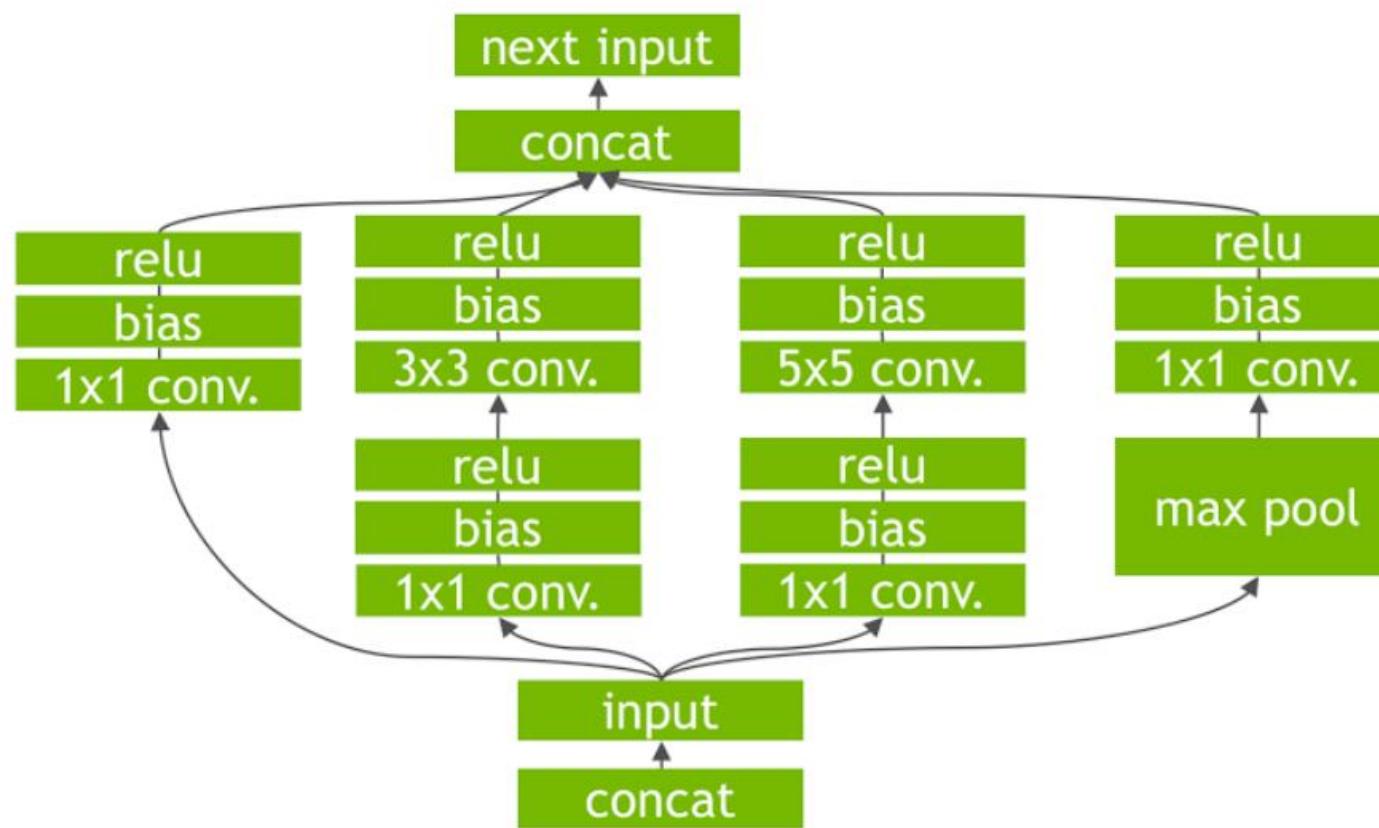
Например → convolution + batch norm,
можно заменить на один
convolution

Практически на каждую операцию в TensorRT доступно несколько вариантов CUDA ядер с альтернативными реализациями. Реализации отличаются потреблением памяти и доступными вариантами формата входных/выходных данных (NCHW, NHWC...), а главное доступными вариантами типов данных (fp32, tf32, fp16, int8, int4)

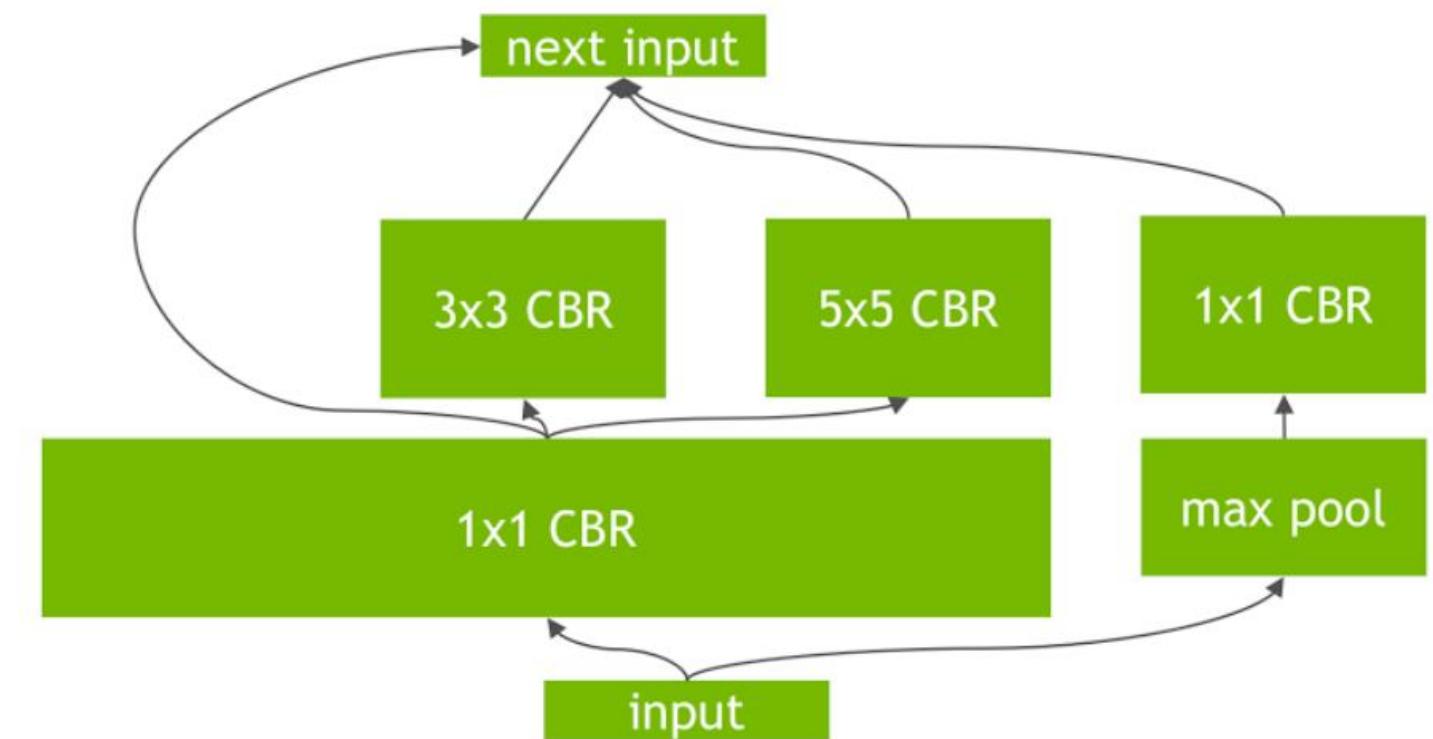
Объединение операций

Layer fusion

Un-Optimized Network



TensorRT Optimized Network

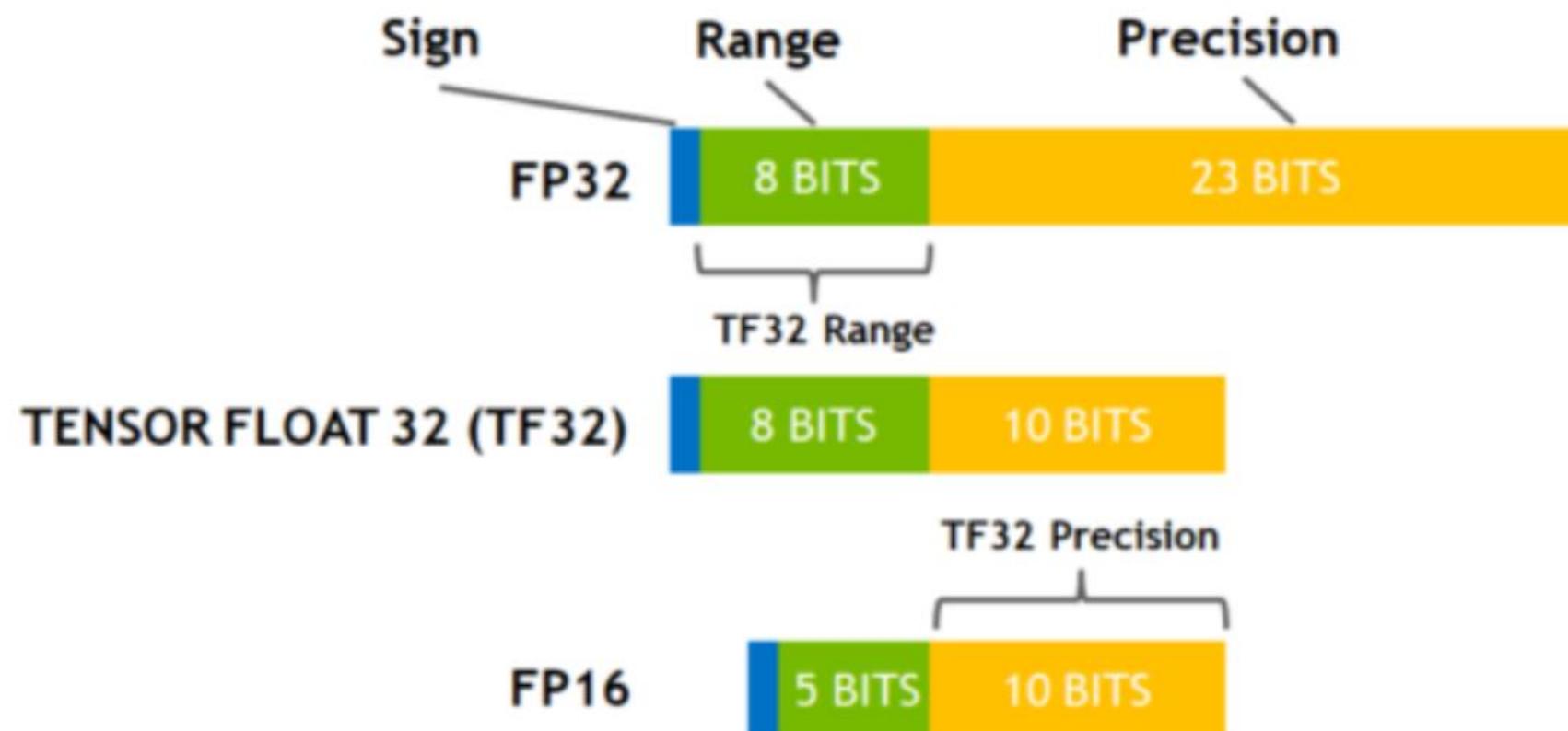


Как оптимизируется модель?

-
- 1 TensorRT перебирает варианты доступных CUDA ядер, входных/выходных форматов и тип данных для вычислений в поисках комбинации с минимальным latency
 - 2 В условиях mixed precision не всегда выгоден самый быстрый алгоритм, так как может потребоваться конвертация входных/выходных данных
 - 3 Объем доступной памяти, также влияет на конечный результат. Как правило самый быстрый алгоритм потребляет наибольшее кол-во памяти
-

fp16 и tf32

- tf32 практически никогда не портит точность
- fp16 приводит к проблемам, если у вас есть LayerNormalization, или достаточно большие матричные умножения



Что делать с просадкой точности fp16?

1

Вернитесь к fp32 и убедитесь что проблема в fp16

2

Если у вас есть динамические оси передвигните opt
значение на минимум/максимум

3

Если сеть обучалась в fp32, то стоит найти вариант
который учили в fp16

trtexec

→ утилита командной строки
для тестирования и измерения latency/throughput

Позволит быстро без кода проверить разные варианты оптимизации,
но не предоставляет интерфейса для проверки точности модели

Пример компиляции с trtexec

```
$ trtexec --fp16 --onnx=my-model-opt.onnx --saveEngine=my-engine.engine --buildOnly --minShapes=x:1x3x224x224 --maxShapes=x:32x3x224x224 --optShapes=x:32x3x224x224

&&& RUNNING TensorRT.trtexec [TensorRT v8400] # /home/igor/Distr/TensorRT-8.4.0.6/bin/trtexec --fp16 --onnx=my-model-opt.onnx --saveEngine=my-engine.engine --buildOnly --minShapes=x:1x3x224x224 --maxShapes=x:32x3x224x224 --optShapes=x:32x3x224x224
[04/18/2024-16:19:02] [I] === Model Options ===
[04/18/2024-16:19:02] [I] Format: ONNX
[04/18/2024-16:19:02] [I] Model: my-model-opt.onnx
[04/18/2024-16:19:02] [I] Output:
[04/18/2024-16:19:02] [I] === Build Options ===
[04/18/2024-16:19:02] [I] Max batch: explicit batch
[04/18/2024-16:19:02] [I] Memory Pools: workspace: default, dlaSRAM: default, dlaLocalDRAM: default, dlaGlobalDRAM: default
[04/18/2024-16:19:02] [I] minTiming: 1
[04/18/2024-16:19:02] [I] avgTiming: 8
[04/18/2024-16:19:02] [I] Precision: FP32+FP16
```

Замеры latency/throughput с trtexec

```
$ trtexec --loadEngine=my-engine.engine --shapes=x:1x3x224x224  
&&& RUNNING TensorRT.trtexec [TensorRT v8400] # /home/igor/Distr/TensorRT-8.4.0.6/bin/trtexec --loadEngine=my-engine.engine --shapes=x:1x3x224x224  
[04/18/2024-16:28:00] [I] === Model Options ===  
[04/18/2024-16:28:00] [I] Format: *  
[04/18/2024-16:28:00] [I] Model:  
[04/18/2024-16:28:00] [I] Output:
```

```
[I] === Performance summary ===  
[I] Throughput: 508.791 qps  
[I] Latency: min = 2.01904 ms, max = 2.40648 ms, mean = 2.06718 ms, median = 2.04236 ms, percentile(99%) = 2.40073 ms  
[I] End-to-End Host Latency: min = 3.47217 ms, max = 4.42708 ms, mean = 3.70519 ms, median = 3.64288 ms, percentile(99%) = 4.37744 ms  
[I] Enqueue Time: min = 0.161133 ms, max = 1.30855 ms, mean = 0.799936 ms, median = 0.785889 ms, percentile(99%) = 0.993027 ms  
[I] H2D Latency: min = 0.0795898 ms, max = 0.121582 ms, mean = 0.100333 ms, median = 0.101807 ms, percentile(99%) = 0.107422 ms  
[I] GPU Compute Time: min = 1.92505 ms, max = 2.29788 ms, mean = 1.96159 ms, median = 1.93542 ms, percentile(99%) = 2.29172 ms  
[I] D2H Latency: min = 0.00317383 ms, max = 0.0150146 ms, mean = 0.00525533 ms, median = 0.00488281 ms, percentile(99%) = 0.0100098 ms  
[I] Total Host Walltime: 3.00713 s  
[I] Total GPU Compute Time: 3.00123 s  
[W] * GPU compute time is unstable, with coefficient of variance = 3.89226%.  
[W] If not already in use, locking GPU clock frequency or adding --useSpinWait may improve the stability.  
[I] Explanations of the performance metrics are printed in the verbose logs.
```

Polygraphy

- фреймворк предоставляющий высокоуровневый интерфейс для запуска/дебагинга моделей на различных инференс фреймворках в том числе и TensorRT

Polygraphy упрощает работу с TensorRT давая возможность передавать данные напрямую из torch не задумываясь об особенностях выделения памяти. Так же вам доступны все настройки доступные в низкоуровневом API TensorRT

Пример с Polygraphy

```
from polygraphy.backend.trt import CreateConfig
from polygraphy.backend.trt import engine_from_network
from polygraphy.backend.trt import NetworkFromOnnxPath
from polygraphy.backend.trt import save_engine
from polygraphy.backend.trt import TrtRunner

model = NetworkFromOnnxPath("my-model.onnx")
config = CreateConfig(fp16=True)

engine = engine_from_network(model, config=config)
save_engine(engine, path="my-model.engine")

input_data = torch.rand(1, 3, 224, 224)
with TrtRunner(engine) as trt_runner:
    output = trt_runner.infer(feed_dict={"x": input_data})

[I] Configuring with profiles:
  Profile 0:
    {x [min=[1, 3, 224, 224], opt=[1, 3, 224, 224], max=[1, 3, 224, 224]]}
]
[I] Building engine with configuration:
Flags | [FP16]
Engine Capability | EngineCapability.DEFAULT
Memory Pools | [WORKSPACE: 11172.19 MiB, TACTIC_DRAM: 11172.19 MiB]
Tactic Sources | [CUBLAS, CUBLAS_LT, CUDNN, EDGE_MASK_CONVOLUTIONS, JIT_CONVOLUTIONS]
Profiling Verbosity | ProfilingVerbosity.DETAILED
Preview Features | [FASTER_DYNAMIC_SHAPES_0805, DISABLE_EXTERNAL_TACTIC_SOURCES_FOR_CORE_0805]
```

Главное

-
- 1 TensorRT требует подготовки ONNX (onnx-graphsurgeon, или onnx-simplifier)
 - 2 Чем меньше динамических осей, тем легче TensorRT оптимизировать модель
 - 3 fp16 — самый простой способ удвоить производительность
 - 4 Для быстрых замеров latency/throughput можно воспользоваться trtexec
 - 5 Polygraphy — поможет упростить запуск TensorRT
-

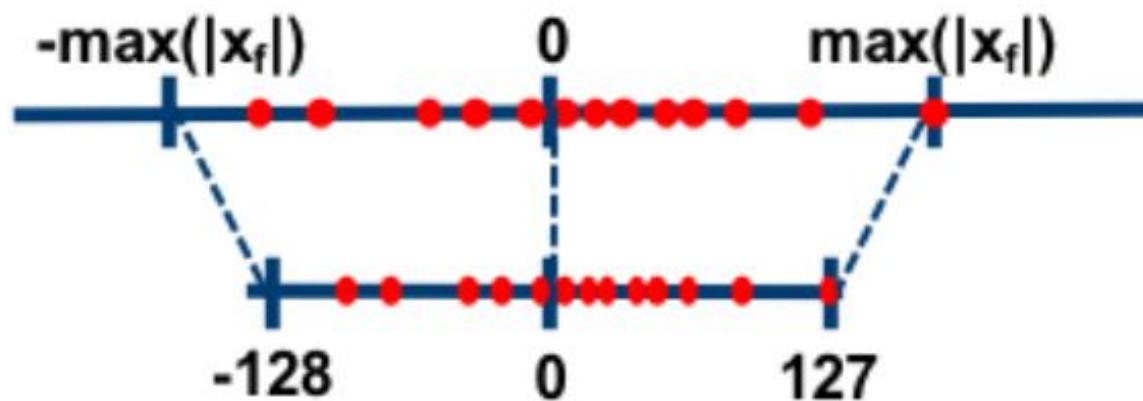
Квантование в TensorRT

Доступные типы для квантования

-
- | | |
|------|---|
| INT8 | → Post Training Quantization, Quantization Aware Training |
| INT4 | → только Quantization Aware Training и только веса |
| FP8 | → только Quantization Aware Training. Невозможно использовать с INT8 в одной модели. На данный момент доступно только на H100 |
-

INT8

→ Поддерживается только симметричное квантование, как для активаций так и для весов



Для активации применяется скалярное (per tensor) квантование (1 коэффициент на тензор).

Веса поддерживают как скалярное, так и векторное (per channel) квантование (независимый коэффициент на каждый канал). Также доступно блочное квантование

INT8

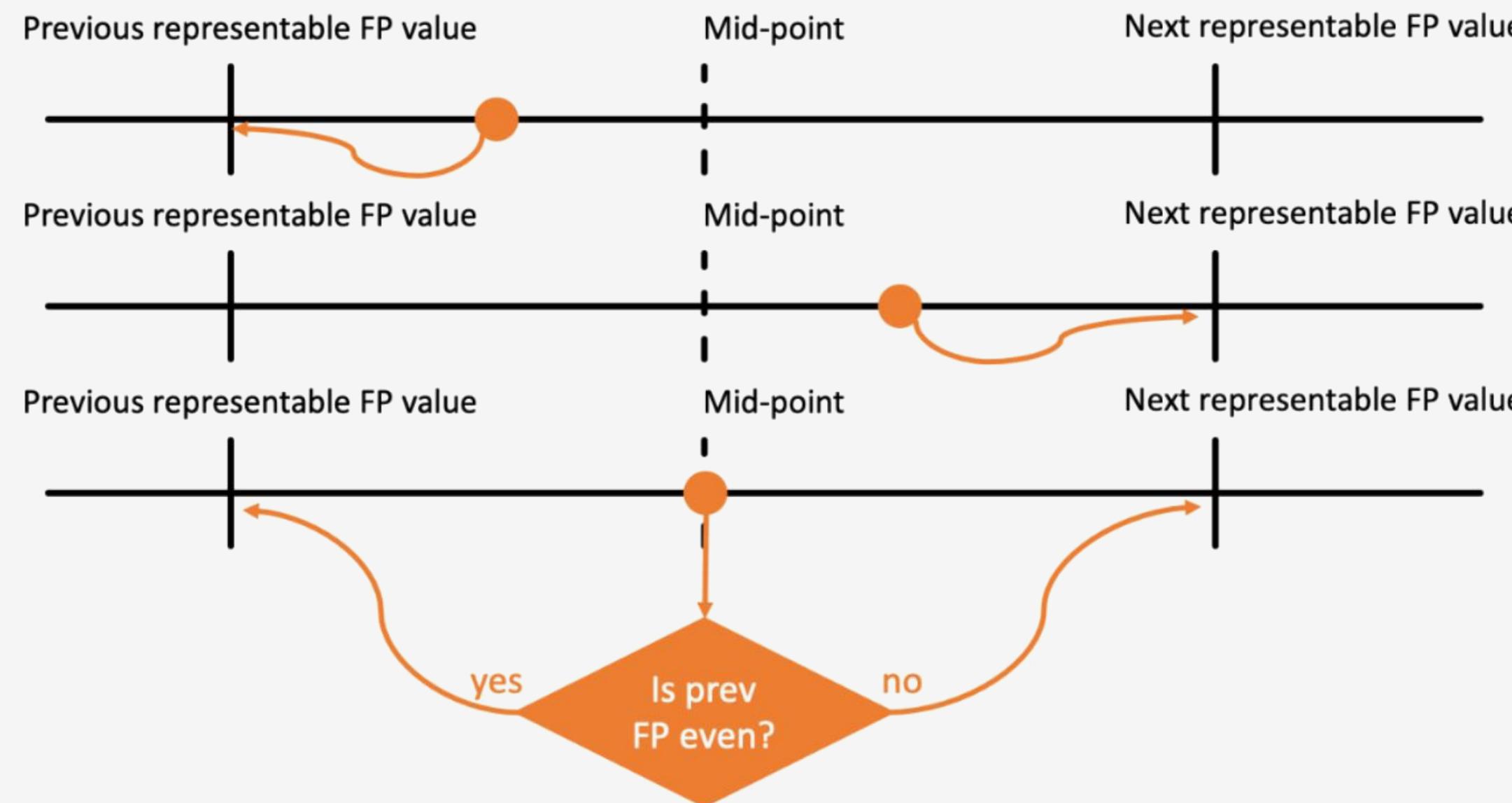
$$x_q = \text{roundWithTiesToEven}(\text{clip}(x/s, -128, 127))$$

$$x = x_q^* s$$

$$s = \max(\text{abs}(x_{\min}), \text{abs}(x_{\max}))/127$$

x — float point значение x_q — int8 квантованное значение [-128, 127] s — scale

roundWithTiesToEven



Post Training Quantization

Доступные методы калибровки

-
- | | |
|--------------------------------|--|
| IInt8MinMaxCalibrator | → вычисляет scale на основе минимального и максимального значения. Хорошо работает для NLP |
| IInt8LegacyCalibrator | → существует только для обратной совместимости |
| IInt8EntropyCalibrator | → первый вариант энтропийного калибратора. Позволяет автоматически избавиться от выбросов |
| IInt8EntropyCalibrator2 | → последняя версия энтропийного калибратора. Позволяет автоматически избавиться от выбросов. Рекомендуется для большинства случаев |
-

Для Imagenet сетей достаточно ~500 картинок

Пример с Polygraphy

```
import tensorrt as trt
from polygraphy.backend.trt import CreateConfig
from polygraphy.backend.trt import Calibrator
from polygraphy.backend.trt import engine_from_network
from polygraphy.backend.trt import NetworkFromOnnxPath
from polygraphy.backend.trt import save_engine
from polygraphy.backend.trt import TrtRunner

model = NetworkFromOnnxPath("my-model.onnx")
calibrator = Calibrator(
    data_loader={"x": torch.rand(1, 3, 224, 224)} for _ in range(50)),
    algo=trt.CalibrationAlgoType.ENTROPY_CALIBRATION_2,
)
config = CreateConfig(fp16=True, int8=True, calibrator=calibrator)

engine = engine_from_network(model, config=config)
save_engine(engine, path="my-model.engine")

input_data = torch.rand(1, 3, 224, 224)
with TrtRunner(engine) as trt_runner:
    output = trt_runner.infer(feed_dict={"x": input_data})

[I] Configuring with profiles:[  
    Profile 0:  
        {x [min=[1, 3, 224, 224], opt=[1, 3, 224, 224], max=[1, 3, 224, 224]]}  
    ]  
[I] Using calibration profile: {x [min=[1, 3, 224, 224], opt=[1, 3, 224, 224], max=[1, 3, 224, 224]]}  
[I] Building engine with configuration:  
    Flags | [FP16, INT8]  
    Engine Capability | EngineCapability.DEFAULT  
    Memory Pools | [WORKSPACE: 11172.19 MiB, TACTIC_DRAM: 11172.19 MiB]  
    Tactic Sources | [CUBLAS, CUBLAS_LT, CUDNN, EDGE_MASK_CONVOLUTIONS, JIT_CONVOLUTIONS]  
    Profiling Verbosity | ProfilingVerbosity.DETAILED  
    Preview Features | [FASTER_DYNAMIC_SHAPES_0805, DISABLE_EXTERNAL_TACTIC_SOURCES_FOR_CORE_0805]  
    Calibrator | Calibrator(<generator object <genexpr> at 0x7fb637a4b190>, BaseClass=<class  
CALIBRATION_2: 2>)
```

Quantization Aware Training

Что нужно помнить?

- 1 Fake quant необходимо поставить на вход и весах всех conv, linear

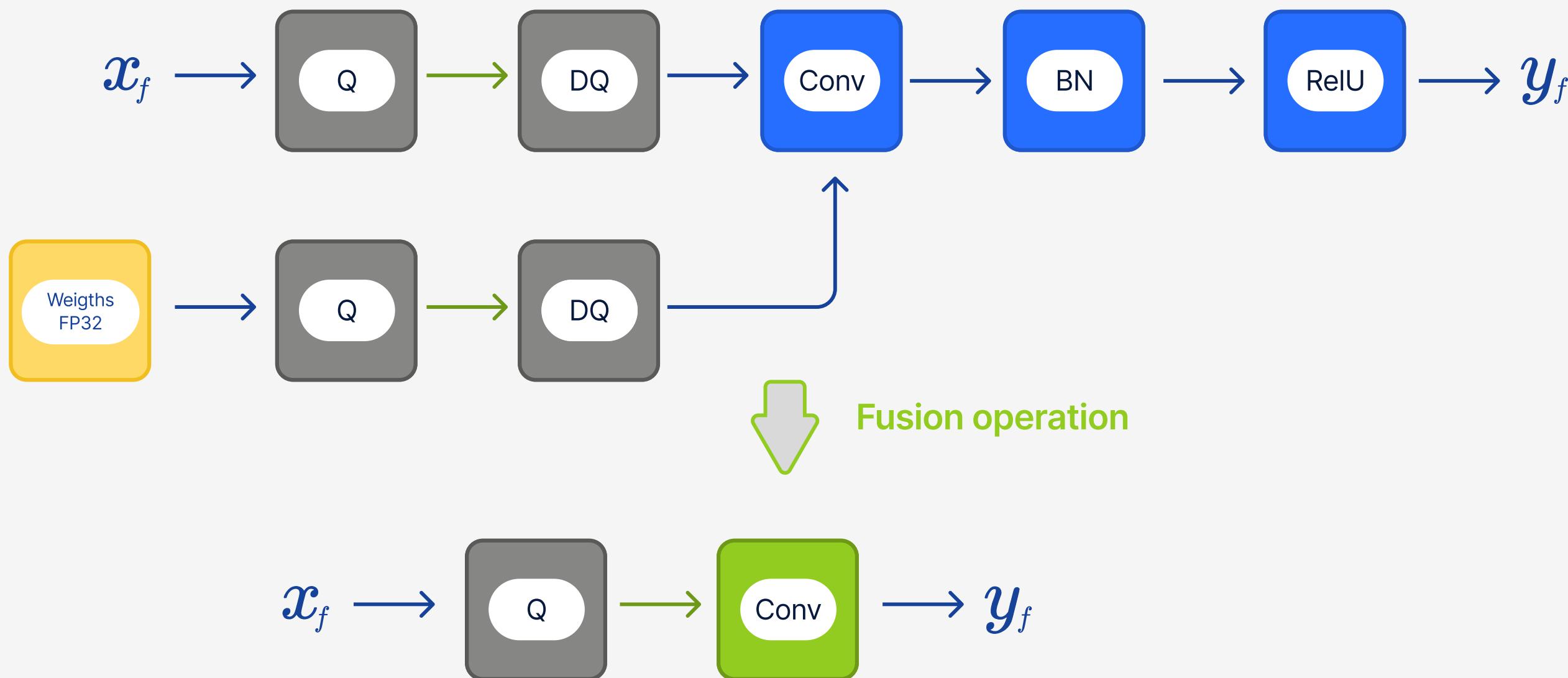
- 2 Последний слой классификатора обычно не квантуется
(большая просадка точности)

- 3 В TensorRT bias не квантуется

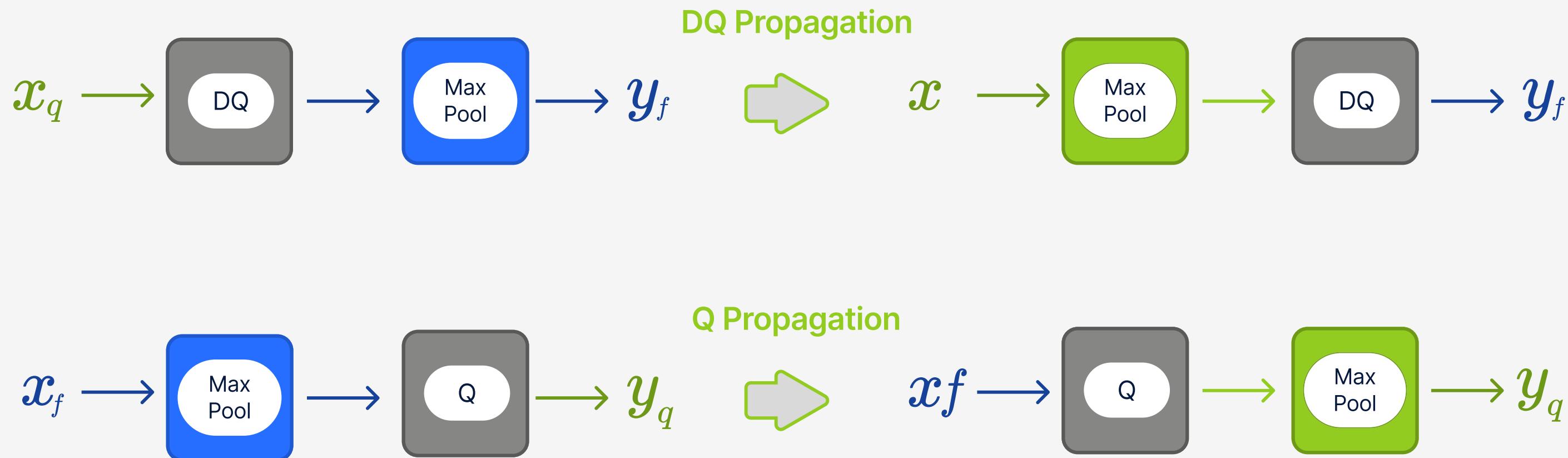
- 4 MatMul не квантуется, если оба входа не веса

- 5 Квантуйте оба входа Add операции (skip connection)

Fake quant (Q/DQ)



Fake quant (Q/DQ)



Главное

-
- 1 Начните с Post Training Quantization
 - 2 Используйте `lInt8MinMaxCalibrator` для NLP и LLM,
в остальных случаях `lInt8EntropyCalibrator2`
 - 3 Для Imagenet сетей достаточно ~500 картинок
 - 4 Если просела точность придется заняться Quantization
Aware Training
 - 5 Последний слой классификатора обычно не квантуется
(большая просадка точности)
-

TensorRT LLM

Что такое TensorRT LLM?

Это надстройка над TensorRT, которая предназначена для работы с большими языковыми моделями

Почему вообще понадобилось выделять TensorRT LLM в отдельный проект?

- Attention блоки плохо оптимизируются как набор отдельных операции. Attention нужно оптимизировать как целое
- Квантование LLM моделей требует особый подход
- LLM модели не всегда надо ускорять, зачастую требуется в первую очередь оптимизировать память

Оптимизация attention

Compute-bound
операции



время исполнения определяется количеством элементарных арифметических операций. Как пример можно привести conv с большим количеством каналов

Memory-bound
операции



время исполнения определяется кол-вом обращений к памяти, сами вычисления занимают меньшую часть общего времени исполнения. Как пример можно привести поэлементные операции, LayerNormalization...

Оптимизация attention

Flash attention

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d}$$

softmax — memory bound

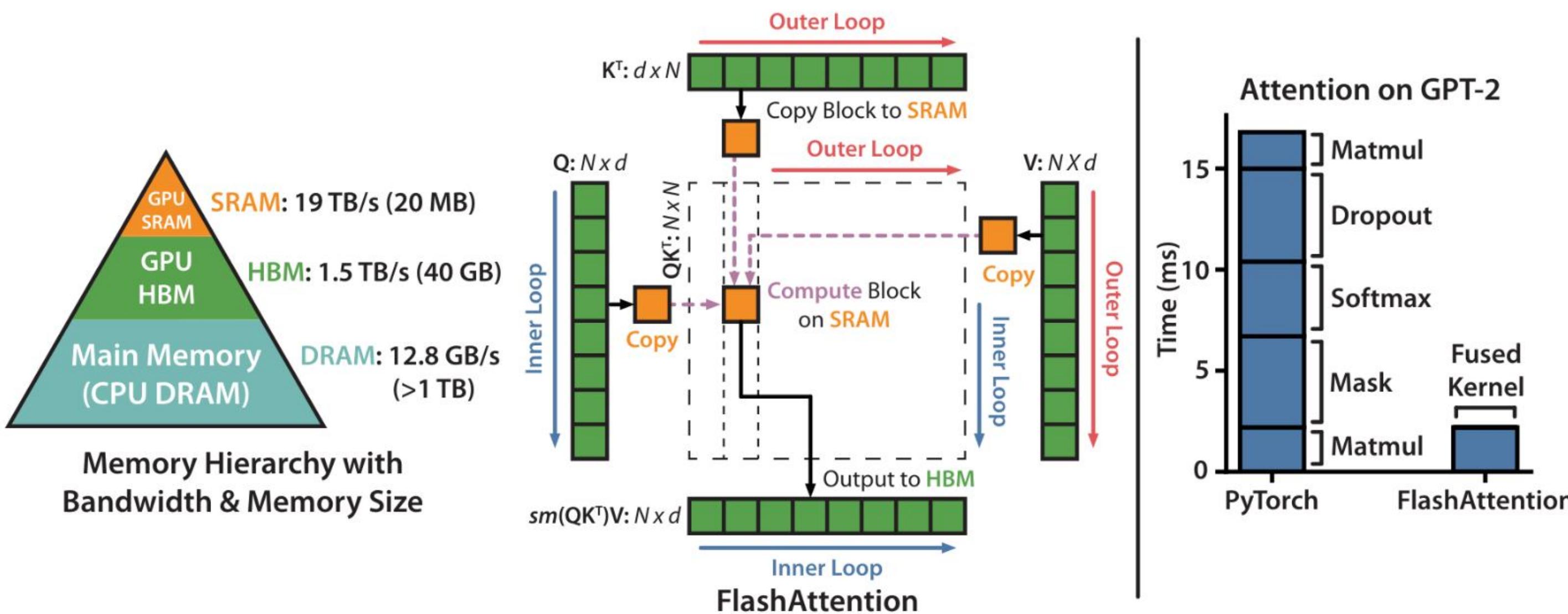
Основная идея для ускорения расчетов
— это не хранить матрицу \mathbf{P} целиком

[arxiv.org/pdf/2205.14135 ↗](https://arxiv.org/pdf/2205.14135.pdf)

mask — memory bound

Нужно разбить query, key и value
на блоки и считать \mathbf{P} блоками (tiling)
которые можно держать в самой
быстрой памяти SRAM не обращаясь
к основной памяти

Оптимизация attention Flash attention



Что добавили в TensorRT LLM

-
- 1 В TensorRT LLM добавлены плагины с оптимизацией памяти аналогичной flash attention
 - 2 Поддерживается INT4 для весов
 - 3 Поддерживается BF16, FP16, FP8
 - 4 Рескейлинг квантование для LLM моделей (дает хороший результат без дообучения)
-