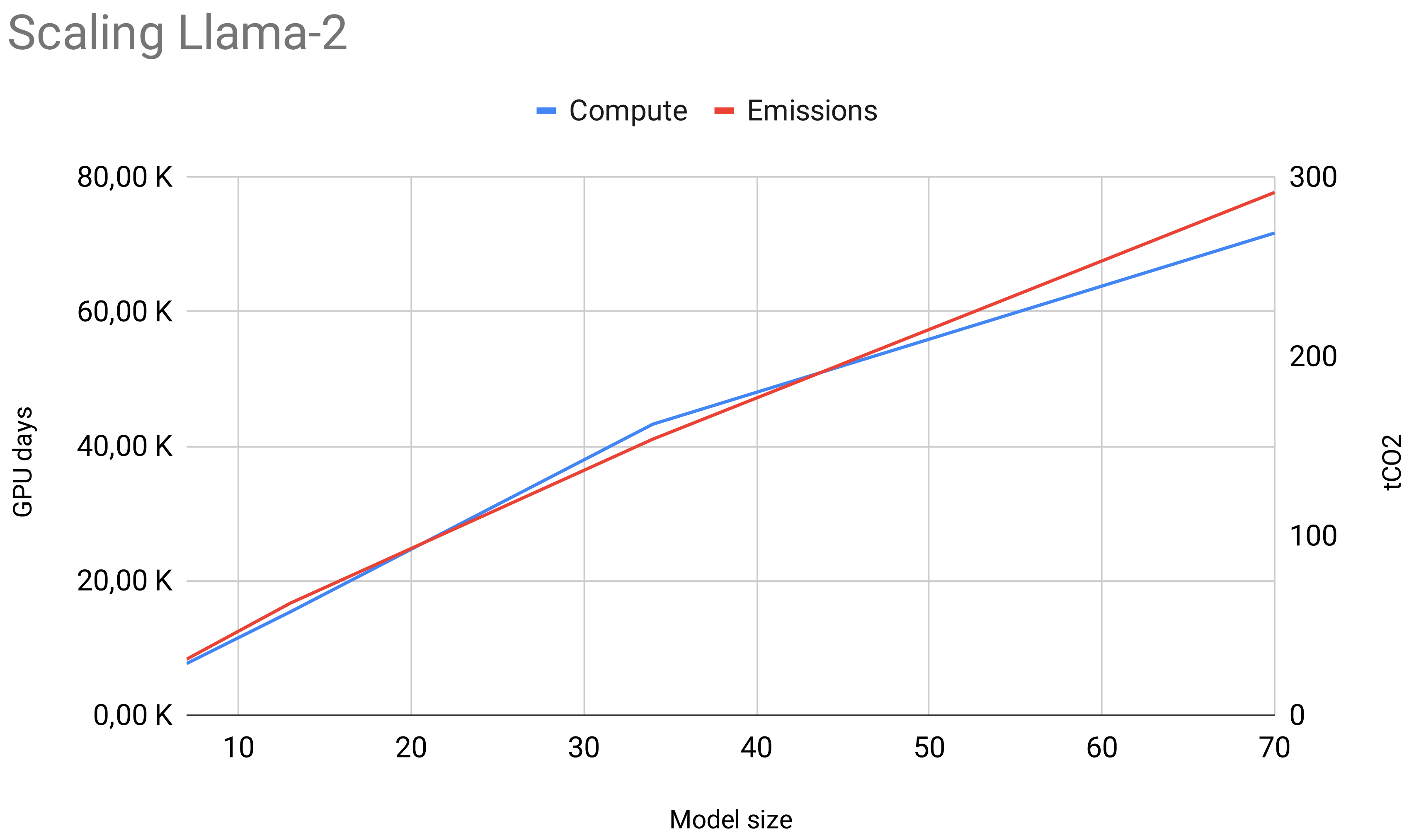
Course 4: Efficient NLP

The cost of pre-training LMs



The cost of using LMs

```
[13] from transformers import AutoTokenizer, AutoModelForCausalLM

     tokenizer = AutoTokenizer.from_pretrained("TinyLlama/TinyLlama-1.1B-Chat-v0.6")
     model = AutoModelForCausalLM.from_pretrained("TinyLlama/TinyLlama-1.1B-Chat-v0.6").cuda()
```

```
tokenizer.padding_side = "left"
sentences = tokenizer([
    "<|user|>\n What is cheesecake? \n <|assistant|>\n",
    "<|user|>\n Where is Mars? \n <|assistant|>\n",
    "<|user|>\n Who is Obama? \n <|assistant|>\n",
    "<|user|>\n Write a letter for Santa. \n <|assistant|>\n",
    "<|user|>\n Write me a recommendation letter. \n <|assistant|>\n"]*20, return_tensors="pt", padding=True, truncation=True)
model_out = model(sentences["input_ids"].cuda())
model_out
```

```
---------------------------------------------------------------------------
OutOfMemoryError                          Traceback (most recent call last)
<ipython-input-31-af62093f2f7c> in <cell line: 8>()
      6     "<|user|>\n Write a letter for Santa. \n <|assistant|>\n",
      7     "<|user|>\n Write me a recommendation letter. \n <|assistant|>\n"]*20, return_tensors="pt", padding=True, truncati
----> 8 model_out = model(sentences["input_ids"].cuda())
      9 model_out

                                    11 frames

/usr/local/lib/python3.10/dist-packages/transformers/models/llama/modeling_llama.py in forward(self, hidden_states)
    105         variance = hidden_states.pow(2).mean(-1, keepdim=True)
    106         hidden_states = hidden_states * torch.rsqrt(variance + self.variance_epsilon)
--> 107         return self.weight * hidden_states.to(input_dtype)
    108
    109

OutOfMemoryError: CUDA out of memory. Tried to allocate 18.00 MiB. GPU 0 has a total capacity of 14.75 GiB of which 11.06 MiB i
use. Of the allocated memory 14.18 GiB is allocated by PyTorch, and 437.93 MiB is reserved by PyTorch but unallocated. If rese
setting max_split_size_mb to avoid fragmentation.  See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

Efficient training

Scaling Laws

- Scaling Laws for Neural Language Models (Kaplan et al. 2020)

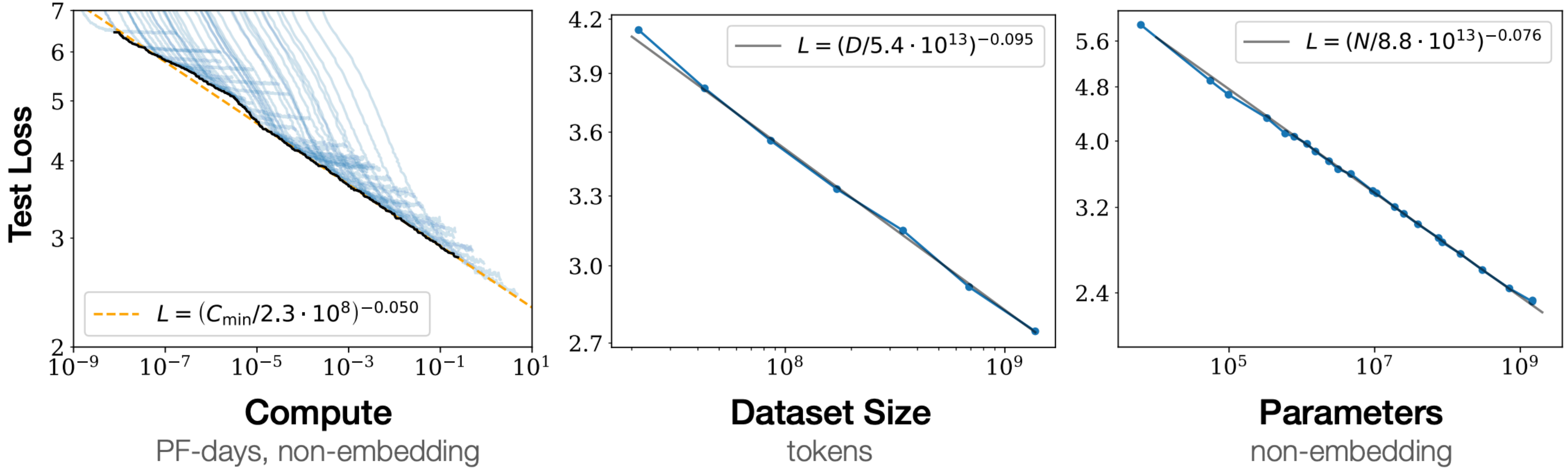
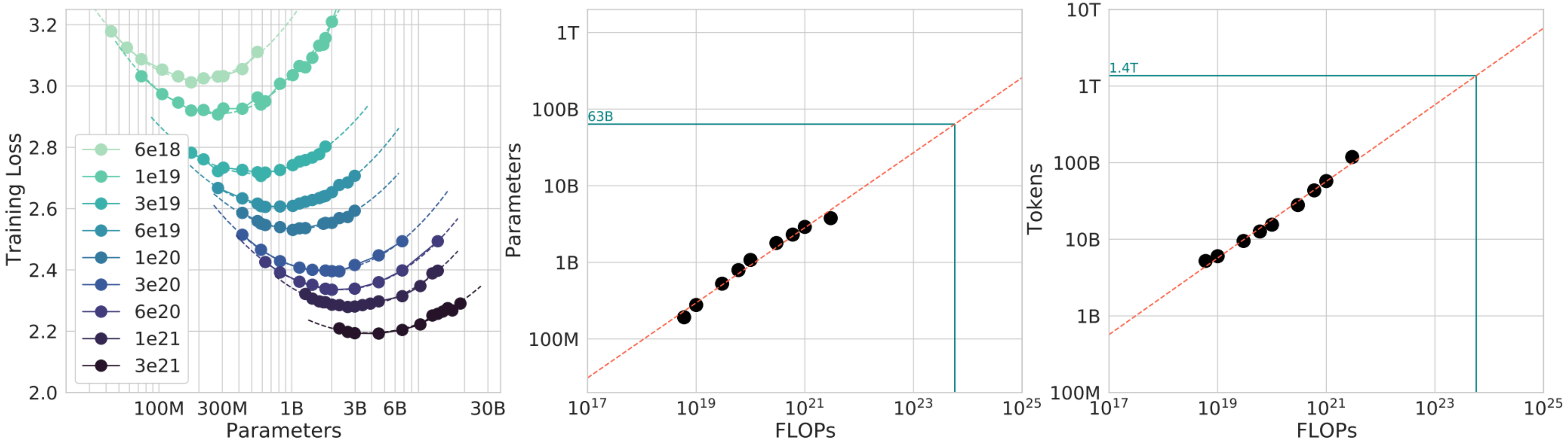


Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

Chinchilla Scaling Laws

- Refinement using more data points & better training recipe
- Given C FLOPS, what model size N and training tokens D should one use?



Chinchilla Scaling Laws

- Propose a form for the final loss:

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}$$

- Fit it on data points
  - E = 1.69 ("*entropy of natural language*")
  - A = 406.4, B = 410.7, α = 0.34, β = 0.28

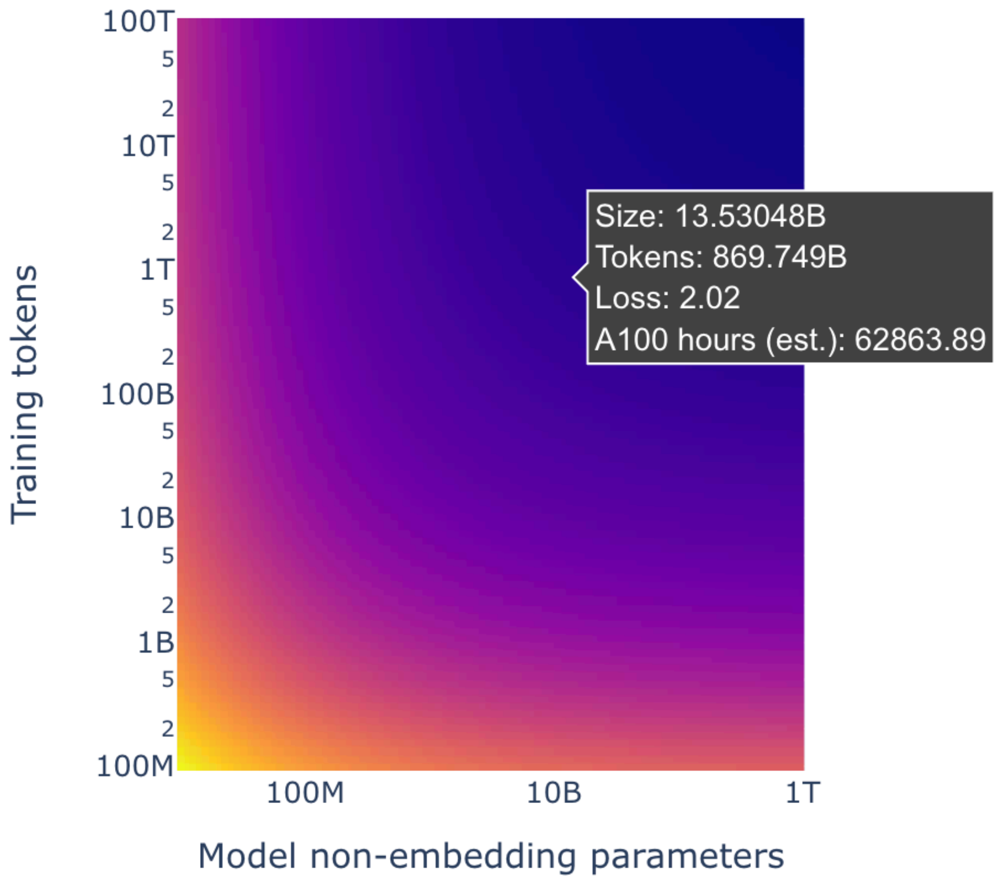
Chinchilla Scaling Laws

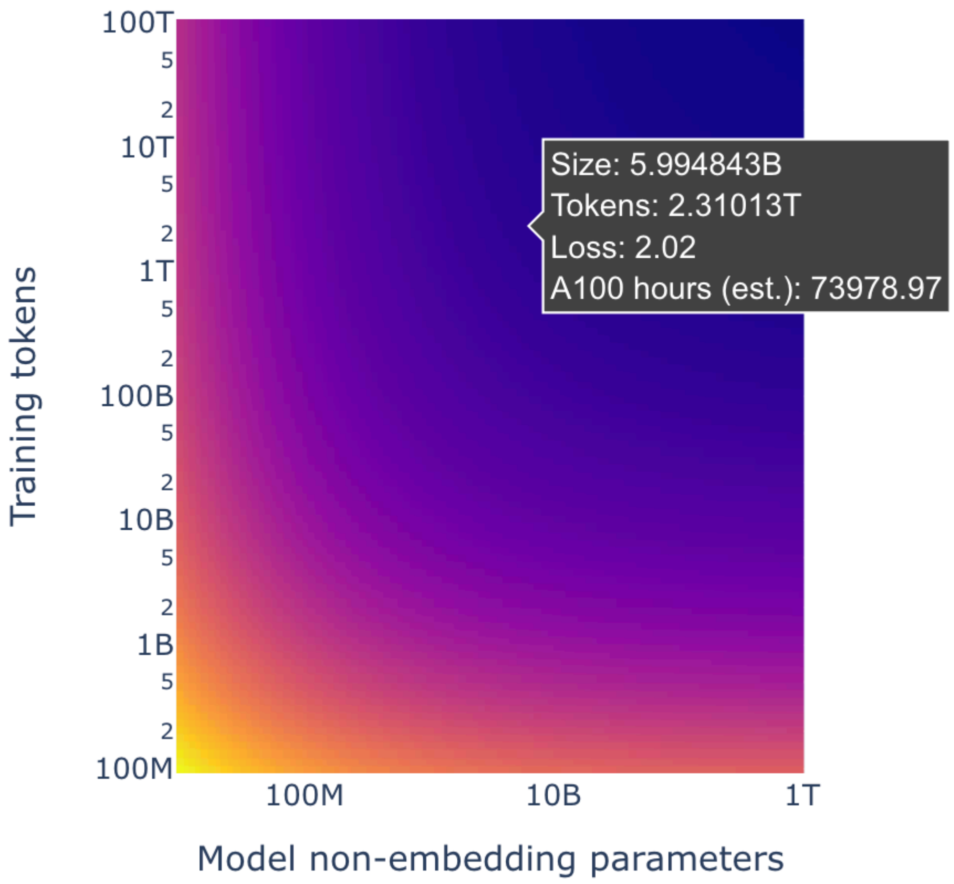
- Compute $C = O(ND)$ ($C \simeq 6ND$)
- For a given compute level C, there exist an optimal $N*$ and $D*$
  - Training a bigger model on less data => worse
  - Training a smaller model on more data => worse

Chinchilla Scaling Laws - In practice

- Train for longer than announced by laws
  - Why?
- Over-train smaller models
  - Trade training compute for inference compute
- Example: Mistral-7B model

Chinchilla Scaling Laws - In practice



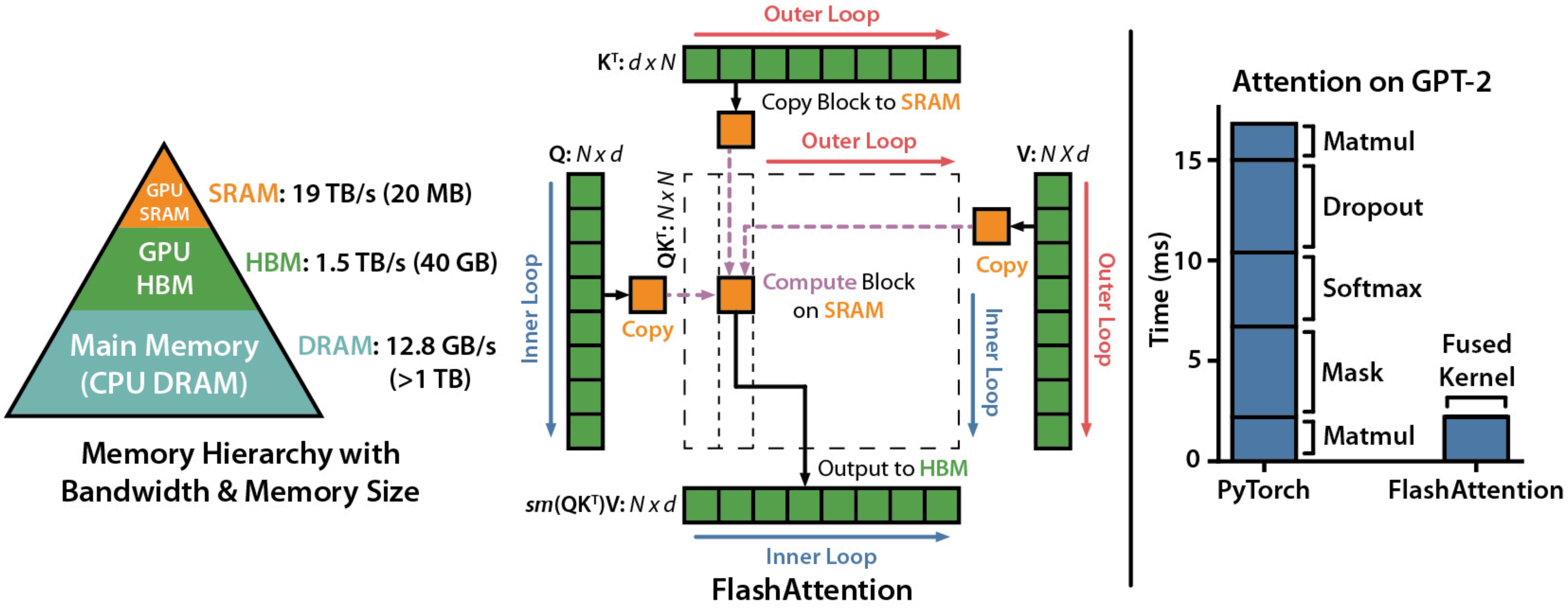


Training LMs

- Batch size matters:
  - No bigger than 1-2M tokens (Kaplan et al., 2020)
  - Maximize parallel computation
- Float precision:
  - `float16`: reduces memory usage, good with V100-gen GPUs
  - `bfloat16`: more stability, but only usable with A100-gen GPUs

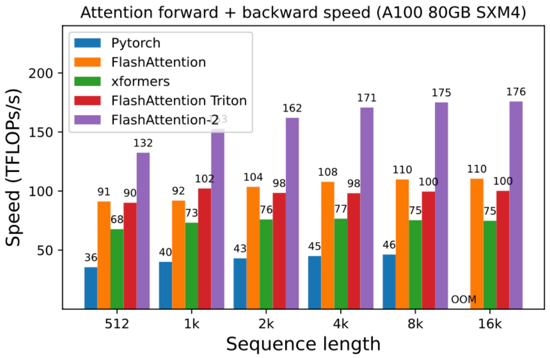
Training LMs - Efficient implementations

- FlashAttention (Dao et al. 2022)

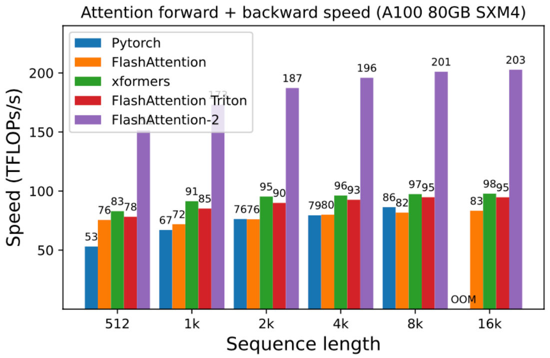


Training LMs - Efficient implementations

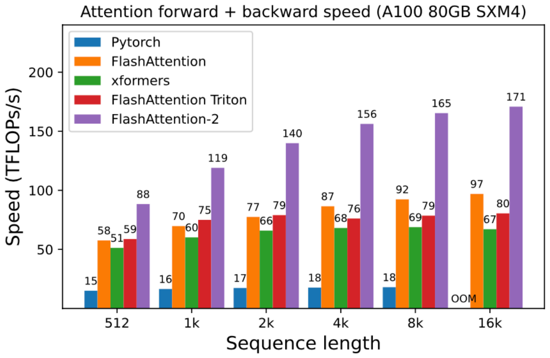
- FlashAttention2 (Dao et al. 2023)



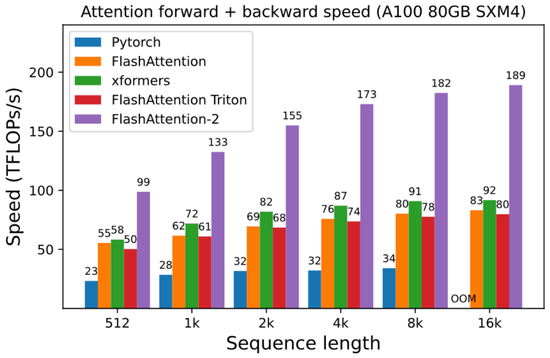
(a) Without causal mask, head dimension 64



(b) Without causal mask, head dimension 128



(c) With causal mask, head dimension 64



(d) With causal mask, head dimension 128

Training LMs - Efficient implementations

- xFormers & Memory-efficient attention (Rabe et al. 2021)
  - Classical implementation

$$s_i = \mathrm{dot}(q, k_i), \qquad s_i' = \frac{e^{s_i}}{\sum_j e^{s_j}}, \qquad \mathrm{attention}(q, k, v) = \sum_i v_i s_i'.$$

  - Memory-efficient implementation

$$s_i = \mathrm{dot}(q, k_i), \qquad s_i' = e^{s_i}, \qquad \mathrm{attention}(q, k, v) = \frac{\sum_i v_i s_i'}{\sum_j s_j'}.$$

- ~SOTA on V100-gen GPUs

Training LMs - Efficient variants

- Linear attention (e.g. Beltagy et al. 2020)

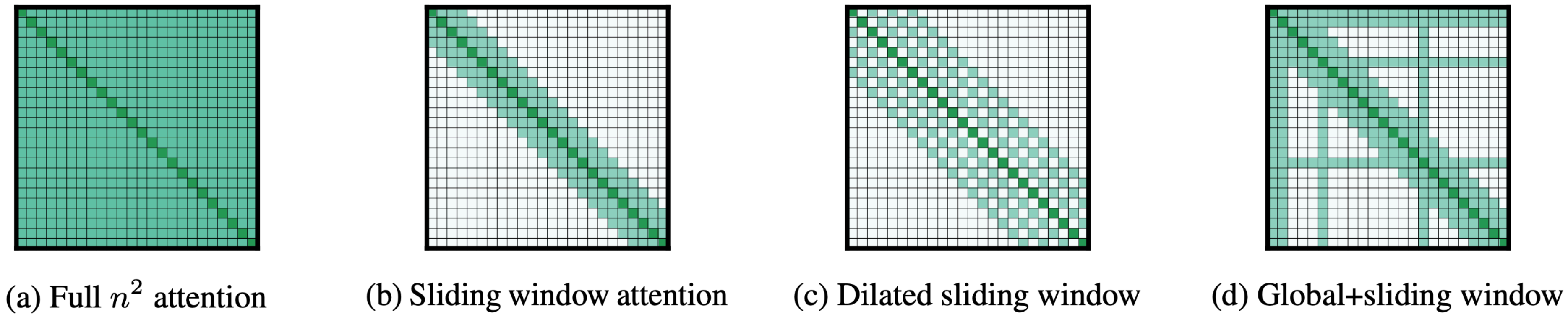


Figure 2: Comparing the full self-attention pattern and the configuration of attention patterns in our Longformer.

- Can be used to adapt model for efficient inference
- Used in Mistral-7B

Training LMs - Large-scale training

- Dream scenario:
  - Model fits on the GPU
  - Forward + backward fit with the `batch_size`
  - Optimization fits memory

Training LMs - Large-scale training

- Optimization OOM scenario
  - Model fits on the GPU
  - Forward + backward fit with the `batch_size`
  - Optimization saturates GPU

- Use memory-efficient optimizers
  - Adafactor: factor momentum matrix in Adam
  - CAME: regularize Adafactor
  - LION: only tracks momentum

Training LMs - Large-scale training

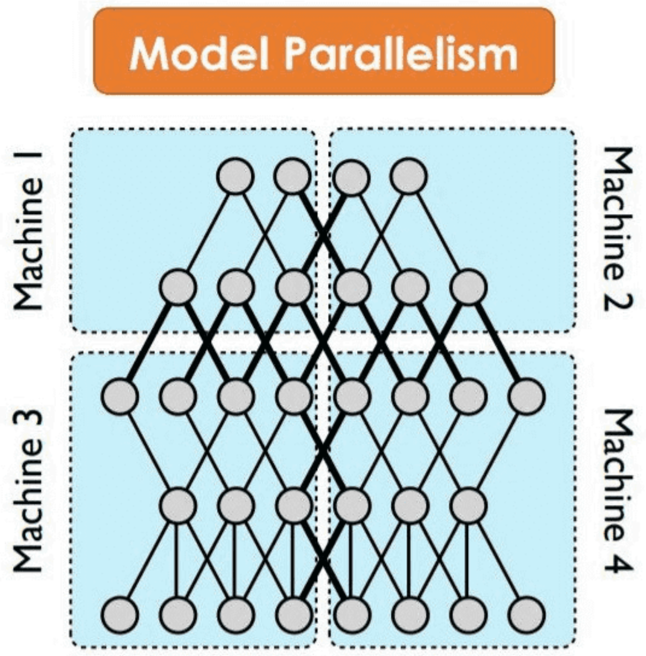
- Forward/backward OOM scenario
  - Model fits on the GPU
  - Forward + backward saturates with the `batch_size`
- Use gradient accumulation
  - Compute forwards with `micro_batch_size` $\ll$ `batch_size`
  - Sum `micro_batch_size//batch_size` gradients
  - Backward once

Training LMs - Multi-GPU training

- **Distirbuted Data Parallel (DDP)** with k GPUs
  - Copy the model on the k GPUs
  - Send a micro-batch to each GPU
  - Compute forward/backward in parallel on each GPU
  - *Send* gradients to one GPU & optimize
  - *Send* weight updates to each GPU

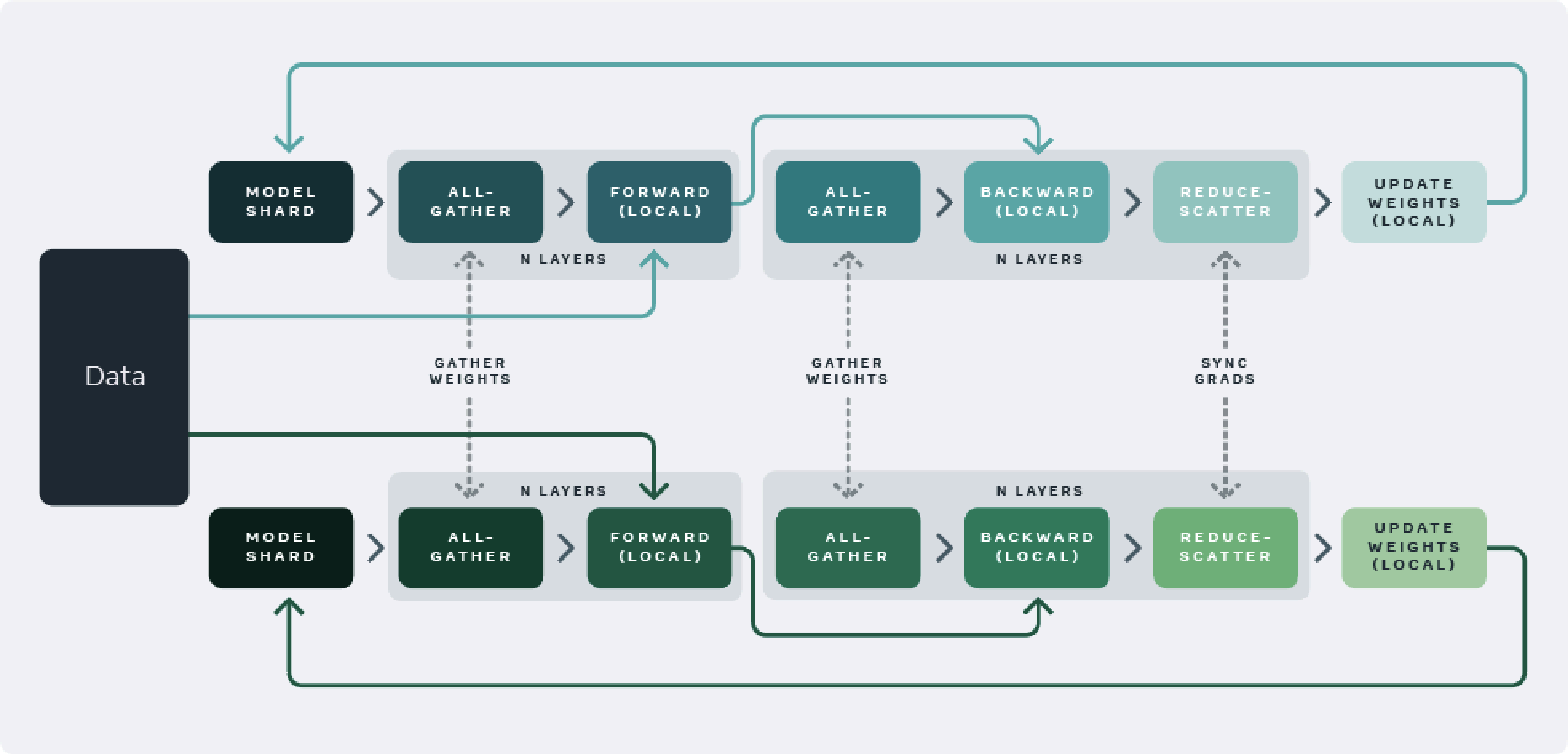
Training LMs - Multi-GPU training

- Model OOM scenario
  - Model does not fit on one GPU (e.g. `micro_batch_size=1` fails)
- Model parallelism



Training LMs - FSDP

Fully sharded data parallel training

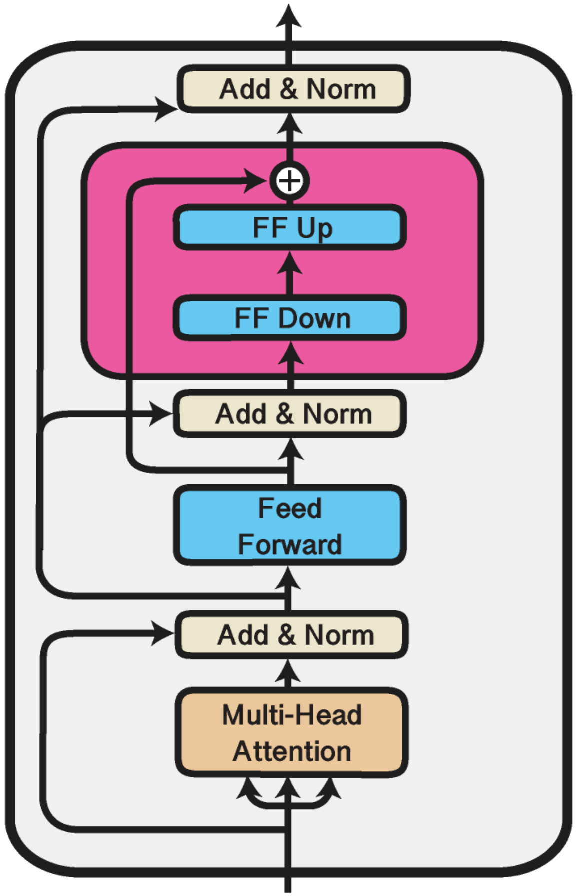


Training LMs - DeepSpeed

- Similar to FSDP:
  - Shares model weights...
  - but also optimizer states...
  - and gradients
- For relevant sizes: not that different in speed ⌛

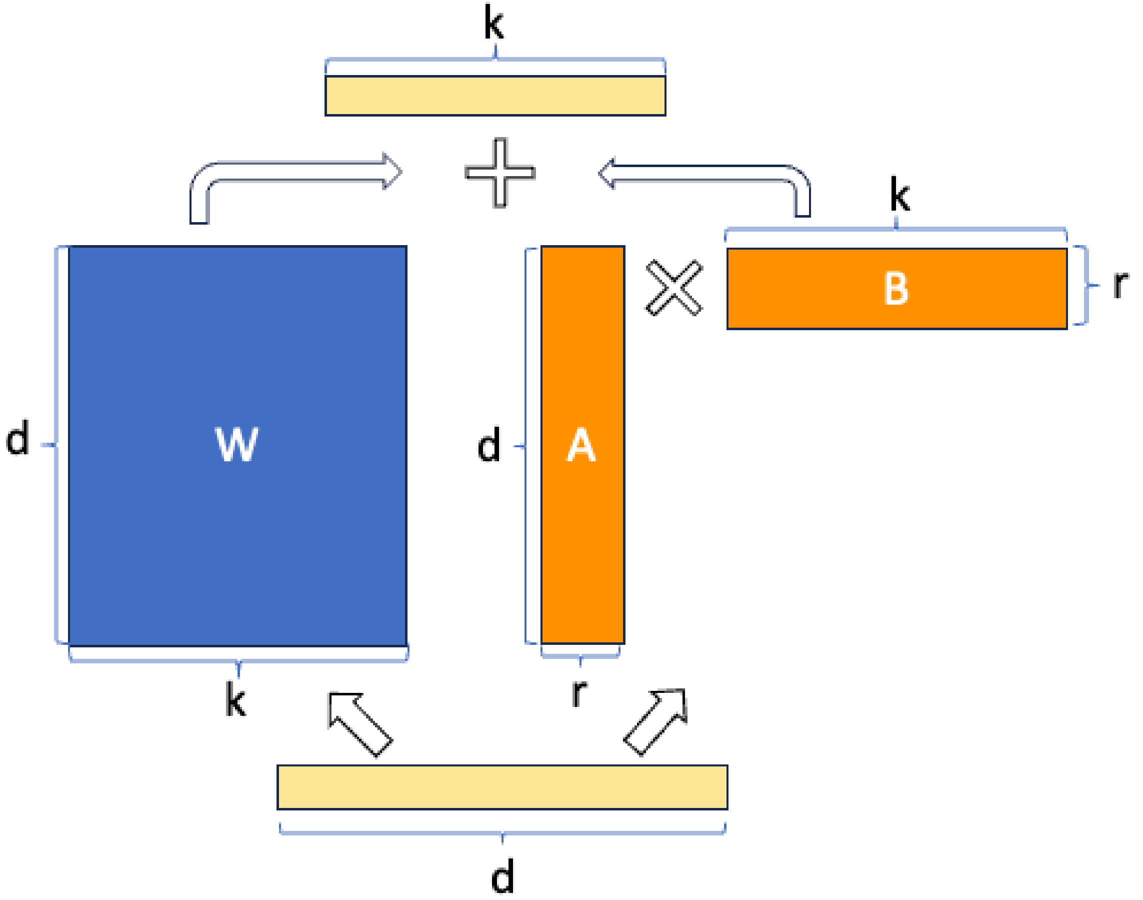
Adapting LMs - Adapters

- Parameter-Efficient Transfer Learning for NLP (Houlsby et al. '19)



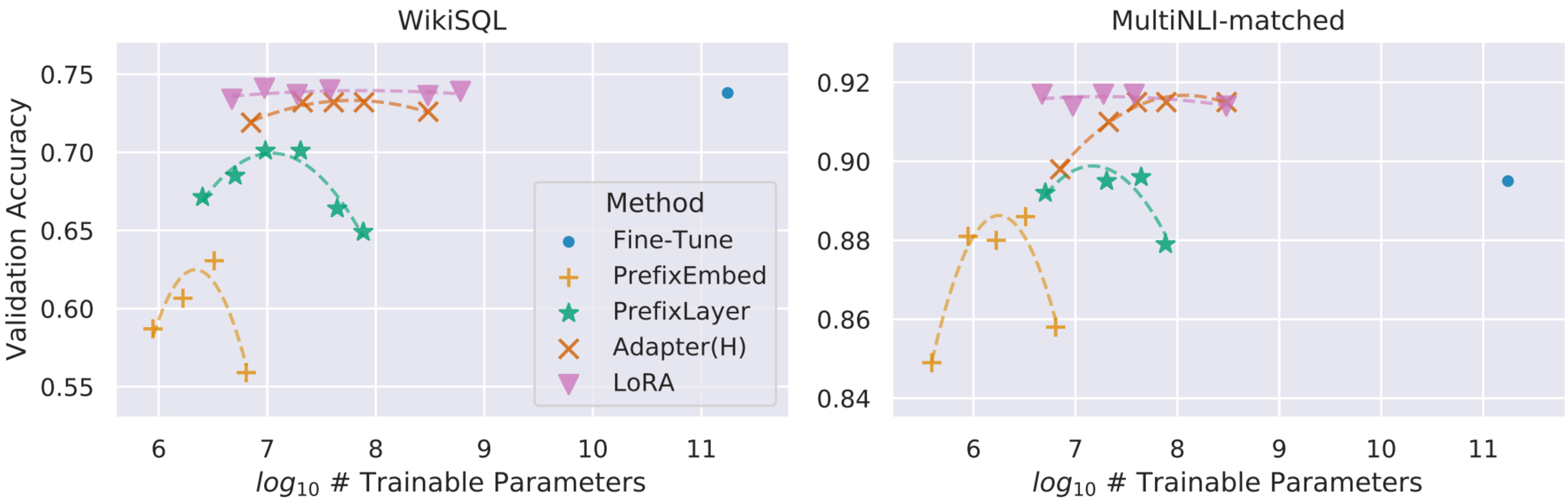
Adapting LMs - LoRA

- Low-Rank Adaptation of Large Language Models (Hu et al. '21)



Adapting LMs - LoRA vs Adapters

- Better + more stable results across hyper-parameters



Efficient inference

Previous methods hold

- Efficient attention implementations & variants
  - FlashAttention / xFormers
  - Linear attention
- Model parallelism (FSDP & DeepSpeed)
- LORA weights for fast model "switching"
  - Keep big model in memory
  - Load task-specific LoRA when required

Quantization

- Changes the data type of a model (e.g. `float32 -> int4`)
- Models are usually trained in `float16` or `bfloat16` for stability
- Needs rescaling:

$$Q_{i_4}(0.3) \neq 0$$

LM quantization

- GPTQ (Frantar et al. 2023)

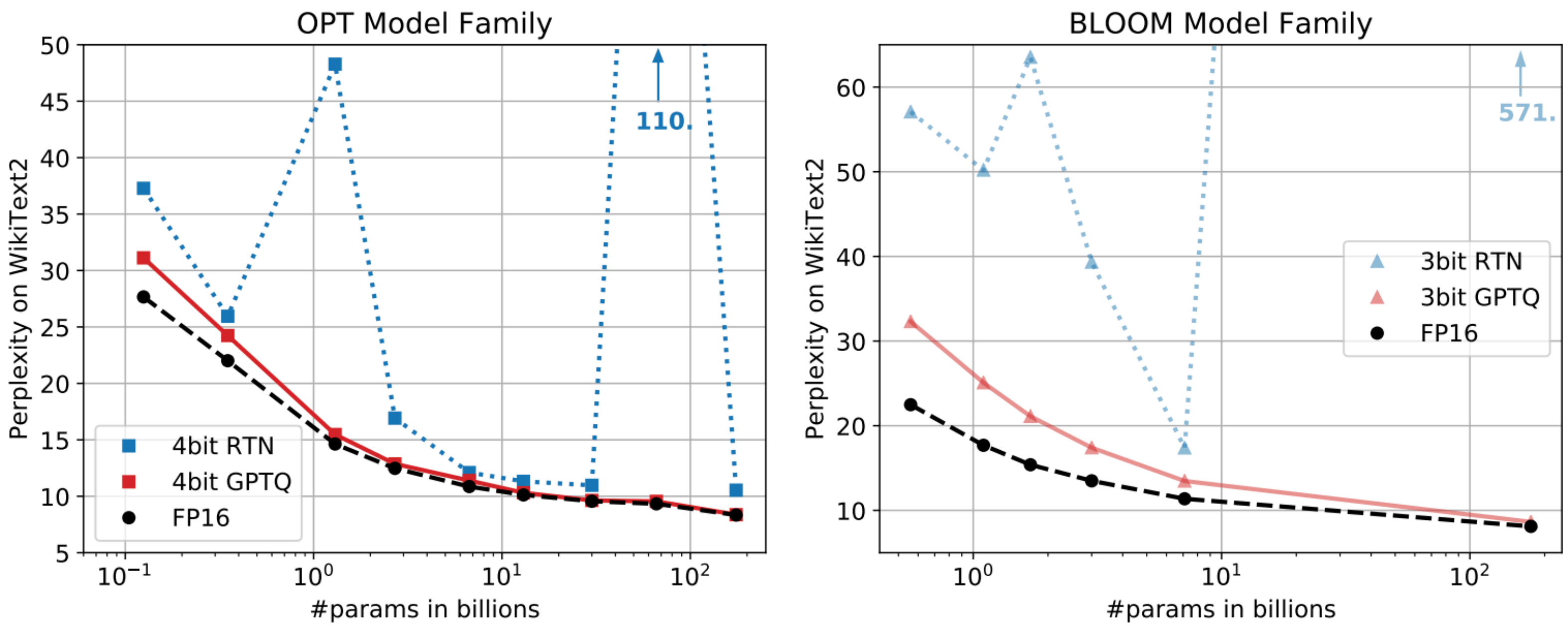


Figure 1: Quantizing OPT models to 4 and BLOOM models to 3 bit precision, comparing GPTQ with the FP16 baseline and round-to-nearest (RTN) (Yao et al., 2022; Dettmers et al., 2022).

LM quantization - GPTQ

Consider quantization as an optimization problem:

$$\argmin_{\hat{W}} ||WX - \hat{W}X||_2^2$$

where W is a weight matrix to quantize into $\hat{W}$, and X are data points (e.g. token sequences)

LM quantization - GPTQ

- For each row, quantize some W_{ij} by solving the quadratic problem and adjust the non-quantized coefficients of W_i to minimize impact
- *Empirical*: update order does not matter
- Update at smaller scale and batch whole matrix updates
- Precompute Hessian (needed for adaptation) on non-quantized coefficients since they can be taken left-to-right

LM quantization - GPTQ

- A matter of minutes/hours (on a single A100 GPU)

OPT	13B	30B	66B	175B
Runtime	20.9m	44.9m	1.6h	4.2h
BLOOM	1.7B	3B	7.1B	176B
Runtime	2.9m	5.2m	10.0m	3.8h

Table 2: GPTQ runtime for full quantization of the 4 largest OPT and BLOOM models.

LM quantization - GPTQ

- Inference speed/memory is greatly increased:

GPU	FP16	3bit	Speedup	GPU reduction
A6000 – 48GB	589ms	130ms	4.53×	$8 \to 2$
A100 – 80GB	230ms	71ms	3.24×	$5 \to 1$

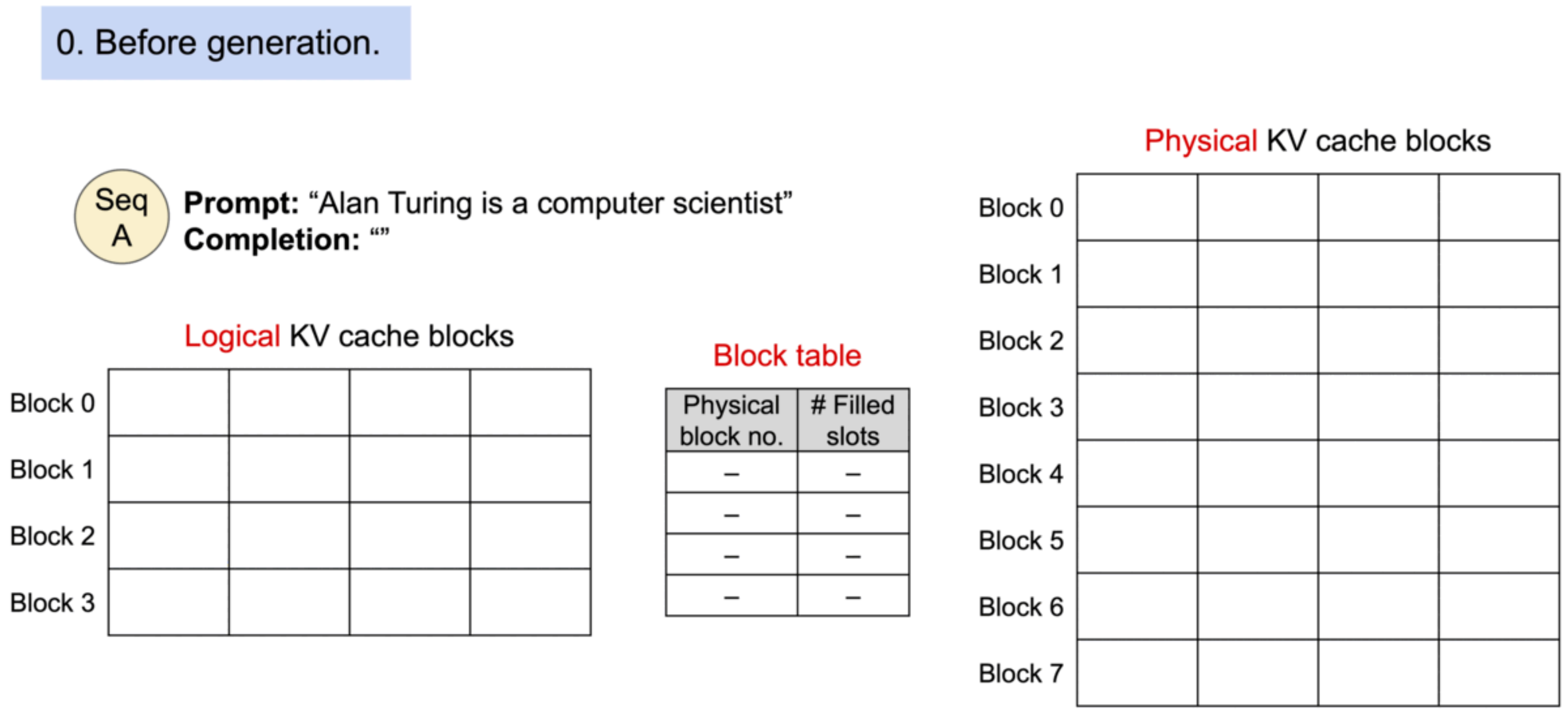
Table 6: Average per-token latency (batch size 1) when generating sequences of length 128.

- While performance is maintained (OPT perplexity ⬇️)

OPT	Bits	125M	350M	1.3B	2.7B	6.7B	13B	30B	66B	175B
full	16	27.65	22.00	14.63	12.47	10.86	10.13	9.56	9.34	8.34
RTN	4	37.28	25.94	48.17	16.92	12.10	11.32	10.98	110	10.54
GPTQ	4	**31.12**	**24.24**	**15.47**	**12.87**	**11.39**	**10.31**	**9.63**	**9.55**	**8.37**

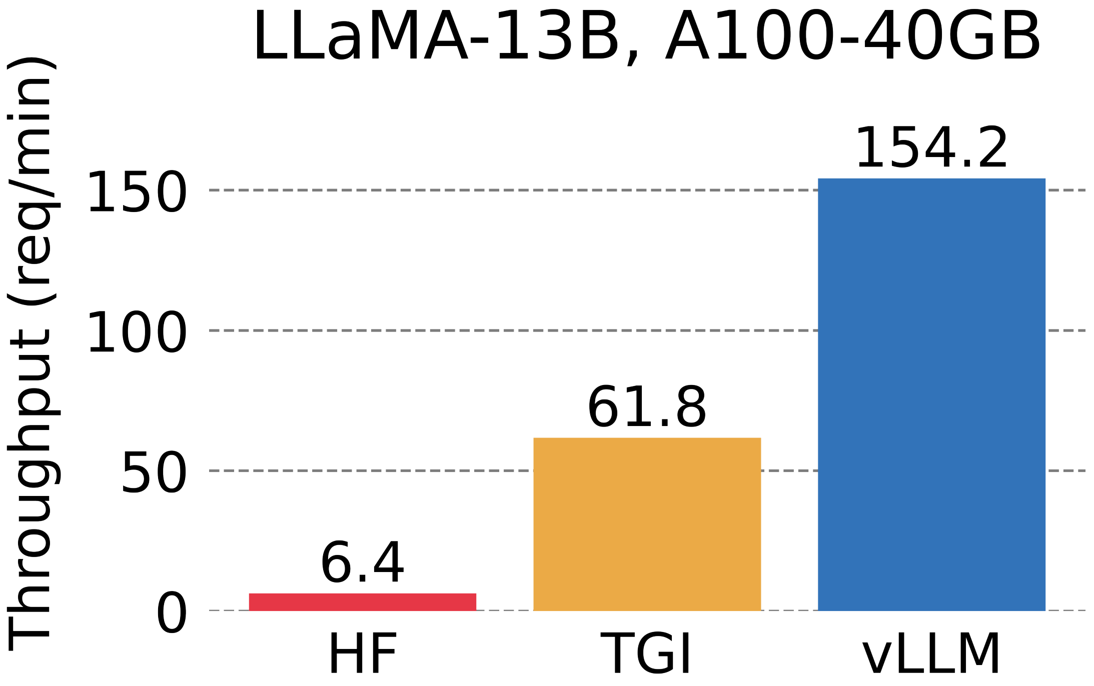
Managing KV cache - vLLM

- Paged Attention (Kwon et al. 2023)



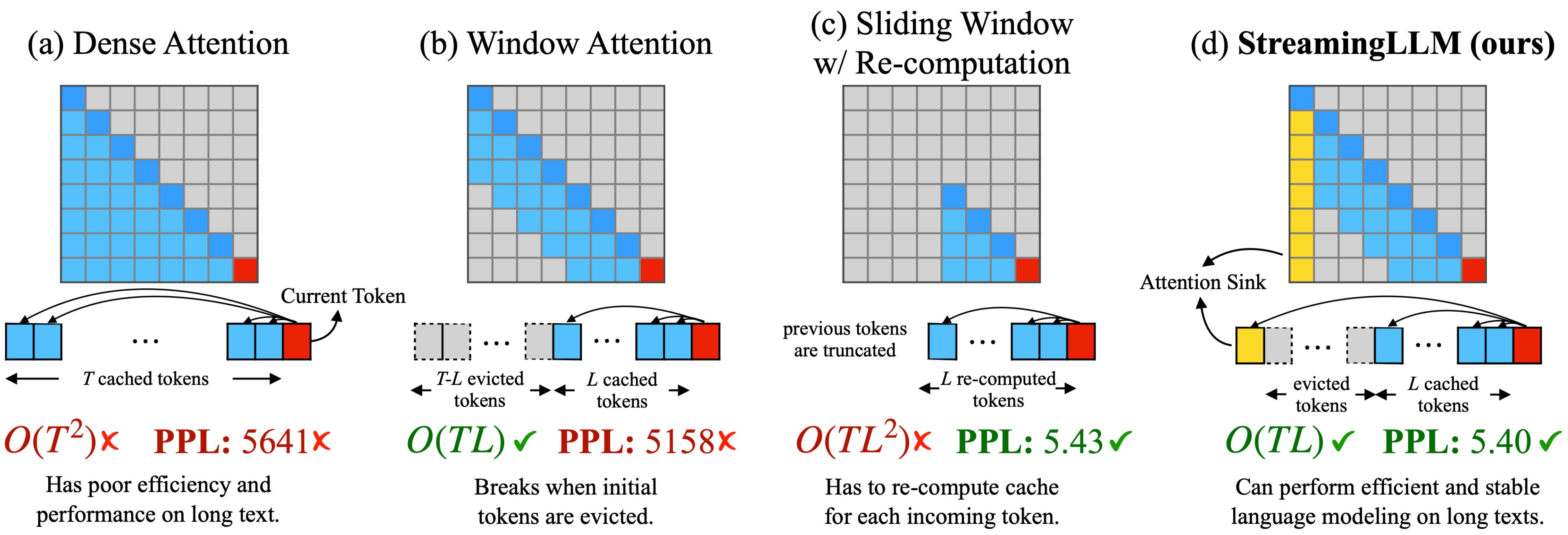
Managing KV cache - vLLM

- Better throughput + parallelization across requests



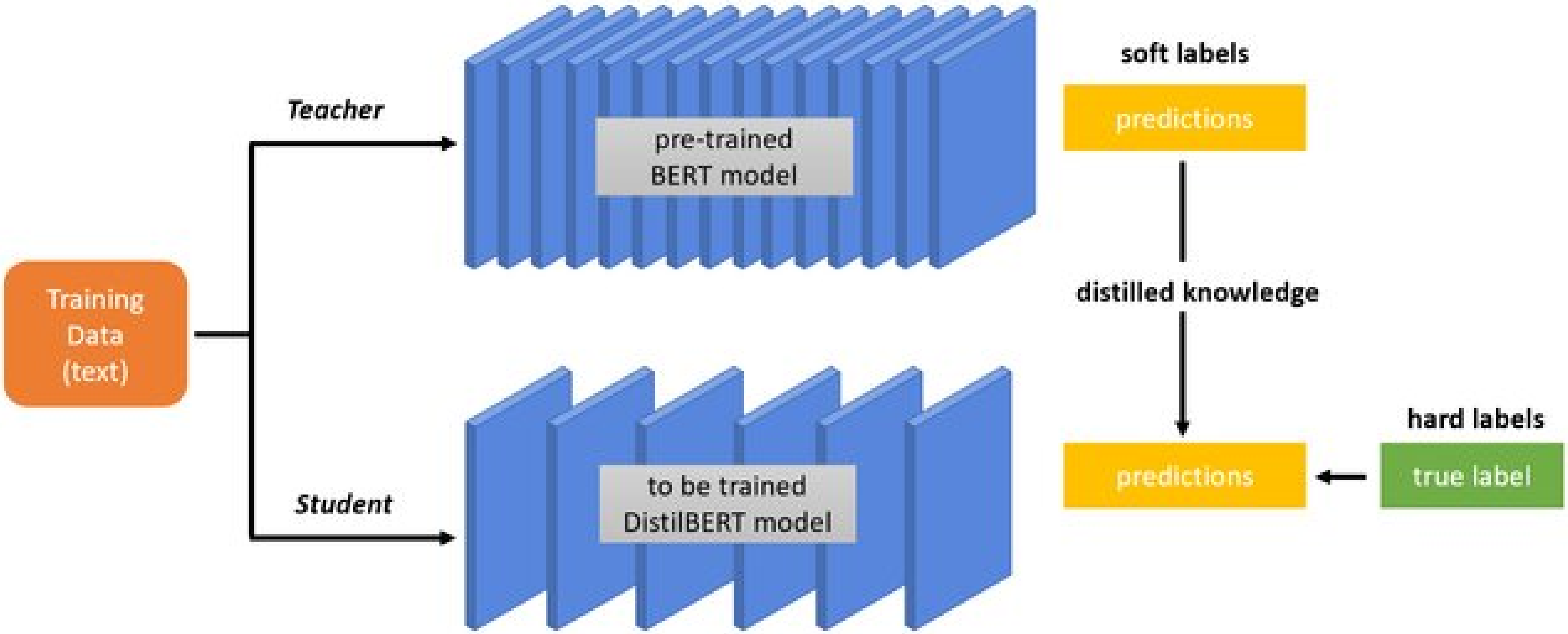
Long KV cache - StreamingLLM

- Efficient Streaming Language Models with Attention Sinks (Xiao et al. 2023)



Model reduction

DistilBERT (Sanh et al. 2019)

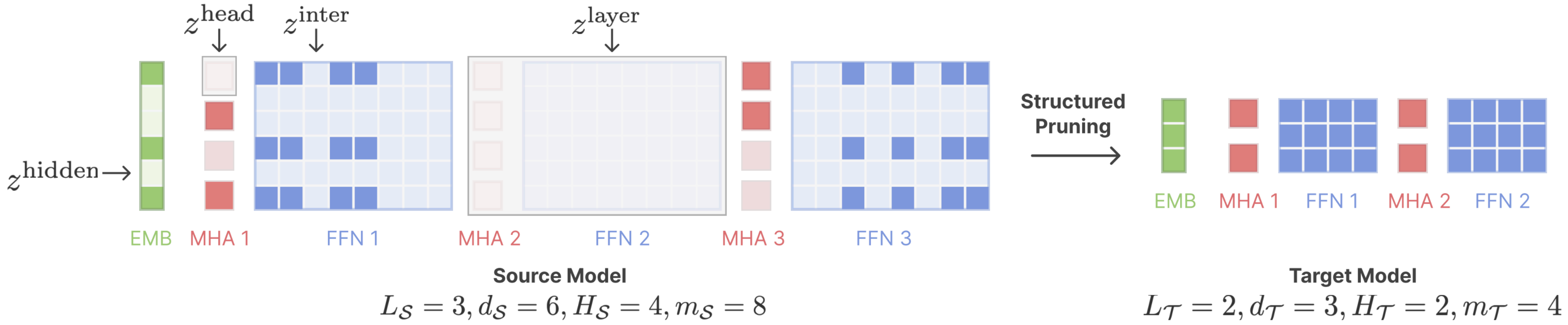


DistilBERT (Sanh et al. 2019)

- Can be expensive if teacher is big
- Still loses performance

Sheared Llama (Xia et al. 2023)

- Remove weights that minimize loss increase



- Continue the pretraining of the obtained reduced model

Sheared Llama (Xia et al. 2023)

- Get a good model with much less data/compute