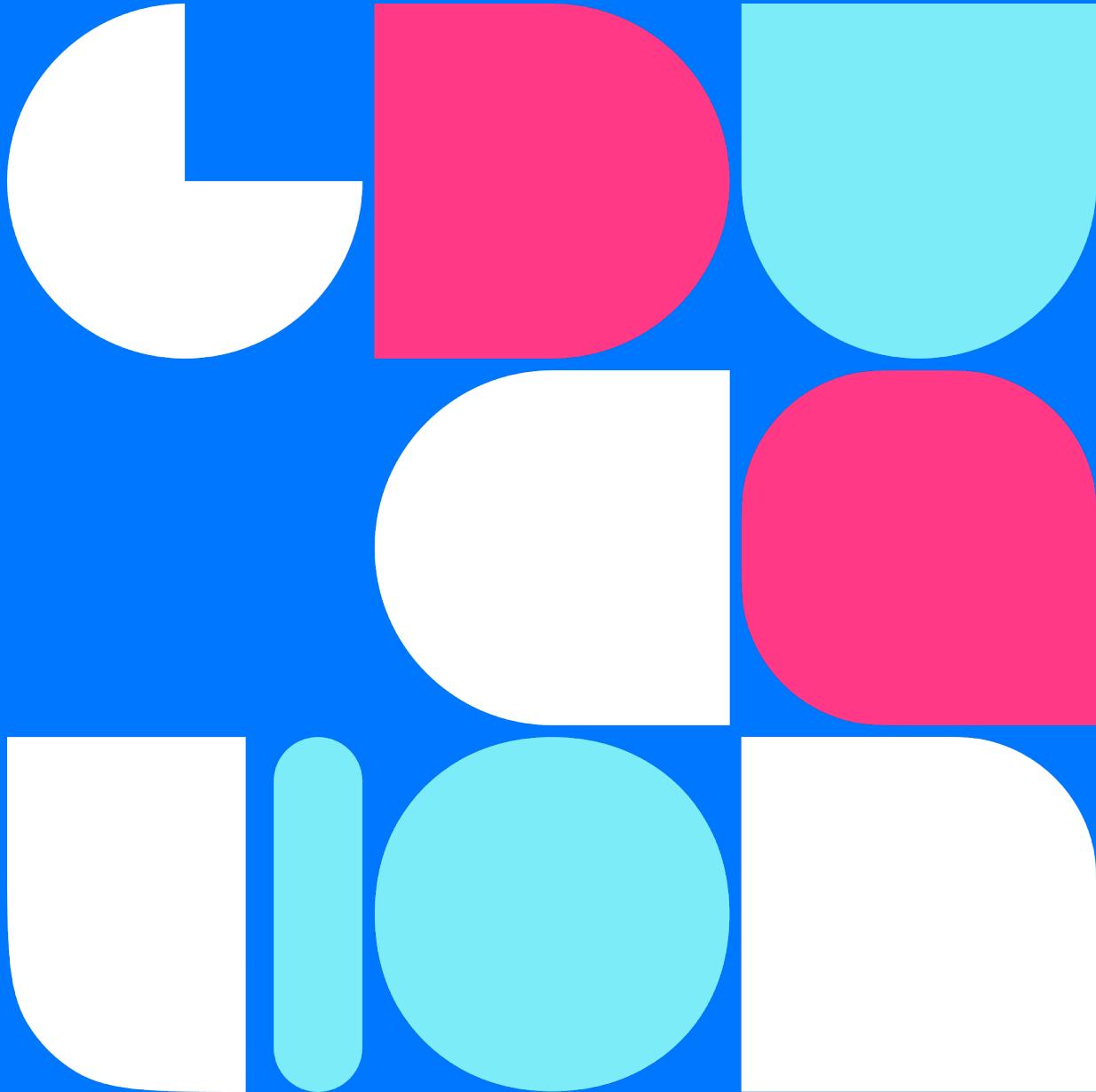




# Dealing with LARGE models

Егор Спирин



# GPU — зачем?

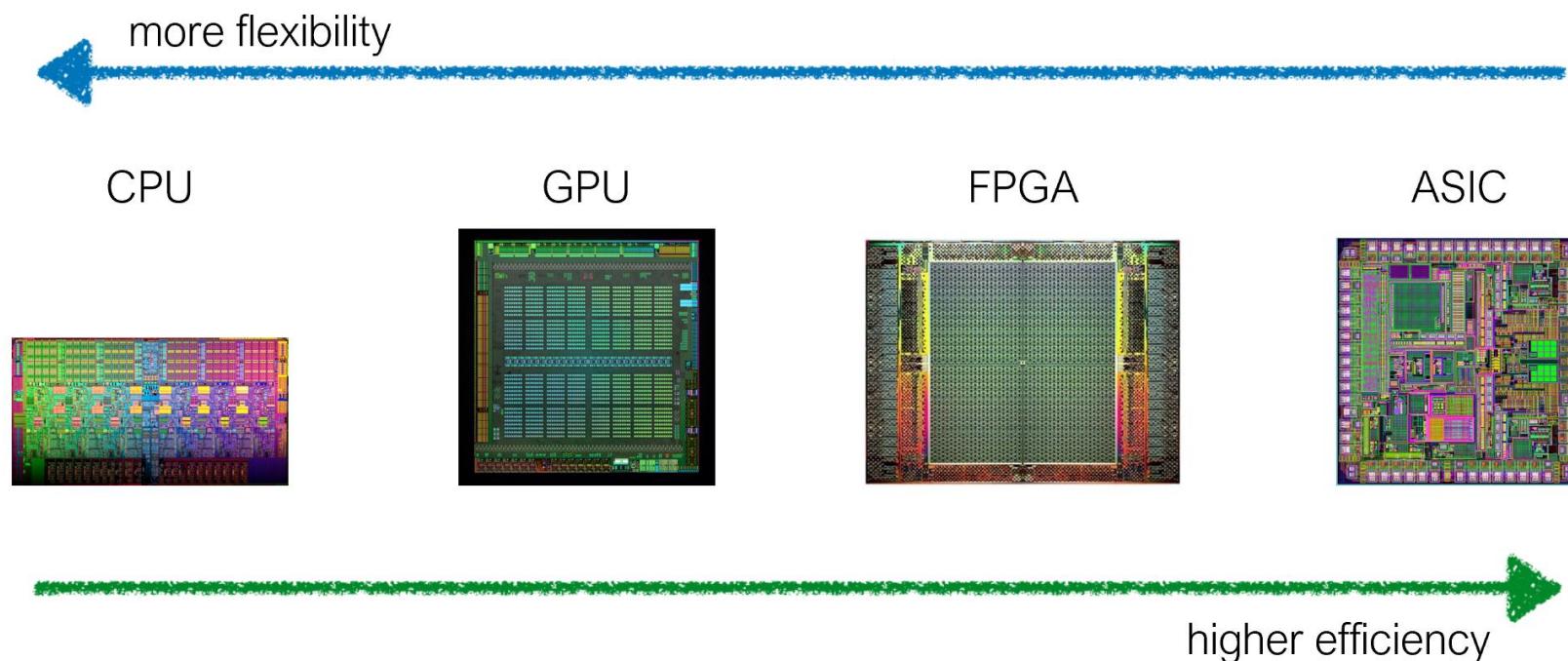
Обучение любой модели — большая вычислительная задача

⇒ Используем специальные устройства для ускорения

👉 GPU/TPU/... — хайповое направление работы

👉 Современные фреймворки поддерживают работы с разными вычислителями

👉 Важно правильно применять!



# Проблема



7B модель уже не редкость, но

- 👉 7B параметров, каждый float32  $\Rightarrow \frac{7 \cdot 10^9 \cdot 4}{1024^3} \approx 26$  GB памяти
- 👉 А еще надо хранить градиенты — 26 GB
- 👉 Adam хранит 1 и 2 момент —  $26 \cdot 2 = 52$  GB памяти

# Проблема



7B модель уже не редкость, но

- 👉 7B параметров, каждый float32  $\Rightarrow \frac{7 \cdot 10^9 \cdot 4}{1024^3} \approx 26$  GB памяти
- 👉 А еще надо хранить градиенты — 26 GB
- 👉 Adam хранит 1 и 2 момент —  $26 \cdot 2 = 52$  GB памяти

Уже надо 104 GB видеопамяти, при этом еще необходимо выделить под активации, которые зависят от длины последовательности и батча, а обучать надо на триллионах токенов...

# Проблема



7B модель уже не редкость, но

- 👉 7B параметров, каждый float32  $\Rightarrow \frac{7 \cdot 10^9 \cdot 4}{1024^3} \approx 26$  GB памяти
- 👉 А еще надо хранить градиенты — 26 GB
- 👉 Adam хранит 1 и 2 момент —  $26 \cdot 2 = 52$  GB памяти

Уже надо 104 GB видеопамяти, при этом еще необходимо выделить под активации, которые зависят от длины последовательности и батча, а обучать надо на триллионах токенов...

Nvidia H100 — 80 GB

AMD Mi300X — 192 GB

LLaMA-3 — 405B

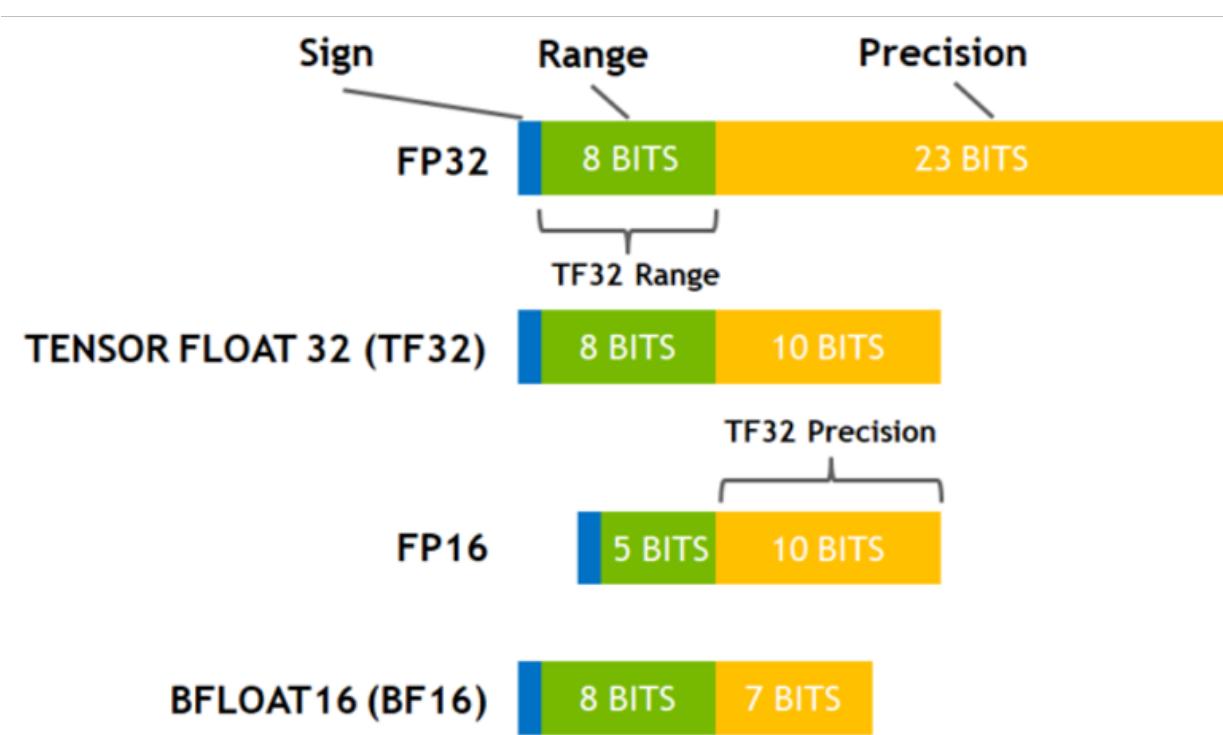
Mistral-Large — 123B

Nemotron-4 — 340B

# Half-precision

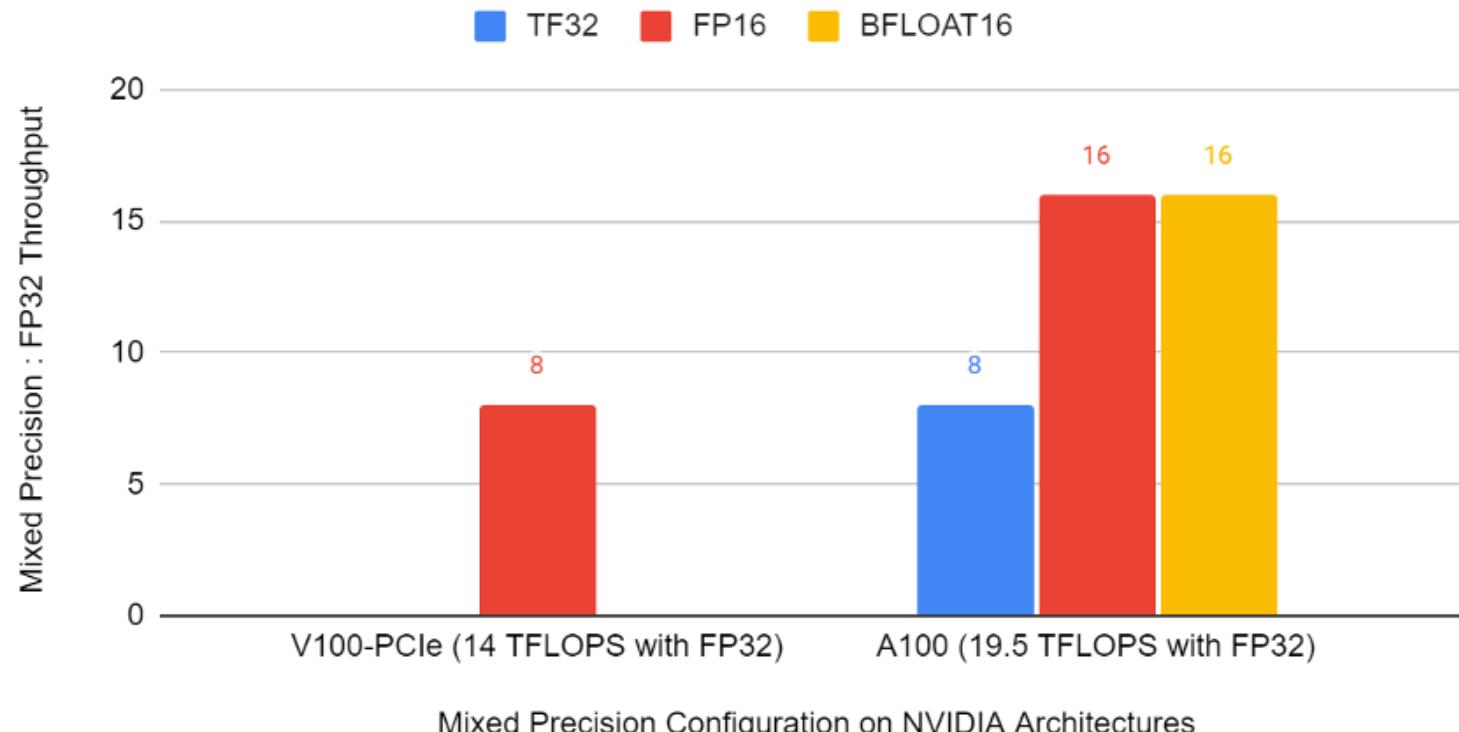
FP32 – использования 32 бит под запись числа

- 👉 Можно использовать и 16 бит, жертвуем макс. значениями и точностью
- 👉 **NVidia** предлагает собственные форматы чисел для более быстрой арифметики



# How fast is it?

Throughput Increases with Every Generation of NVIDIA Architectures



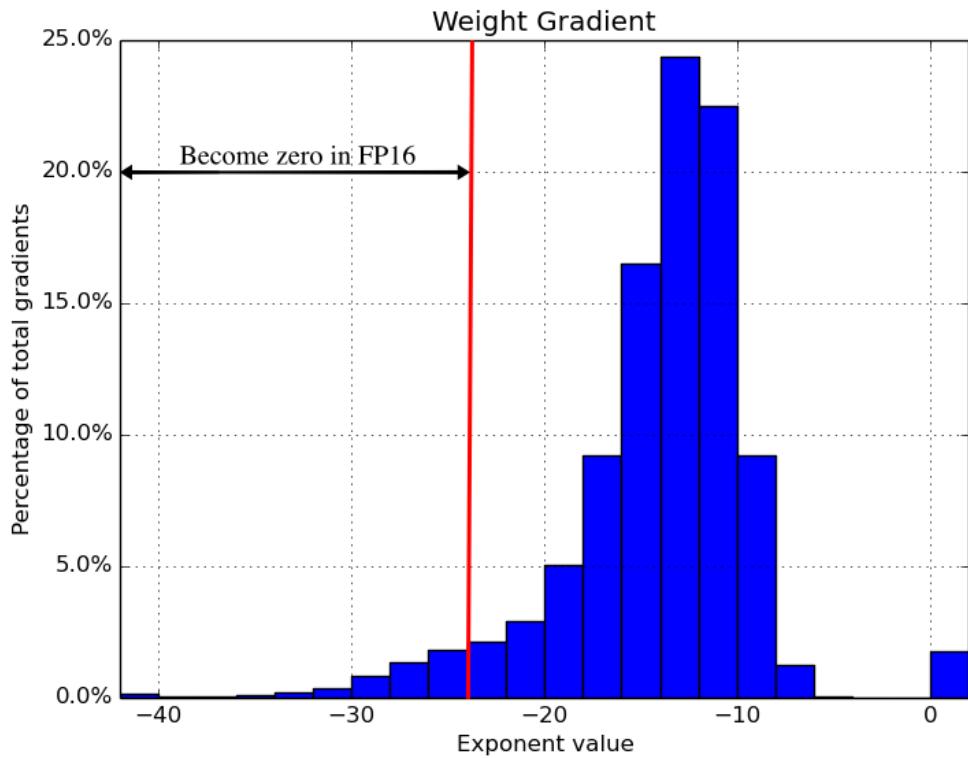
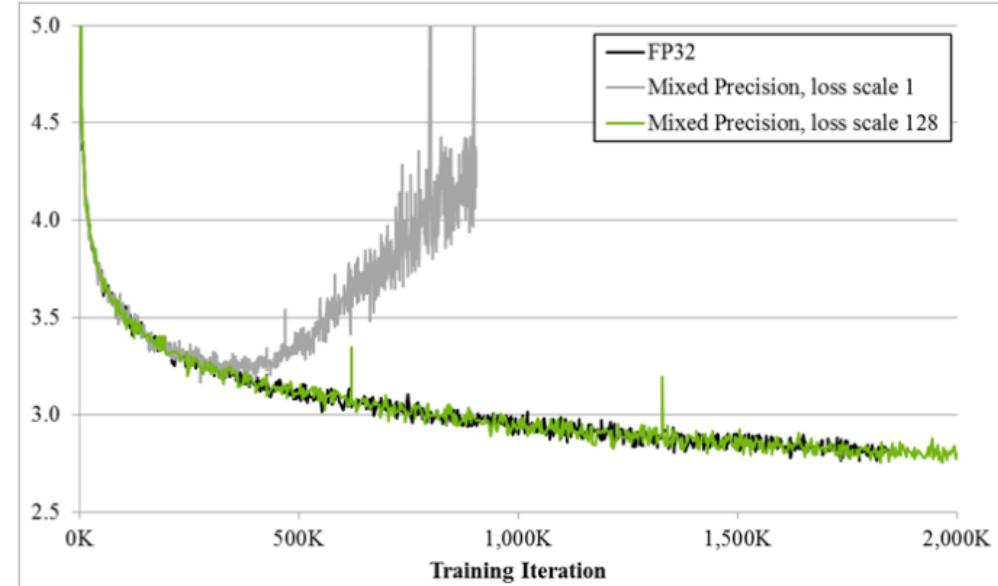
Relative peak throughput of float16 (FP16) vs float32 matrix multiplications on Volta and Ampere GPUs.

# Нюанс!

Обучение в FP16 крайне нестабильно – градиенты зануляются из-за маленьких чисел

## loss scaling:

1. Считаем forward в FP16
2. Умножаем loss на S
3. Считаем градиенты в FP16
4. Делим градиенты на 1/S
5. Обновляем веса



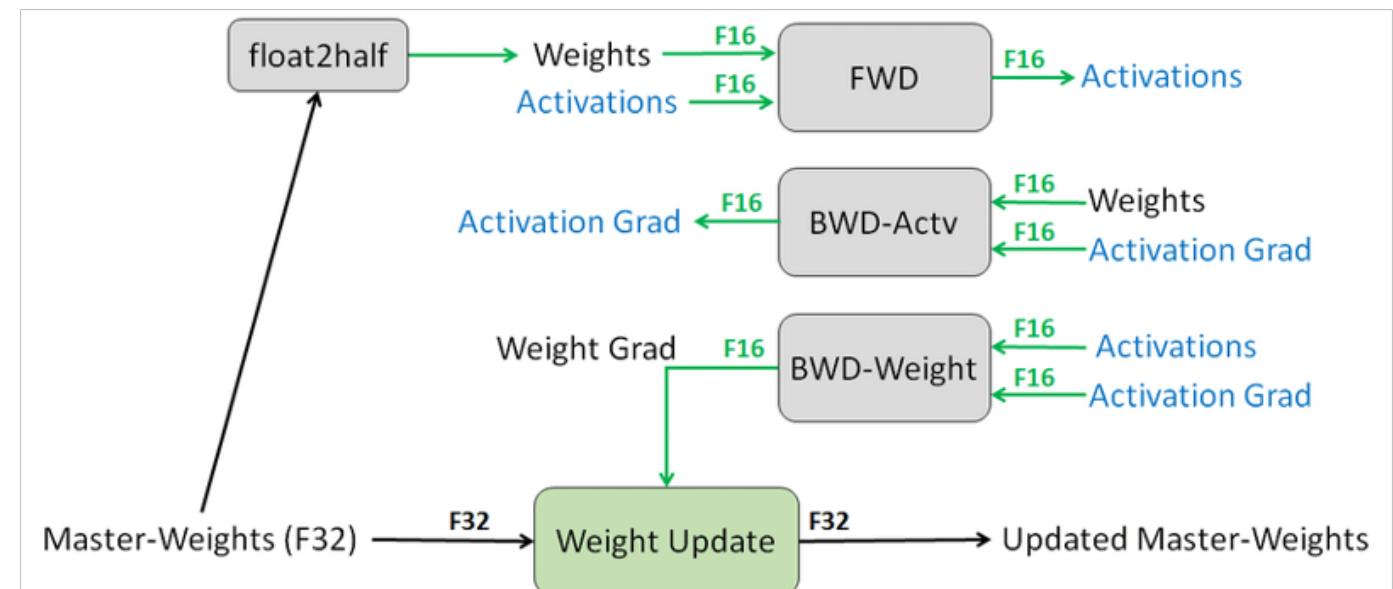
# Mixed Precision

Множество операций требуют от нас большой точности: нормализации, residual connection, gradient accumulation...

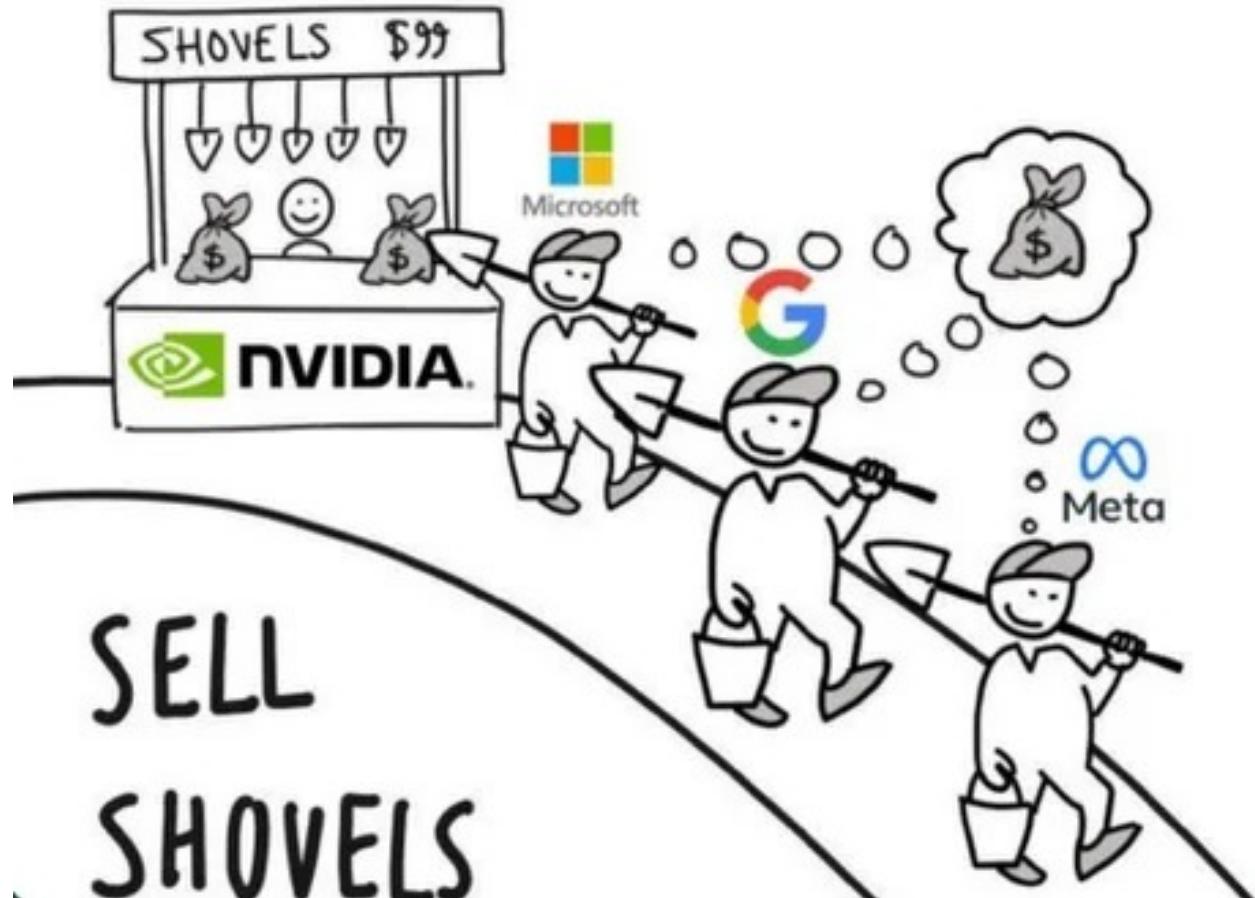
Mixed Precision – используем  
несколько форматов записи числа

- 👉 Веса модели держим в 2 форматах
- 👉 Апкастим в определенных слоях

Быстрее работаем, но больше памяти тратим :(



# WHEN EVERYONE DIGS FOR GOLD



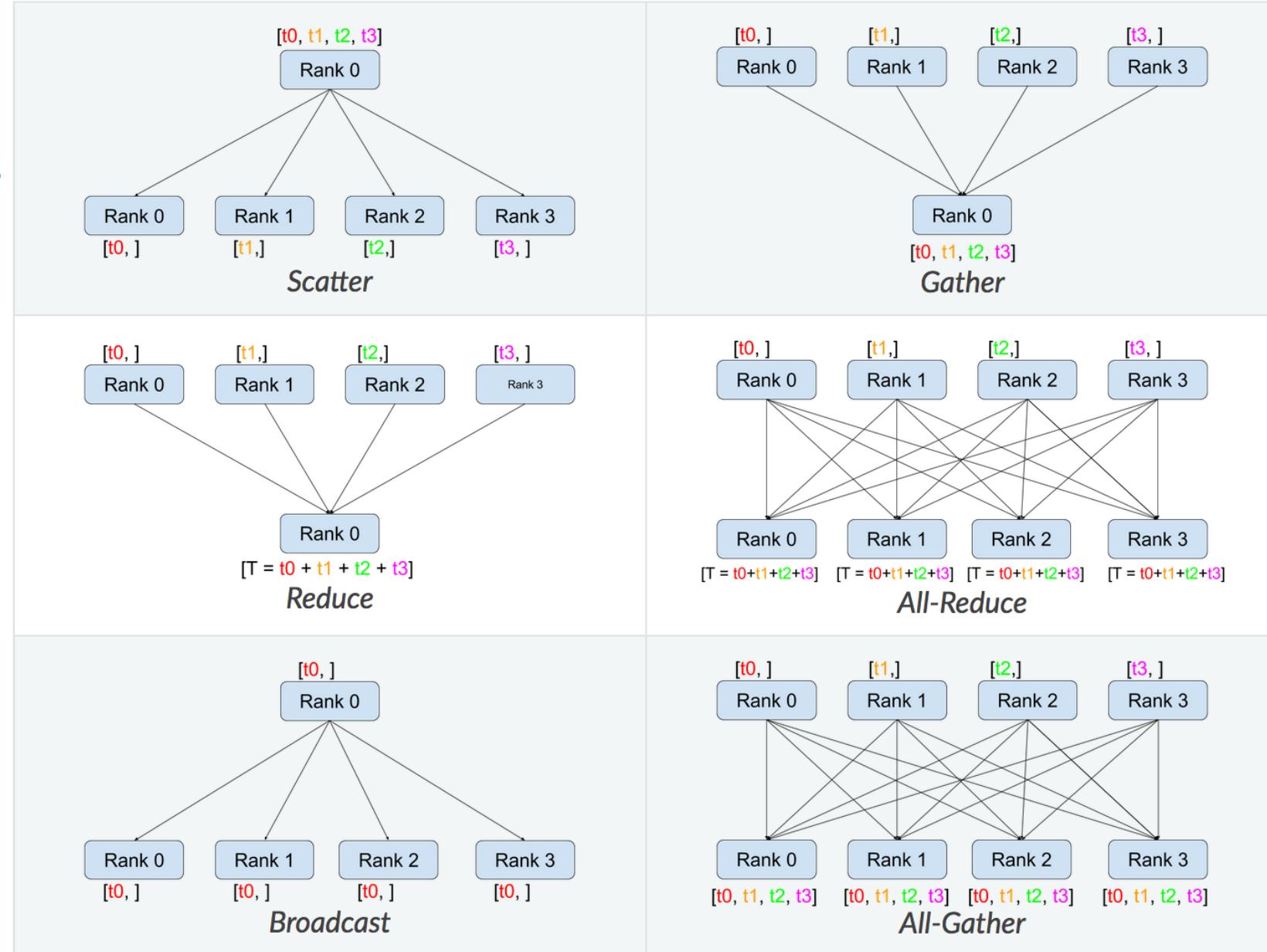
## Training Parallelism

💡 Надо использовать несколько GPU

# Collective Communications

Несколько GPU в обучении ⇒  
необходимо обмениваться  
информацией

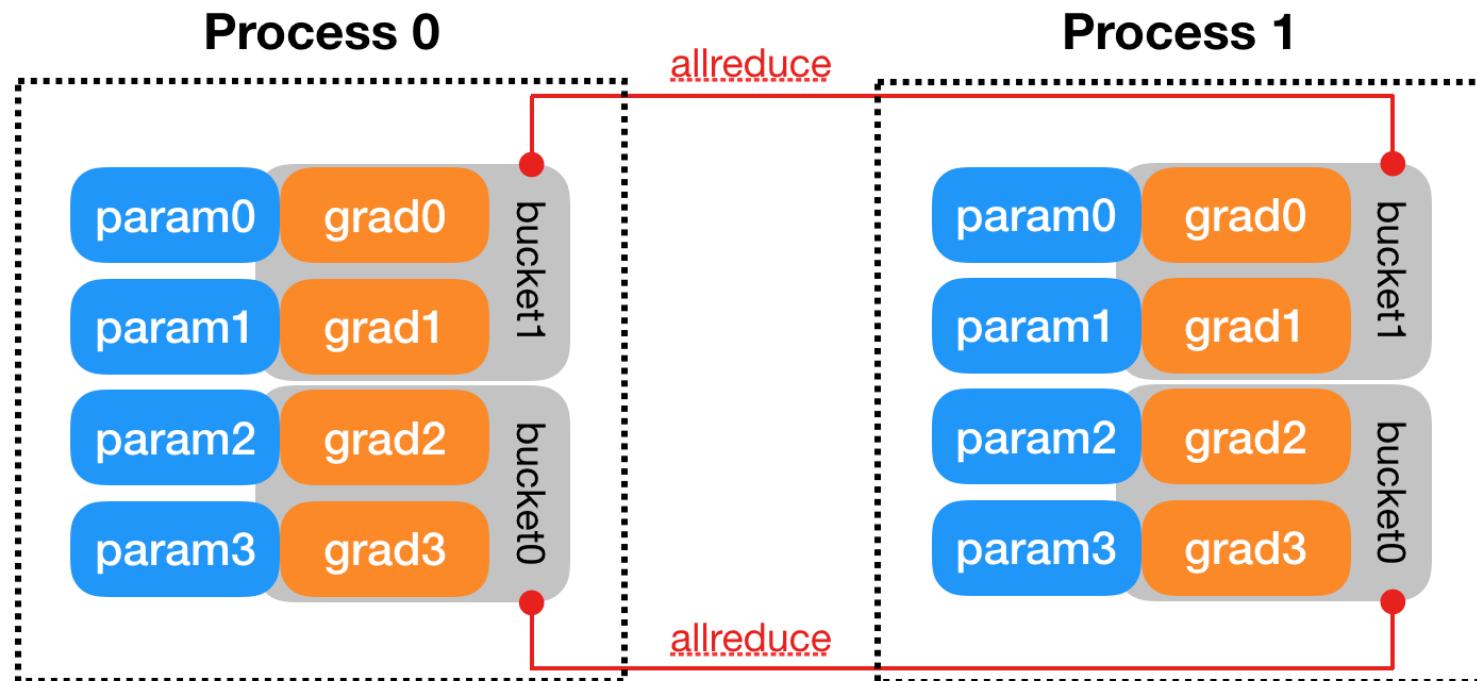
- 👉 Наиболее популярные реализации: NCCL, GLO, MPI
- 👉 Различные стратегии обхода девайсов
- 👉 Скорость передачи по сети становится очень важной



# Data Parallelism

Когда модель влезит на GPU, но необходимо увеличить размер батча

- 👉 Каждое устройство работает со своей частью данных
- 👉 После forward-backward синхронизируем градиенты и обновляем веса на каждом устройстве
- 👉 Итоговый эффективный размер батча:  $\text{batch\_size\_per\_GPU} \times \text{N\_GPU}$



# DDP in PyTorch 🔥

[PyTorch Elastic](#) — упрощает работу с distributed обучением

👉 Master — главная нода в кластере, у нее есть **MASTER\_ADDR**

👉 Параметры окружения:  
**LOCAL\_RANK** — номер GPU в ноде  
**GLOBAL\_RANK** — номер GPU в кластере  
**WORLD\_SIZE** — размер кластера

Запускаем:

```
$ torchrun --nnodes=2 --nproc_per_node=8
  --rdzv_id=100 --rdzv_backend=c10d
  --rdzv_endpoint=$MASTER_ADDR:29400
elastic_ddp.py
```

```
class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = nn.Linear(10, 10)
        self.relu = nn.ReLU()
        self.net2 = nn.Linear(10, 5)

    def forward(self, x):
        return self.net2(self.relu(self.net1(x)))

def demo_basic():
    torch.cuda.set_device(int(os.environ["LOCAL_RANK"]))
    dist.init_process_group("nccl")
    rank = dist.get_rank()
    print(f"Start running basic DDP example on rank {rank}.")
    # create model and move it to GPU with id rank
    device_id = rank % torch.cuda.device_count()
    model = ToyModel().to(device_id)
    ddp_model = DDP(model, device_ids=[device_id])
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(device_id)
    loss_fn(outputs, labels).backward()
    optimizer.step()
    dist.destroy_process_group()
    print(f"Finished running basic DDP example on rank {rank}.")
```

# Sharded DP

Но иногда размер модели больше памяти на GPU...

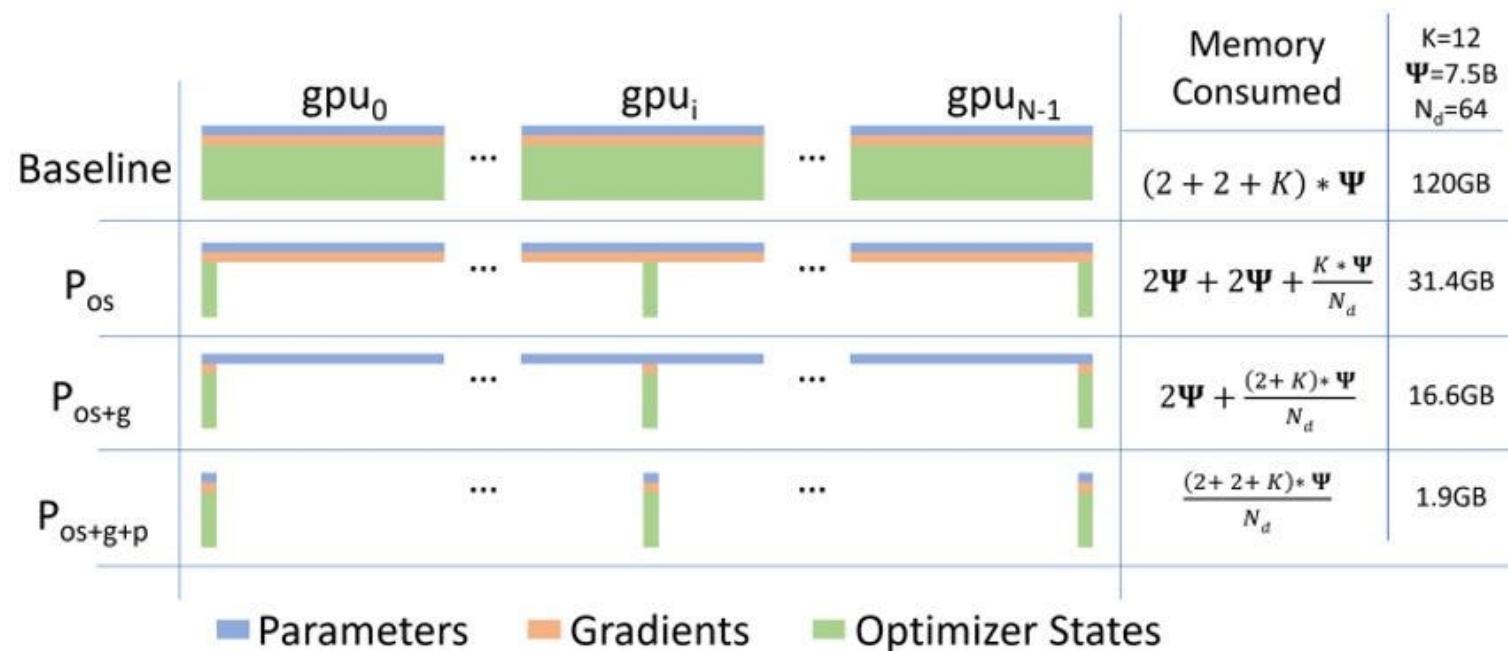
Можем **шардировать** тяжелые части:

Stage 1: оптимизатор

Stage 2: + градиенты

Stage 3: + параметры модели

Все еще DP — каждая GPU работает со своей частью данных, параметры синхронизируют по запросу



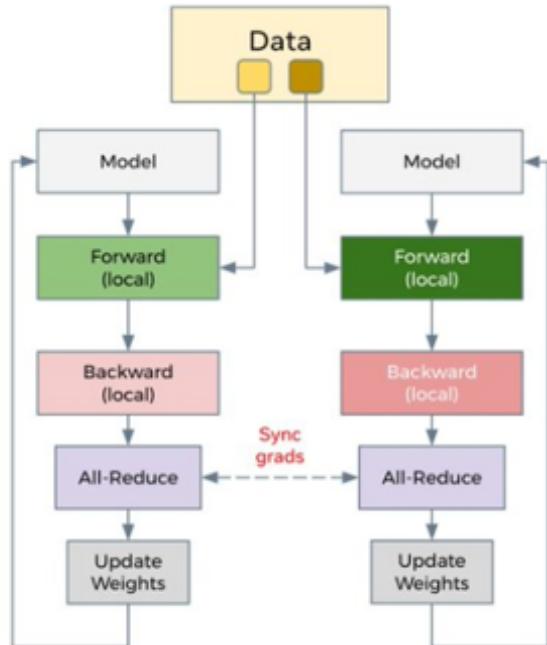
DeepSpeed, Torch FSDP

[2] — ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, Samyam Rajbhandari, Jeff Rasley, SC'20

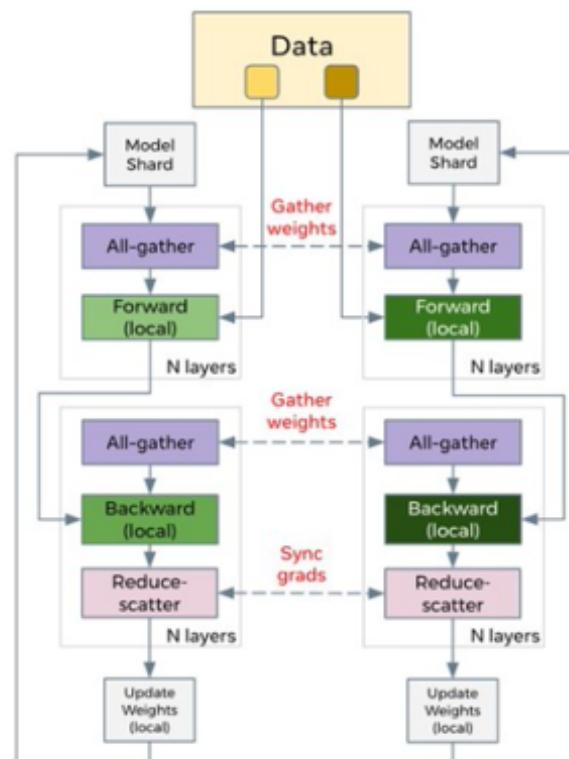
[3] — PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel, Zhao et al., VLDB Endowment'23

# FSDP in PyTorch 🔥

Standard Data Parallel Training



Fully Sharded Data Parallel Training



[FSDP](#): базовое использование аналогично DDP,  
НО ТАКЖЕ МНОГО НЮАНСОВ И ПАРАМЕТРОВ

```
rank = dist.get_rank()
device_id = rank % torch.cuda.device_count()
model = ToyModel().to(device_id)
fsdp_model = FSDP(
    model,
    param_init_fn=my_init_fn,
    auto_wrap_policy=my_wrap_policy
)

optimizer = optim.SGD(fsdp_model.parameters(), ...)
optimizer.zero_grad()
outputs = fsdp_model(torch.randn(20, 10))
...
```



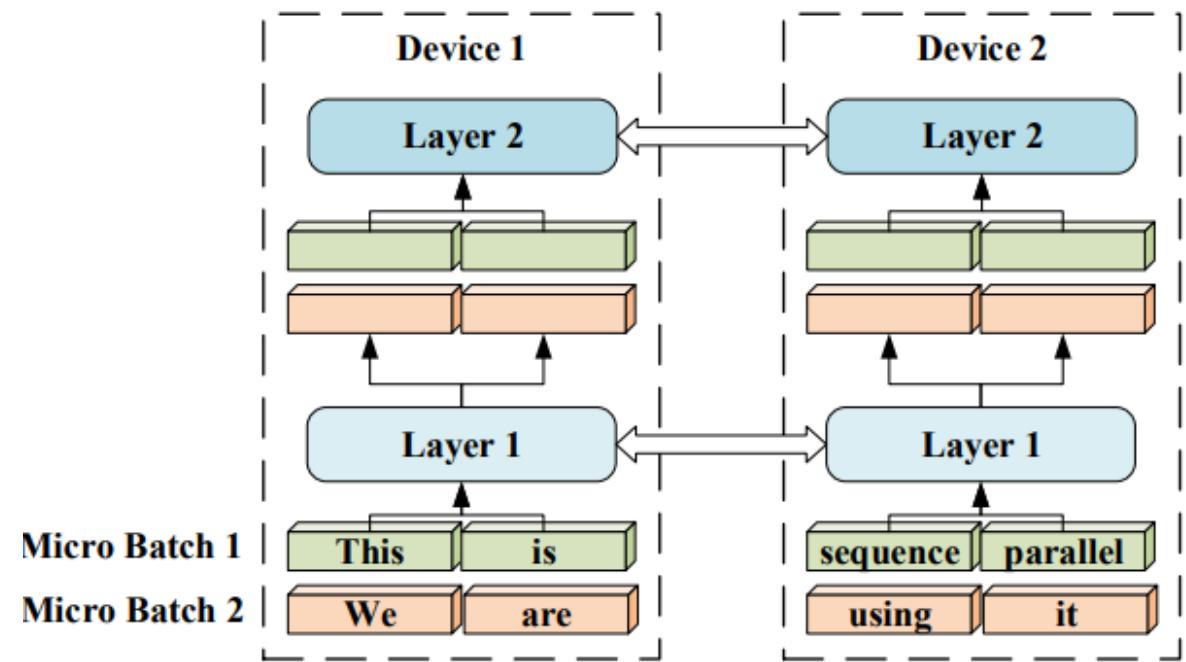
# Huge Global Batch Size Issue

Больше кластер — сильнее дробление на шарды  $\Rightarrow$  больше свободного места

Постоянно увеличивать размер батча нельзя, зато можно увеличить длину контекста!

**Sequence parallelism** — обрабатываем  
более длинные последовательности

- 👉 Нативно можем параллеризовать Dropout, LayerNorm, ...
- 👉 Больше всего памяти уходит на Multi-Head Attention
- 👉 Colossal-AI, Megatron-LM, DeepSpeed-Ulysses



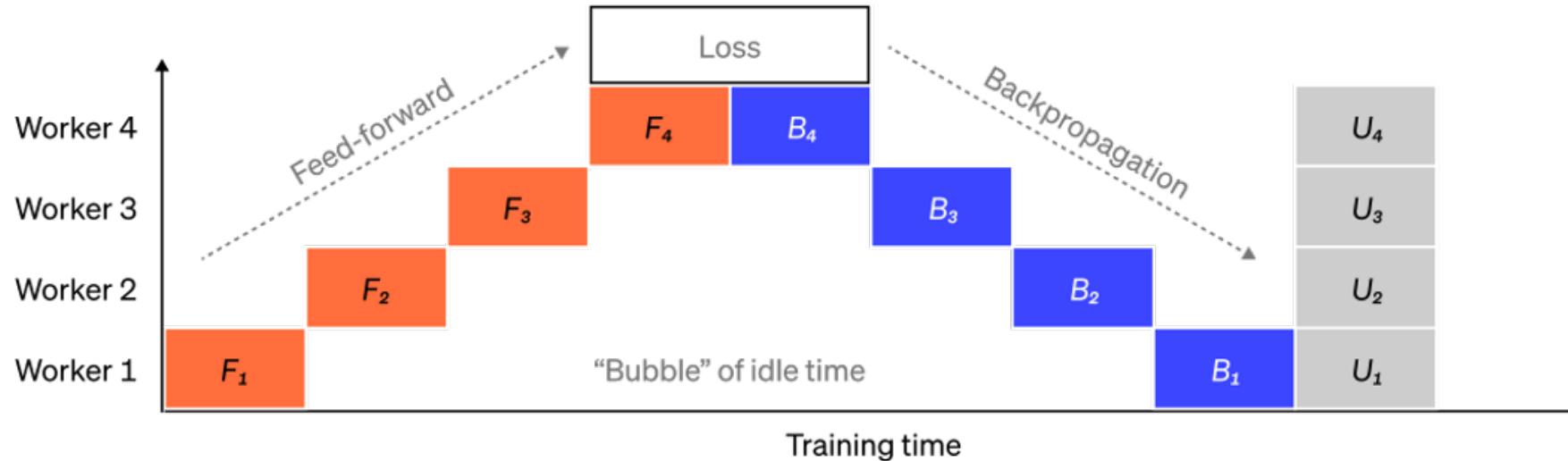
[4] — Sequence Parallelism: Long Sequence Training from System Perspective, Shenggui Li, and Fuzhao Xue et al., ACL'23

[5] — Reducing Activation Recomputation in Large Transformer Models, Korthikanti et al., MLSYS'23

[6] — DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models, Jacobs et al., PODC'24

# Pipeline Parallelism

Разбиваем **модель** на последовательные блоки, каждый помещается на отдельный девайс

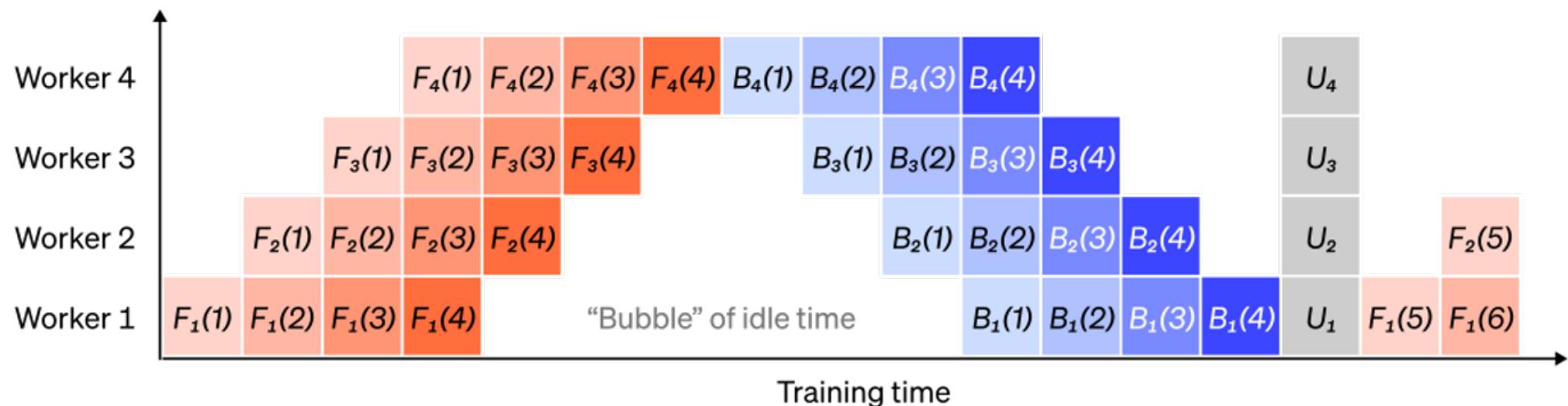


- 👉 Батч с данными загружается на Worker 1
- 👉 Worker X обрабатывает батч и передает Worker X+1
- 👉 Во время backward повторяем в обратную сторону
- 😢 В моменте используем только 1 GPU, остальные простоявают
- 😢 Хорошо, если модель бьется на последовательные блоки...

# GPipe

💡 Пока ждем остальных считаем следующий микробатч (chunk) ~ аккумуляция градиентов

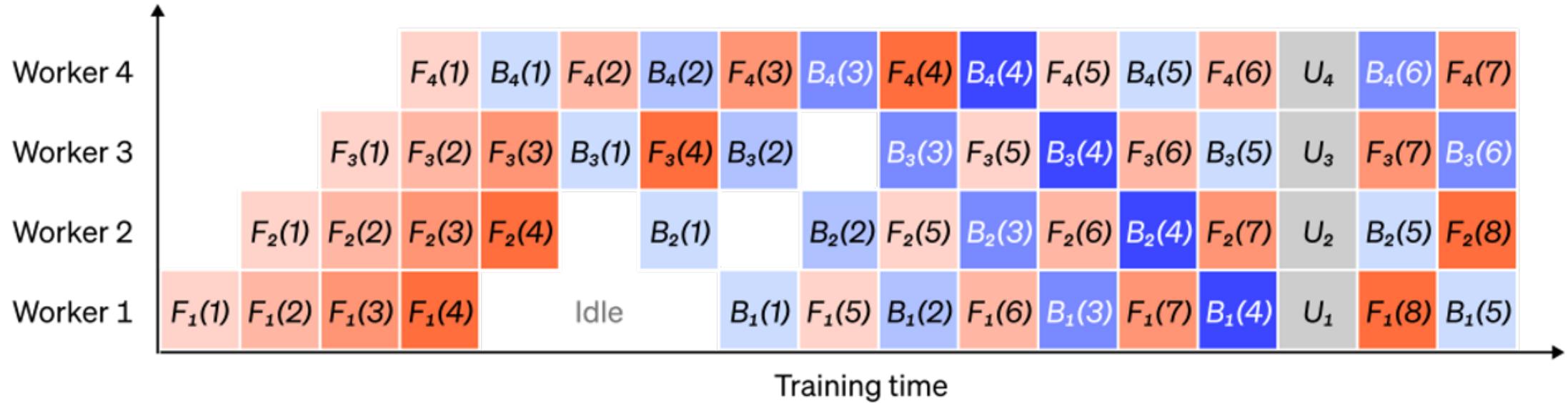
Теперь размер пузыря зависит от числа чанков



👉 1 чанк ~ наивный РР

👉 Много чанков — либо слишком маленькие микробатчи, либо слишком большой итоговой батч

# PipeDream | порядок важен!



Нет единой версии модели, каждый воркер работает со своей копией

- 👉 Weight Stashing: храним несколько копий модели, для каждой пары forward-backward
- 👉 Vertical Sync: иногда синхронизируем воркеров

[8] — PipeDream: Fast and Efficient Pipeline Parallel DNN Training, Harlap et al., SOSP'19

[9] — Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, Narayanan et al., SC'21

[10] — Breadth-First Pipeline Parallelism, Joel Lamy-Poirier et al., MLSys'23

# PP in PyTorch 🔥

Pipeline Parallelism сложнее интегрировать — требует разбираться в архитектуре каждой модели + не подходит к каждой модели

- 👉 Строится DAG по трассировке модели
- 👉 Операции и параметры разделяются по группам
- 👉 Переписывается `forward` метод

`SplitPoint.BEGINNING/END` — сплит до или после указанного модуля

```
class Model(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.emb = torch.nn.Embedding(10, 3)
        self.layers = torch.nn.ModuleList(
            Layer() for _ in range(2)
        )
        self.lm = LMHead()

    def forward(self, x: torch.Tensor):
        x = self.emb(x)
        for layer in self.layers:
            x = layer(x)
        x = self.lm(x)
        return x

model = Model()
# An example micro-batch input
x = torch.LongTensor([1, 2, 4, 5])

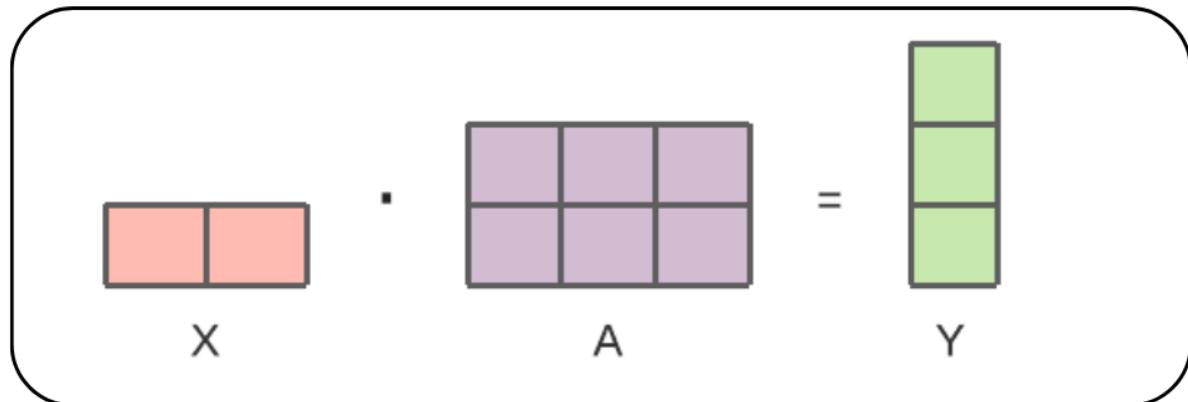
pipe = pipeline(
    module=model,
    mb_args=(x,),
    split_spec={
        "layers.1": SplitPoint.BEGINNING,
    }
)
```

# Tensor Parallelism

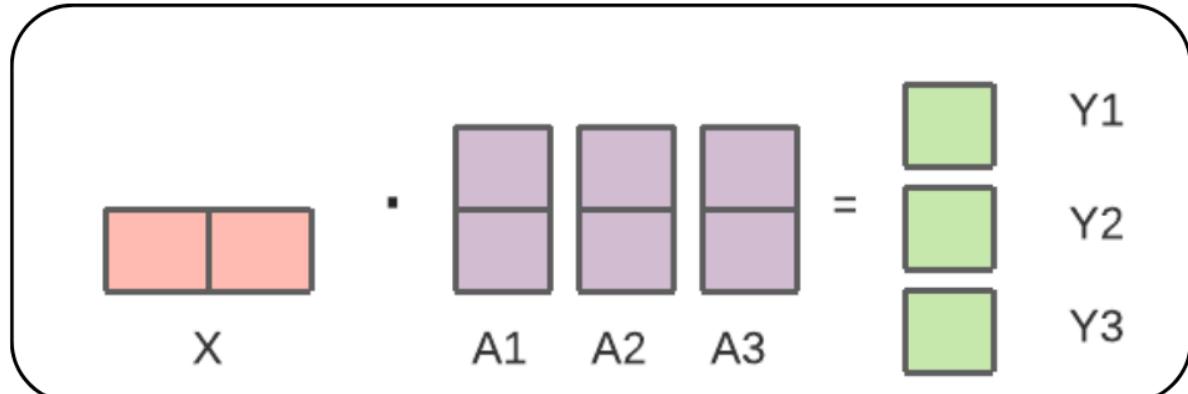
Большие матричные умножения можно разбить на несколько устройств

- 👉 PyTorch 2.3+
- 👉 DeepSpeed (tensor slicing)
- 👉 Megatron-LM

inputs      weights      outputs



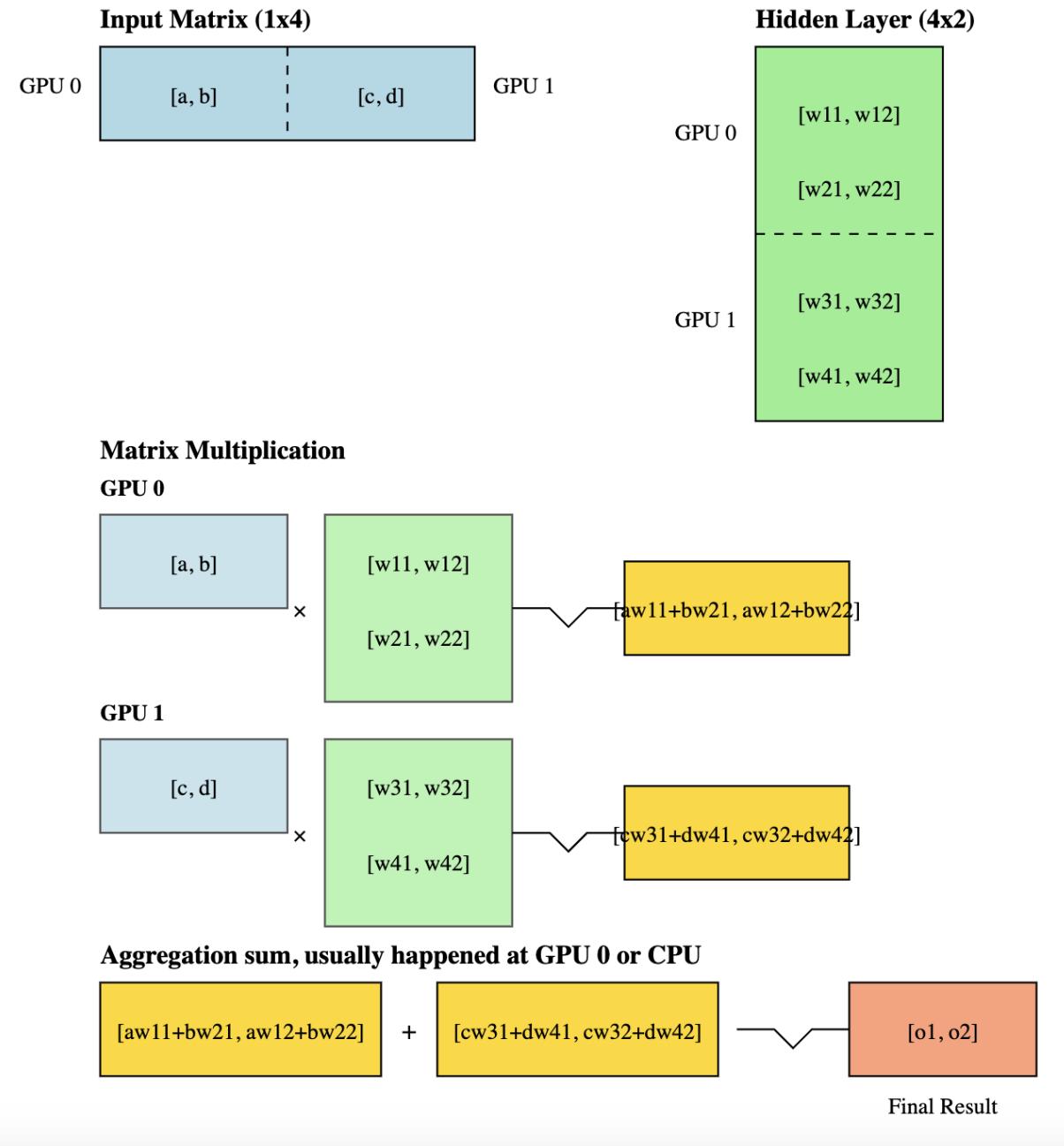
is equivalent to



# Row-Wise Parallel

Матрица весов разбивается  
**построчно** на  $n$  частей

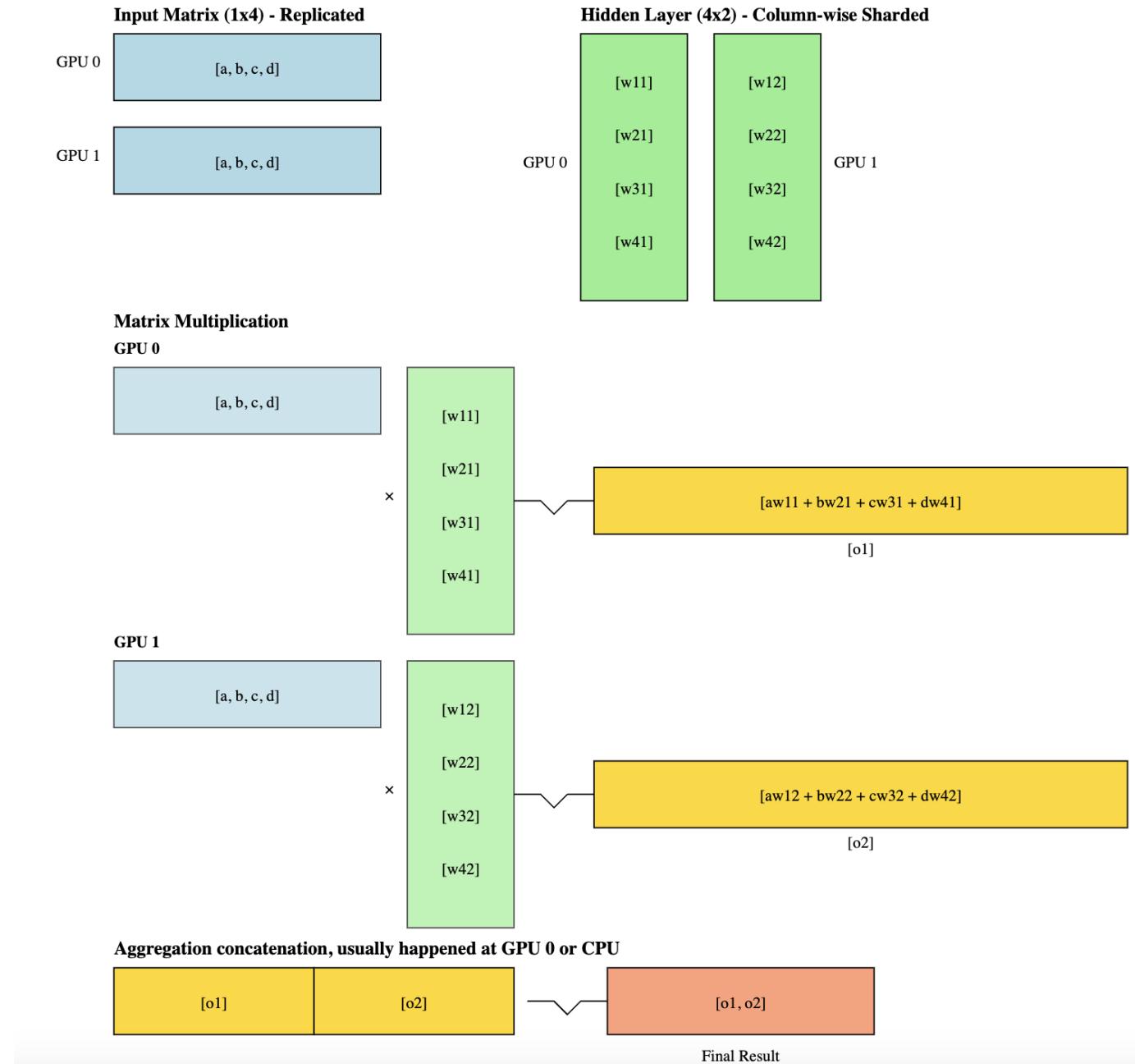
1. Input бьется на  $n$  частей, каждая часть отправляется на отдельный девайс
2. На девайсе перемножаем часть input и часть весов
3. Собираем все части на мастер устройстве
4. Агрегрируем — суммируем покомпонентно



# Column-Wise Parallel

Матрица весов разбивается **по столбцам** на n частей

1. Input реплицируется n раз, каждая копия отправляется на отдельный девайс
2. На девайсе перемножаем копию input и часть весов
3. Собираем все части на мастер устройстве
4. Агрегрируем — конкатенация

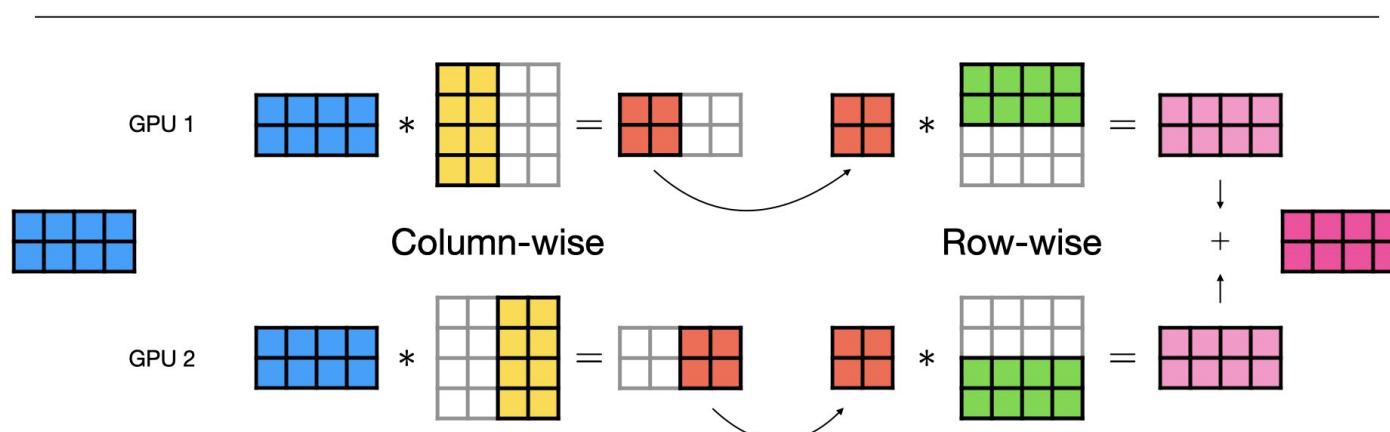


# Row vs. Column

Выбор параллелизации зависит от типа слоя, его очередности и комбинации с остальными

- 👉 Embeddings, CrossEntropyLoss — row-wise параллелизация, разные токены словаря уходят на разные устройства
- 👉 MLP:  $\text{ACT}(X \cdot W_1 + b_1) \cdot W_2 + b_2$ 
  - Если ReLU активация, то нельзя row-wise:  $\max(0, X_1) + \max(0, X_2) \neq \max(0, X_1 + X_2)$
  - Оптимально комбинировать!

$$\begin{array}{c} \text{blue grid} \\ * \end{array} * \begin{array}{c} \text{yellow grid} \\ * \end{array} * \begin{array}{c} \text{green grid} \\ = \end{array} \begin{array}{c} \text{pink grid} \end{array}$$



# TP in PyTorch 🔥

Совсем новинка и может постоянно меняться API

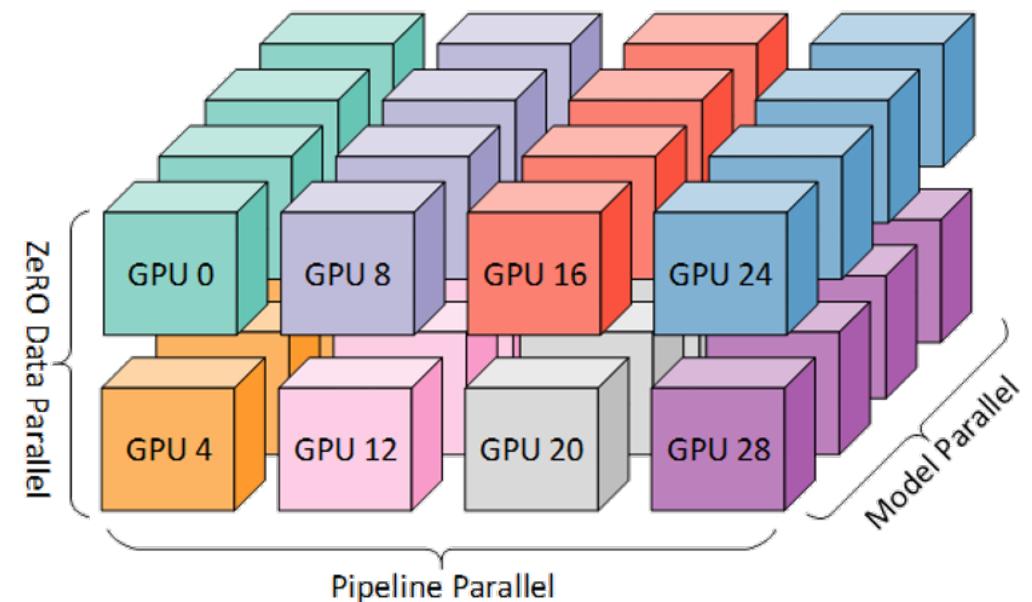
- 👉 Работает совместно с FSDP — FSDP 2.0
- 👉 Указываем mesh и plan — по каким девайсам будет DP, по каким TP
- 👉 Отдельно можно делать SequenceParallel

```
class FeedForward(nn.Module):  
    def __init__(self, input_dim, hidden_dim):  
        super().__init__()  
        self.w1 = nn.Linear(input_dim, hidden_dim)  
        self.w2 = nn.Linear(hidden_dim, input_dim)  
        self.w3 = nn.Linear(input_dim, hidden_dim)  
  
    def forward(self, x):  
        return self.w2(F.silu(self.w1(x)) * self.w3(x))  
  
# Device mesh  
tp_mesh = init_device_mesh("cuda", (8,))  
  
# Mesh  
layer_tp_plan = {  
    "w1": ColwiseParallel(),  
    "w2": RowwiseParallel(),  
    "w3": ColwiseParallel(),  
}  
  
module = FeedForward(...)  
parallelize_module(  
    module=module,  
    device_mesh=tp_mesh,  
    parallelize_plan=layer_tp_plan,  
)
```

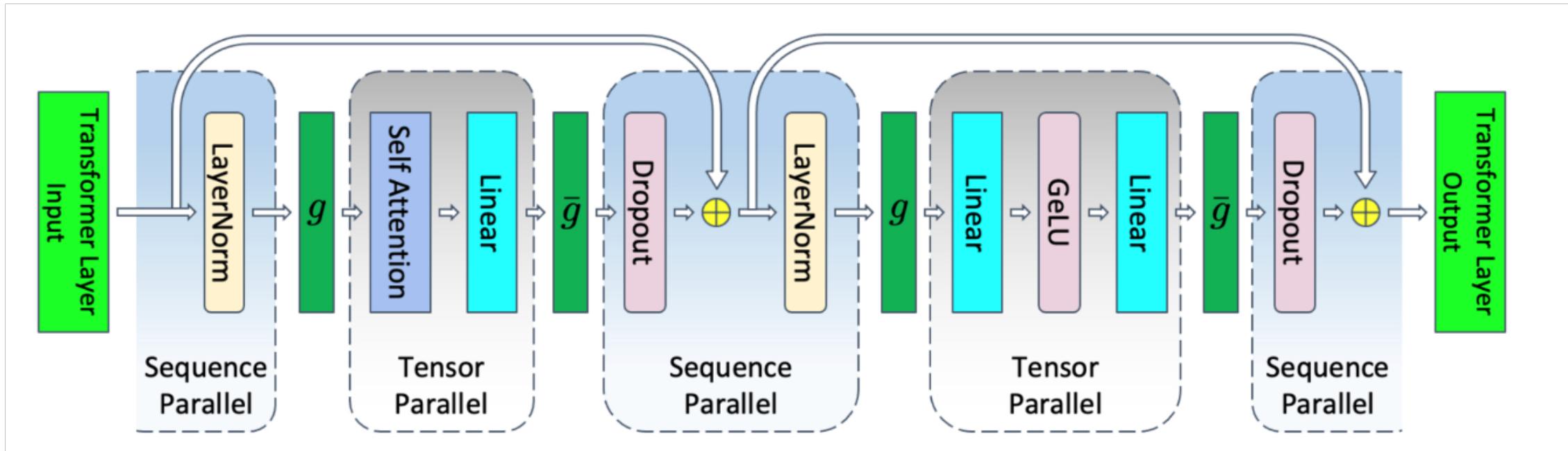
# 2D, 3D, 4D, ...

Почти все можно комбинировать друг с другом!

- 👉 8 GPU — DP на 8 девайсов
- 👉 8x8 GPU
  - Sharded DP на 64 девайса
  - DP на 8 девайсов + PP на 8 девайсов
  - DP на 8 девайсов + TP на 8 девайсов
- 👉 8x8xN GPU
  - Sharded DP на все девайсы
  - DP + PP + TP
  - Подключаем SP



Важно понимать нагрузку на сеть!



# Tl;Dr

## Data Parallelism

- + Не требует модификации модели
- + AllReduce раз в шаг
- Никак не оптимизирует память

## Sharded Data Parallelism

- + Не требует модификации модели
- + Существенно сокращает потребление памяти
- Forward и Backward делают множество синхронизаций между девайсами

## Tensor Parallelism

- + Более "широкие" слои
- + Оптимизирует потребление памяти
- Требует вмешательство модели
- Scatter/Reduce раз в шаг

## Pipeline Parallelism

- + Оптимизирует потребление памяти
- + Scatter/Reduce раз в шаг для 1 девайса
- Существенные доработки в коде модели
- Работает не со всеми моделями

# Все еще не хватает памяти

## 1. CPU Offload

Давайте сгружать на CPU что-то ненужное

1. Оптимизатор (DeepSpeedCPUAdam)
2. Активации

## 2. Activation Recomputation / Gradient Checkpointing

Давайте пересчитывать активации заново

1. Определяем границы пересчета, например, слой трансформера или простая активация
2. Поддерживается в моделях от 😊

## 3. Memory-efficient оптимизаторы

1. 4-byte оптимизаторы: Adafactor, LION
2. 8-bit оптимизаторы: bitsandbytes.Adam8bit

## 4. Fused Kernels

Объединяем несколько операций в одну

1. Flash Attention, Liger Kernels, ...

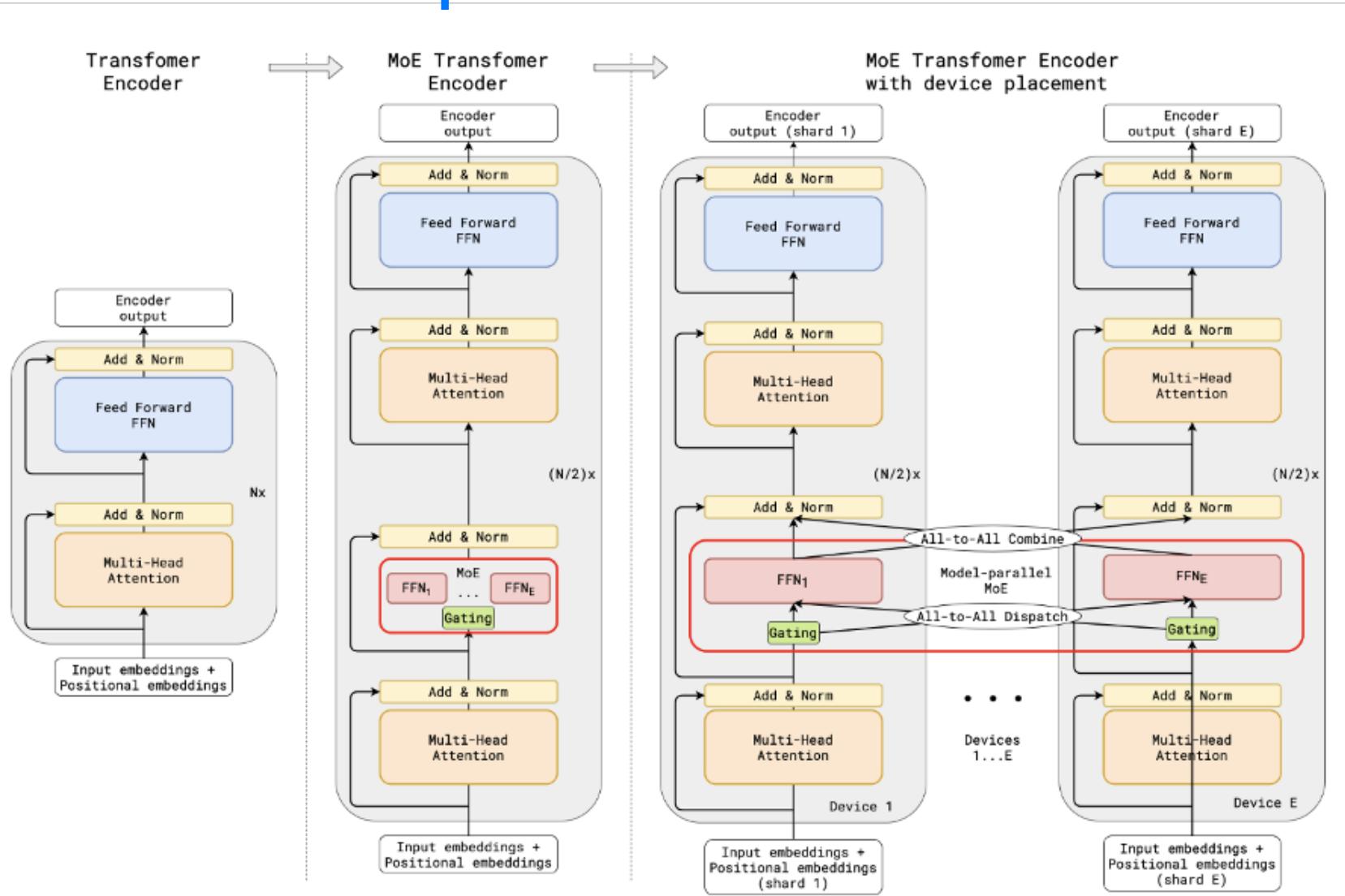


[12] — Reducing Activation Recomputation in Large Transformer Models, Korthikanti et al., MLSys'24

[13] — Flash-Attention: Fast and Memory-Efficient Exact Attention with IO-Awareness, Dao et al., NIPS'22

[14] — Liger Kernel: Efficient Triton Kernels for LLM Training, Hsu et al., 2024

# MoE & Expert Parallelization



MoE — sparse-модель

- 👉 Значительно увеличиваем число параметров
- 👉 Expert Parallelism — каждый эксперт помещается на отдельный девайс
- 👉 Router — определяет для каждого токена эксперта
- 👉 Перед и после эксперта надо синхронизировать токены между девайсами

# На этом все



Егор Спирин — [vk.com/boss](https://vk.com/boss)  
VK Lab — [vk.com/lab](https://vk.com/lab)