LLMs: Parameters, Data, Hardware, Scaling

Learning goals

- Learn to calculate Transformer number of parameters
- Understand Transformer computation and memory load
- Learn about Flash Attentions
- Understand Scaling Laws and Chinchilla

LLM PARAMETERS: MAIN COMPONENTS

▸ Source: Kipply's Blog, 2023

- Model parameters are half-precision (bf16 = bfloat16) numbers, 2 bytes each
- One block (decoder unit) consists of:
  - W_q, W_k, W_v matrices which are each $n_{heads} \cdot d_{model} \cdot d_{head}$ and project the input into the queries, keys, and values used in self-attention.
  - A W_0 matrix which is also $n_{heads} \cdot d_{head} \cdot d_{model}$ for the output of self-attention, before the MLP (feedforward) layer
  - MLP weights, which are two matrices each of ${d_{model}}^2 \cdot 4$. Here "4" means that the MLP is 4 times the size of the model embedding dimension.
- In most architectures, $d_{model} = n_{heads} \cdot d_{head}$

LLM PARAMETERS: FORMULA

Combining the above, for one layer/block we get this formula:

$$P_{layer} = 3 \cdot d_{model} \cdot n_{heads} \cdot d_{head} + d_{model} \cdot n_{heads} \cdot d_{head} + 2 \cdot 4 \cdot {d_{model}}^2$$

$$= 4 \cdot d_{model} \cdot n_{heads} \cdot d_{head} + 8 \cdot {d_{model}}^2$$

$$= 4 \cdot d_{model} \cdot d_{model} + 8 \cdot {d_{model}}^2$$

$$= 12 \cdot {d_{model}}^2$$

For a LLM of n layers, we get:

$$\text{total \# parameters } P = 12 \cdot n_{layers} \cdot {d_{model}}^2$$

LLM PARAMETERS: GPT

GPT-3 Small has:

$$n_{params} = 125\,M\ ;\ n_{layers} = 12\ ;\ d_{model} = 768\ ;\ n_{heads} = 12\ ;\ d_{head} = 64$$

GPT-3 Medium has:

$$n_{params} = 350\,M\ ;\ n_{layers} = 24\ ;\ d_{model} = 1024\ ;\ n_{heads} = 16\ ;\ d_{head} = 64$$

Applying the above formula we get $\sim$85 M parameters for GPT-3 Small and $\sim$302 M parameters for GPT-3 Medium.

LLM PARAMETERS: GPT

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

- Applying the above formula we get $\sim$85 M parameters for GPT-3 Small and $\sim$302 M parameters for GPT-3 Medium.
- What are we missing?

LLM PARAMETERS: OTHER COMPONENTS

Numeric illustration as in BERT-base

- Word Embedding parameters – 30522 x 768 = 23,440,896
- Position Embedding parameters – 512 x 768 = 393,216
- Token Type Embedding parameters – 2 x 768 = 1536
- Embedding Layer Normalization, weight and Bias – 768 + 768 = 1536
- There can be other model-specific parameters...

Total additional parameters = 23,837,184

They do not scale with model size.

COMPUTE REQUIREMENTS

Basic equation: Cost to train a transformer model:

$$C \approx \tau T = 6PD$$

▸ Source: Quentin et al., 2023

COMPUTE REQUIREMENTS: C = TAU T = 6 P D

where:

- C: No. of floating-point operations (FLOPs) to train the model: $C = C_{forward} + C_{backward}$
- $C_{forward} \approx 2PD$
- $C_{backward} \approx 4PD$
- $2PD$: "2" comes from the multiply-accumulate operation used in matrix multiplication
- $4PD$: backward pass approximately twice the compute of the forward pass
- In the backward pass at each layer, gradients have to be calculated for the weights at that layer and for the previous layer's output.
- So that the gradient of the previous layer's weights can be calculated.
- τ is throughput of hardware: (No. GPUs) x (FLOPs/GPU)

COMPUTE REQUIREMENTS: C = TAU T = 6 P D

- T is the time spent training the model, in seconds
- P is the number of parameters in the model
- D is the dataset size (in tokens)

COMPUTE UNITS

C can be measured in different units:

- FLOP-seconds which is [Floating Point Ops / Second]
  - We also use multiples GFLOP-seconds, TFLOP-seconds etc.
  - Other multiples like PFLOP-days are used in papers
  - 1 PFLOP-day = $10^{15} \cdot 24 \cdot 3600$ FLOP-seconds
  - Actual FLOPs are always lower than the advertised theoretical FLOPs
- GPU-hours
  - GPU model is also required since they have different compute capacities

PARAMETER VS DATASET

- Model performance depends on number of parameters P, but also on number of training tokens D
- **We need to decide about P and D, so that we get the best performance within the compute budget.**
- Recommended tradeoff between P and D is: $D = 20P$
  - This is usually true for Chinchilla models ▸ Hoffmann et al., 2022, but not for all LLMs
- Training a LLM for less than 200 billion tokens is not recommended
- Rule of thumb: First determine the upmost inference cost, and then train the biggest model within that boundary.
- Different ways to determine P: based on available data, compute budget or inference time

MEMORY REQUIREMENTS

Common questions:

- How big is this model in bytes?
- Will it fit/train in my GPUs?

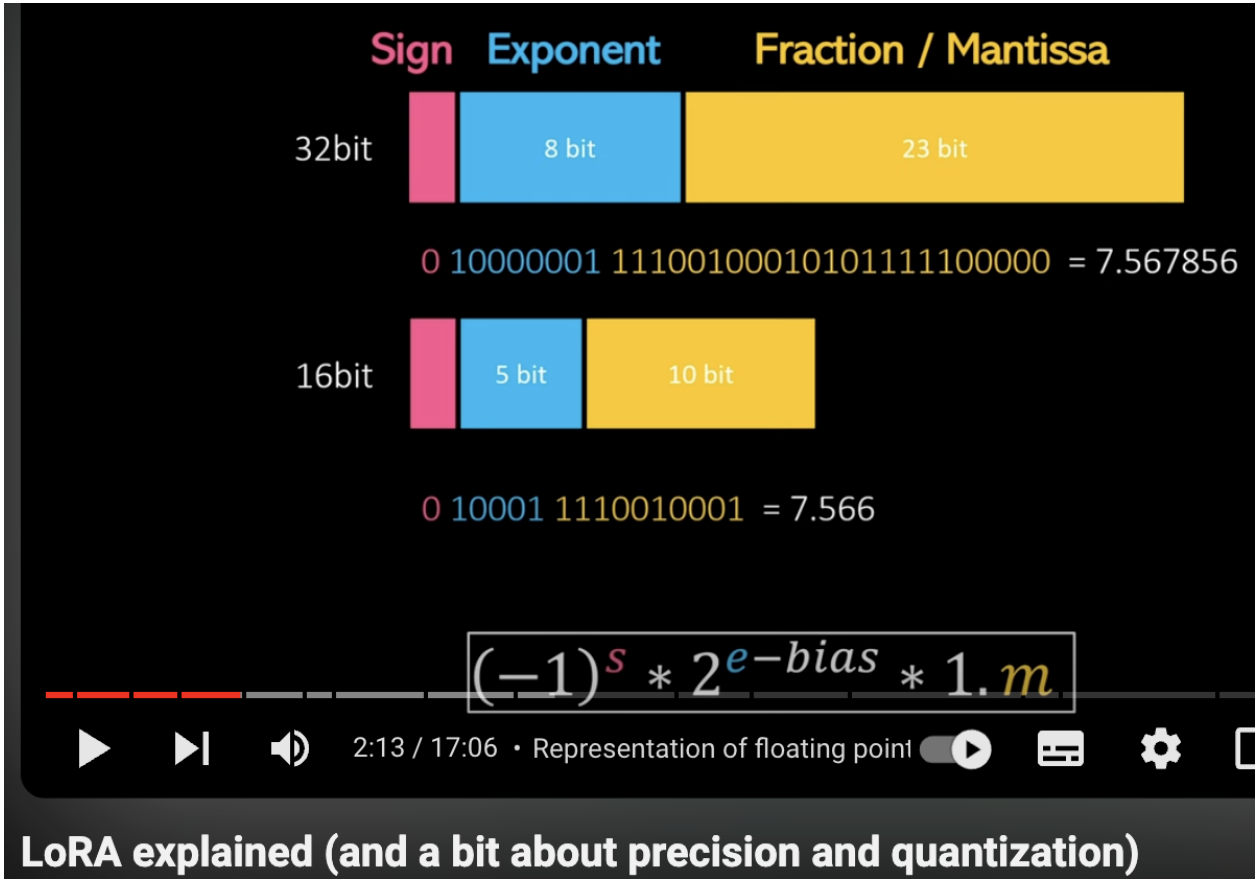
Model size components:

- Model parameters
- Optimizer states
- Gradients
- Activations

NUMBER REPRESENTATIONS

- Pure fp32: single precision floating point number as defined by ▸ IEEE 754 standard, takes 32 bits or 4 bytes
- fp16: half precision float number as defined by ▸ IEEE_754-2008, occupying 16 bits or 2 bytes
- bf16 or brain floating point 16, developed by Google Brain project, occupying 16 bits or 2 bytes
  - gives more bits to exponent (vs. significand/mantissa)
- int8: integer from -128 to 127, occupying 8 bits or 1 byte

FP32 / FP16

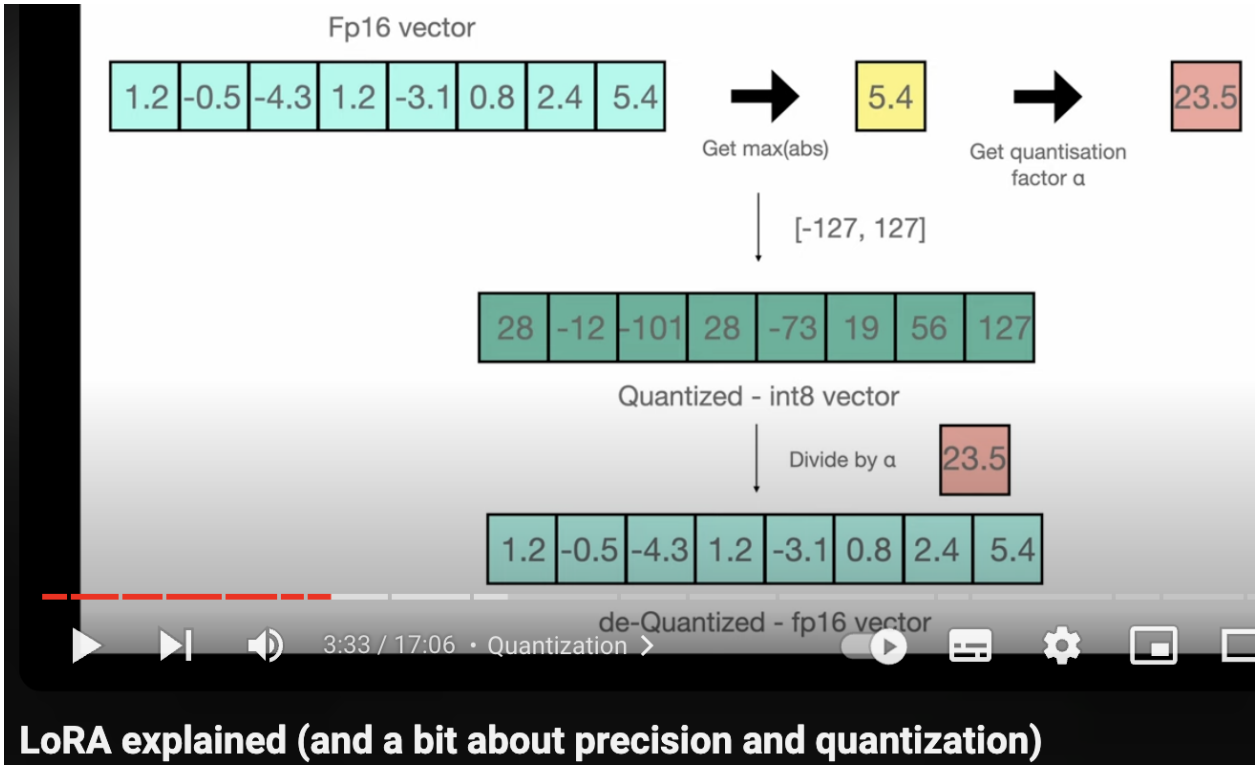

Sign
Exponent
Fraction / Mantissa
32bit
8 bit
23 bit
0 10000001 11100100010101111100000 = 7.567856
16bit
5 bit
10 bit
0 10001 1110010001 = 7.566
$(-1)^s * 2^{e-bias} * 1.m$
2:13 / 17:06 • Representation of floating point
LoRA explained (and a bit about precision and quantization)

INT8 QUANTIZATION

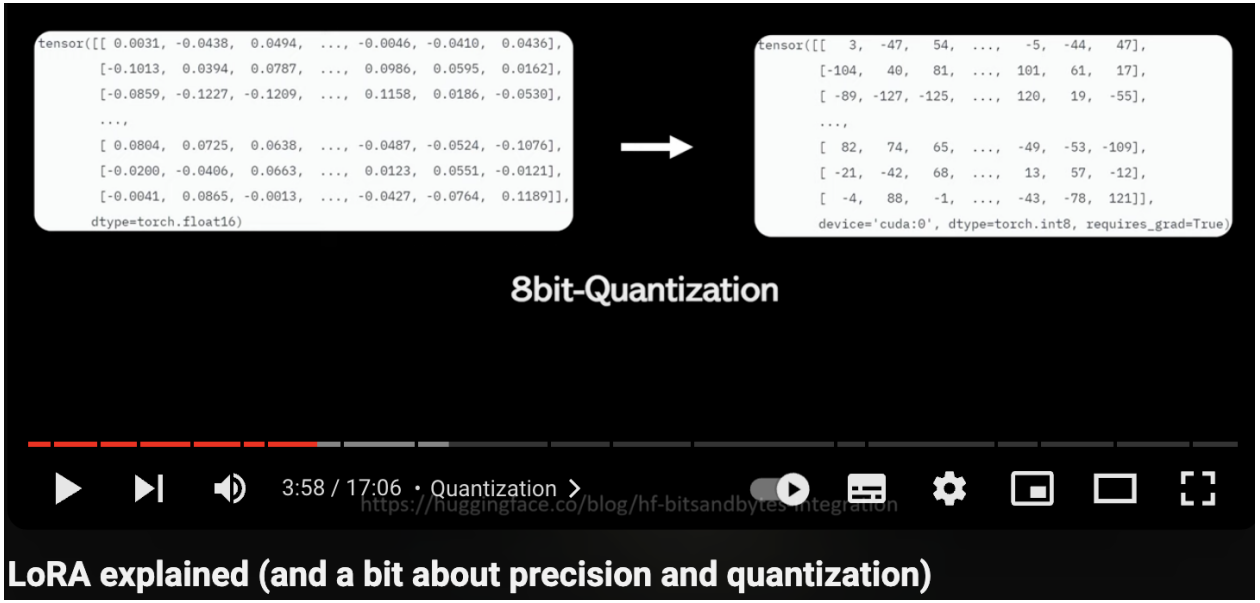
▸ mathworks

- Real_number = stored_integer * scaling_factor

INT* QUANTIZATION



INT* QUANTIZATION



MODEL PARAMETERS

Parameter size depends on chosen representation:

- Pure fp32: $Mem_{model} = 4\ bytes/param \cdot N_{params}$
- fp16 or bf16: $Mem_{model} = 2\ bytes/param \cdot N_{params}$
- int8: $Mem_{model} = 1\ byte/param \cdot N_{params}$

It is practically common to use mixed representations:

- fp32 + fp16
- fp32 + bf16

OPTIMIZER STATES

AdamW: $Mem_{AdamW} = 12\ bytes/param \cdot N_{params}$

- fp32 copy of parameters: 4 bytes/param
- Momentum: 4 bytes/param
- Variance: 4 bytes/param

bitsandbytes: $Mem_{AdamW} = 6\ bytes/param \cdot N_{params}$

- fp32 copy of parameters: 4 bytes/param
- Momentum: 1 byte/param
- Variance: 1 byte/param

GRADIENTS

They are usually stored in the same datatype as the model parameters.

Their memory overhead contribution is:

- fp32: $Mem_{grad} = 4\ bytes/param \cdot N_{params}$
- fp16 or bf16: $Mem_{grad} = 2\ bytes/param \cdot N_{params}$
- int8: $Mem_{grad} = 1\ byte/param \cdot N_{params}$

ACTIVATIONS

- GPUs are bottlenecked by memory, not FLOPs
- Save GPU memory by recomputing activations of certain layers
- Various schemes for selecting which layers to clear
- They take some extra memory, but save even more

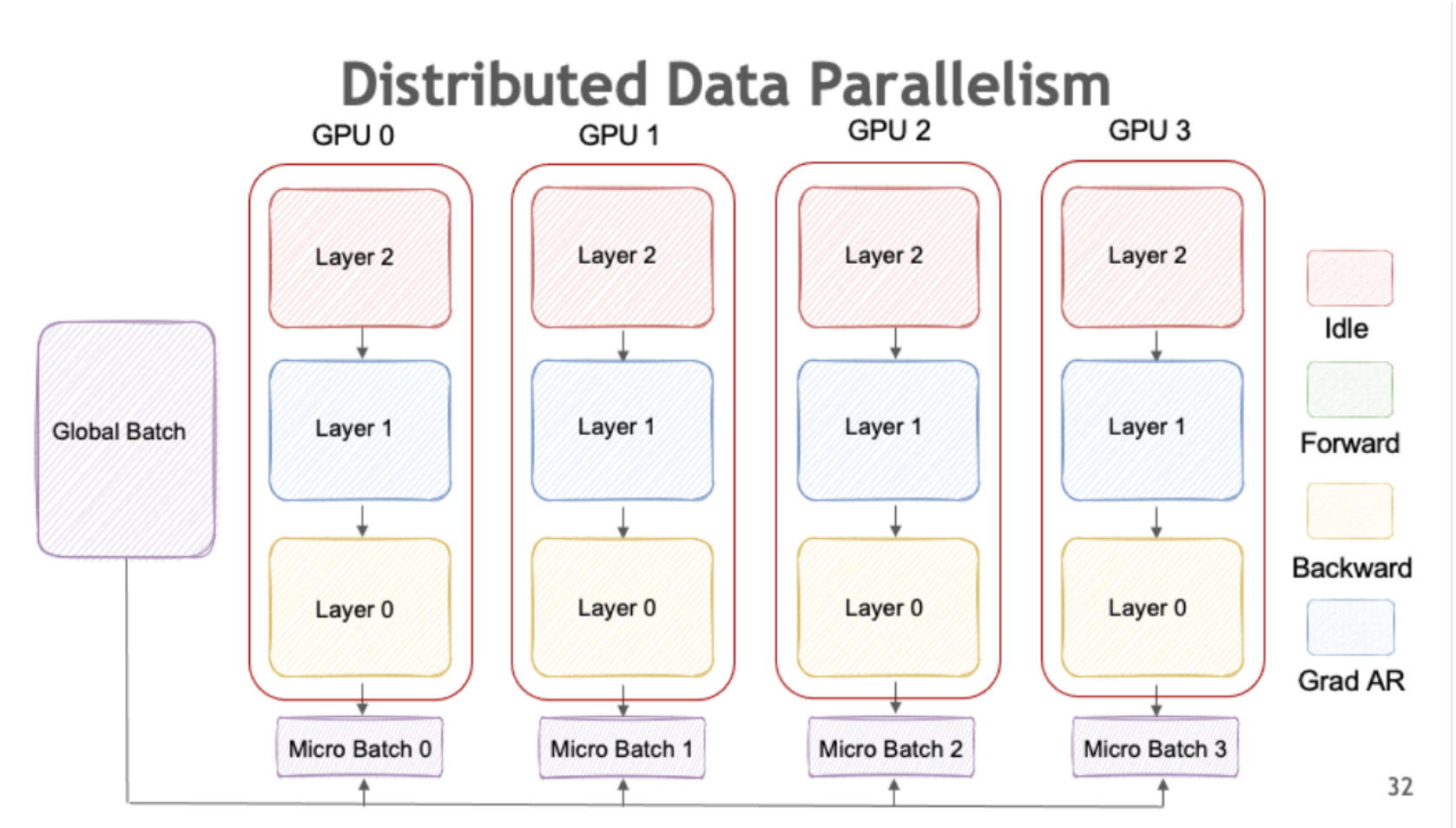
Total memory when training **using** activations:
$Mem_{training} = Mem_{params} + Mem_{opt} + Mem_{grad} + Mem_{activ}$

Total memory when training **without** activations:
$Mem_{training} = Mem_{params} + Mem_{opt} + Mem_{grad}$

DISTRIBUTED TRAINING

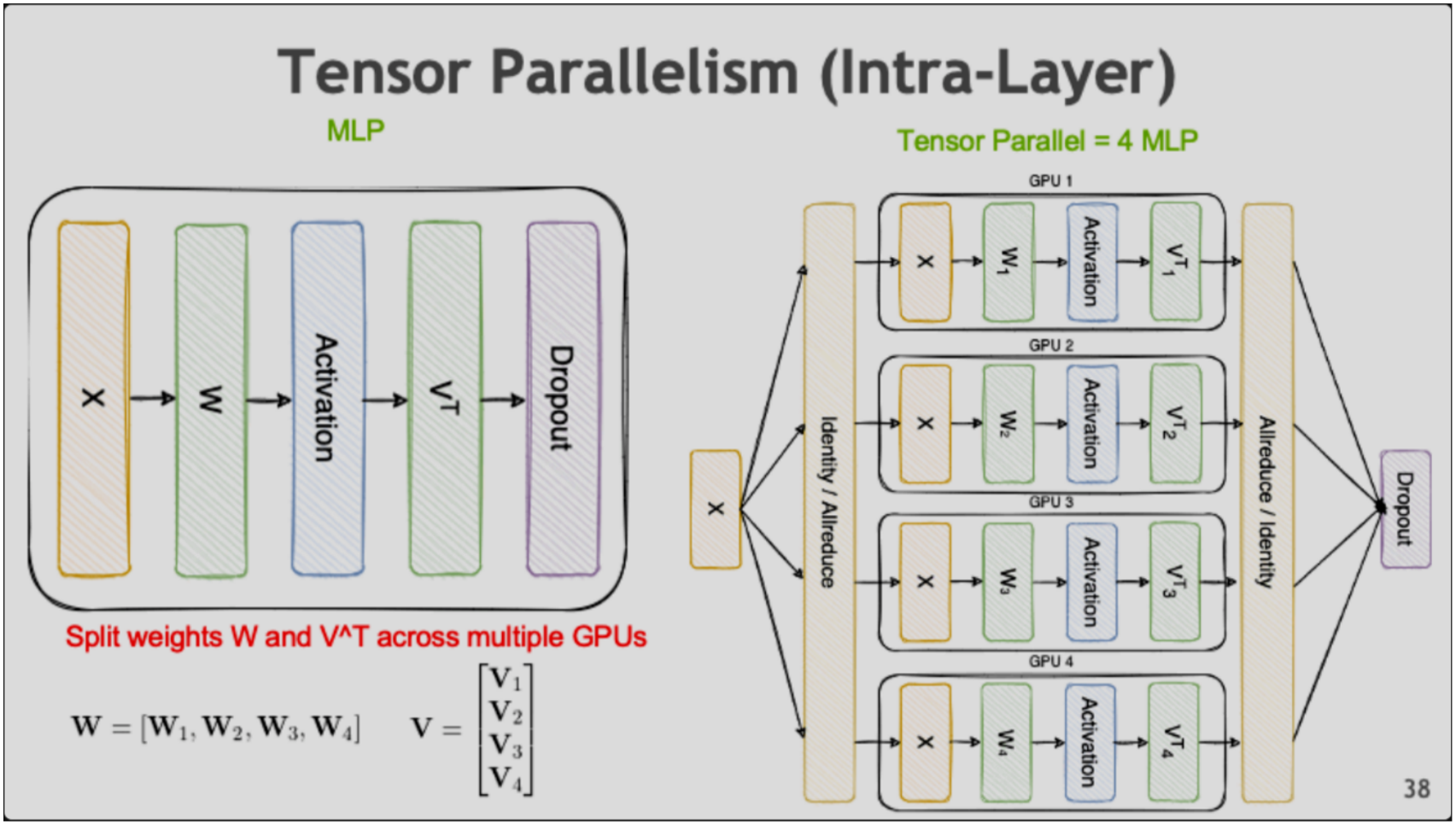
- Avoiding OOM issues
- **Data parallelism:** split the data on different model replicas
- **Tensor parallellism:** split model parameters across GPUs
- Training LLMs faster on many GPUs

DATA PARALLELISM (ANIMATED GIF!)



Source: Nvidia

TENSOR PARALLELISM (ANIMATED GIF!)



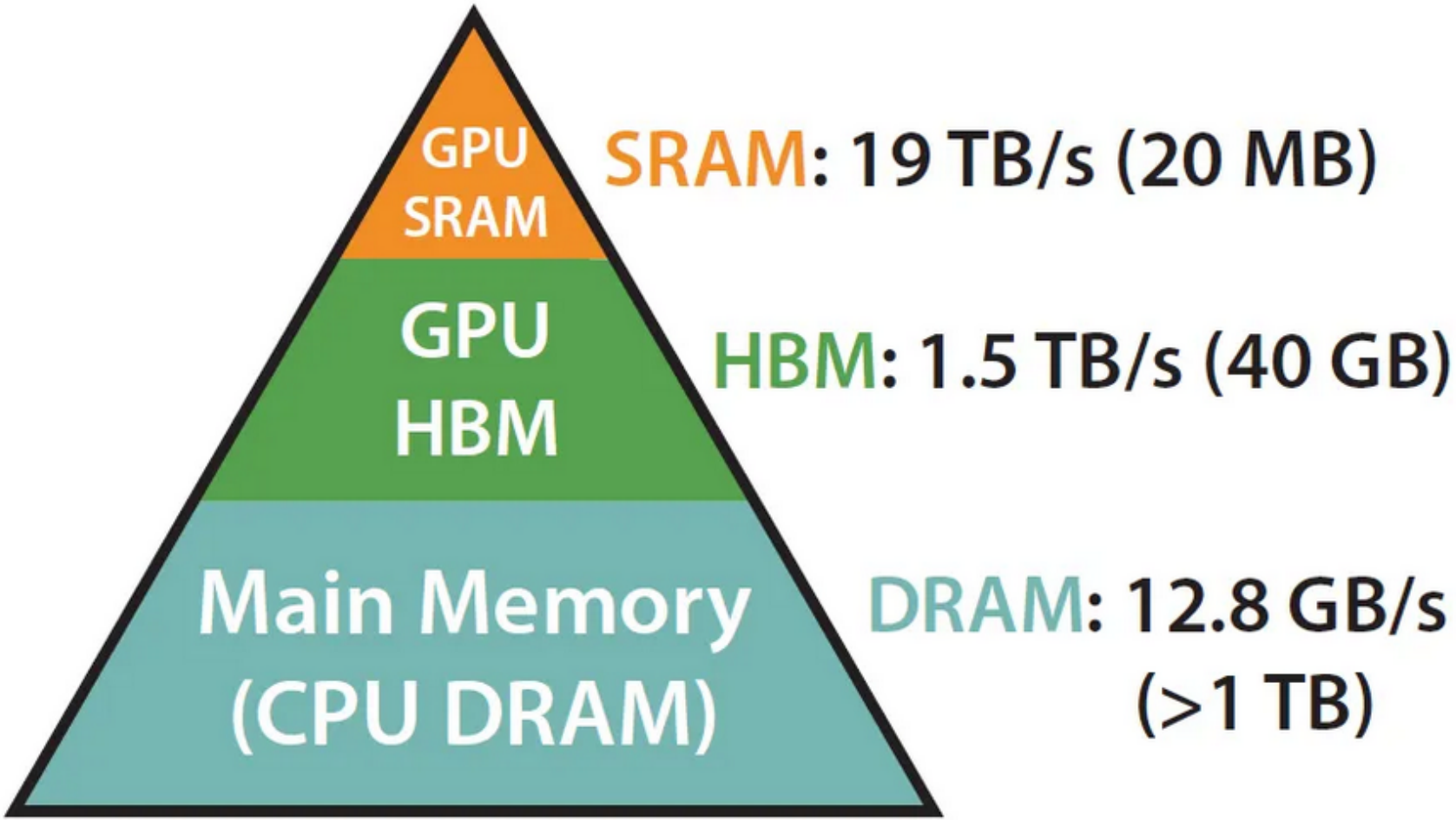
Source: Nvidia

FlashAttention

Fast and Memory-Efficient Exact Attention with IO-Awareness

- Fast
  - 15 % faster than BERT
  - 3x faster than GPT-2
  - 2.4x faster than Megatron-LM
- Memory-efficient
  - Reducing from $O(n^2)$ to $O(n)$
- Exact
  - Same as “vanilla attention”, not an approximation
- IO aware
  - Reducing memory load/store operations

GPU MEMORY HIERARCHY

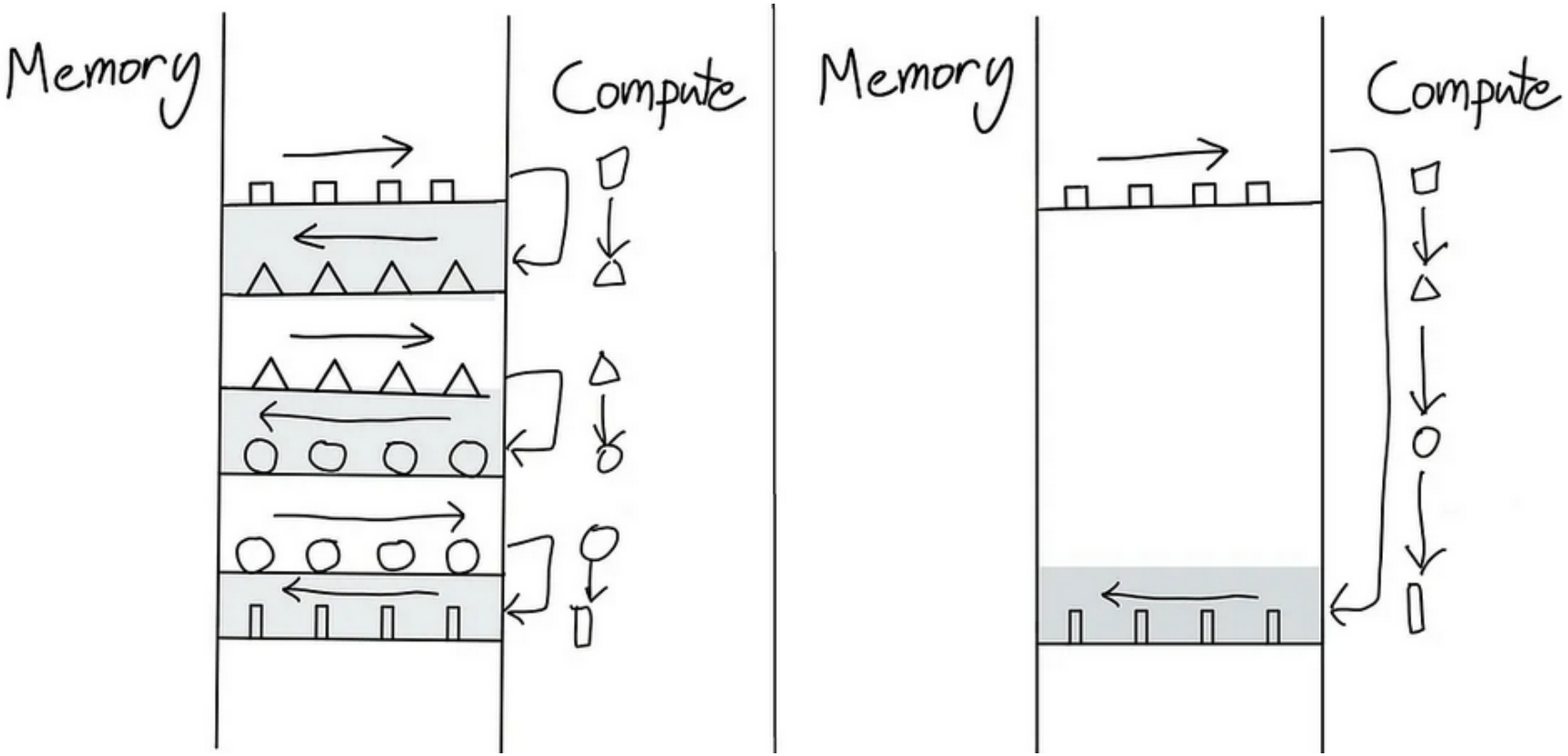


Source: Dao et al. (2022)

COMPUTING CONSIDERATIONS

- GPU compute has been growing faster than memory bandwidth
  - GPU has to wait for data
- Transformer operations are memory-bound
  - Elementwise operations with high memory access
- IO aware means reducing memory load/store operations
- FlashAttention implements the following:
  - Operation fusion to reduce memory access
  - Tiling or chunking the softmax matrix into blocks
  - Recomputation for better memory utilization

OPERATION FUSION



Source: `https://horace.io/brrr_intro.html`

LIMITATIONS AND PROSPECTS

- FlashAttention requires writing attention to CUDA language
  - A new CUDA kernel for each new attention implementation
  - CUDA is lower-level than PyTorch
  - Implementation may not be transferable across GPUs
- Towards IO-Aware Deep Learning
  - Extending beyond attention
- Multi-GPU IO-Aware Methods
  - FlashAttention computation may be parallelizable across multiple GPUs

NUMBER OF PARAMETERS: NOTATION

- Up to now: number of parameters = P
- From now on: number of parameters = N

SCALING LAWS

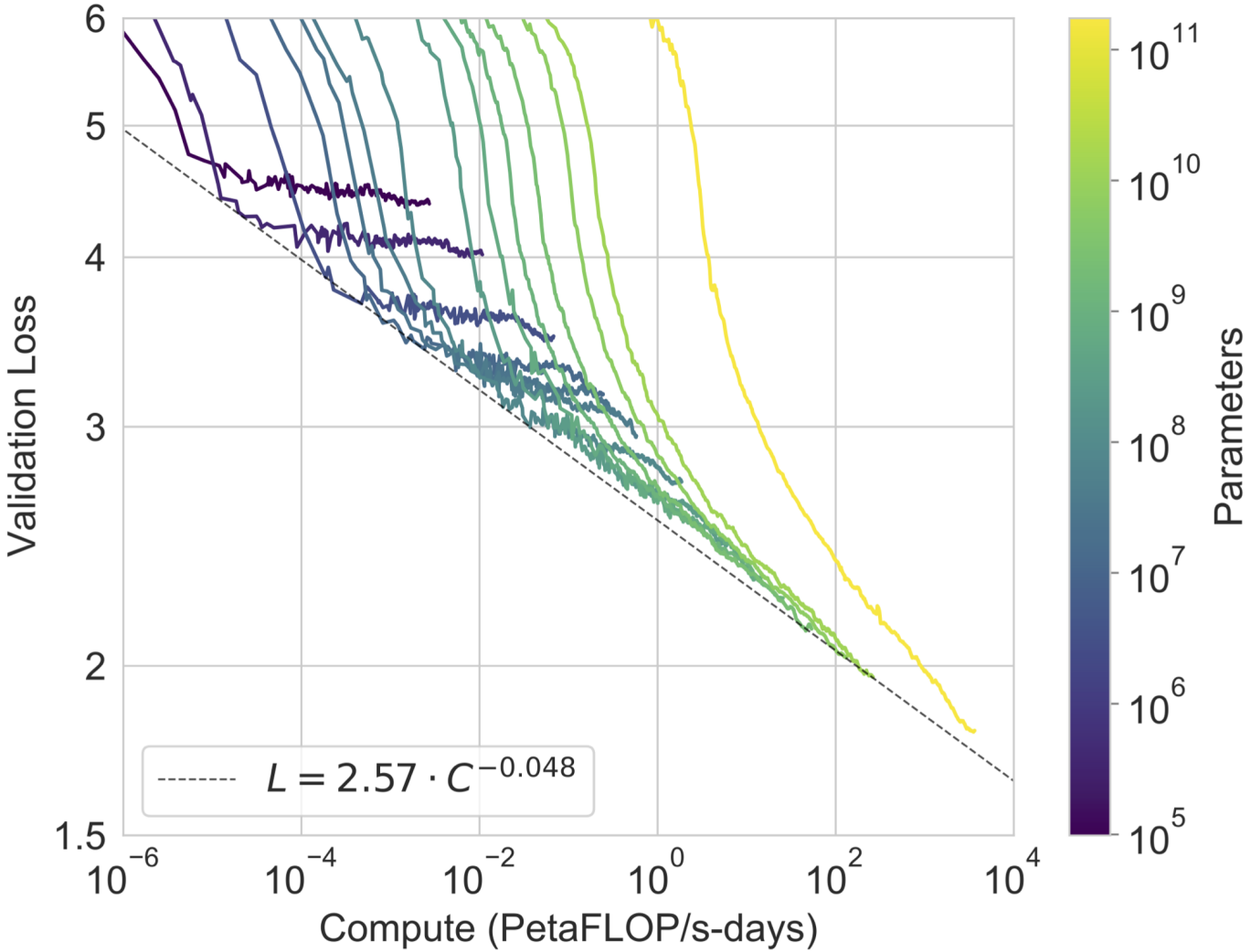
▸ Kaplan et al. (2020)

- Performance depends strongly on scale, weakly on model shape ?!?
  - Scale means: parameters N, data D, and compute C
  - Shape means: depth and width
- Smooth power laws ?!?
  - Performance has power-law relation with each factor N, D, C
  - When not bottlenecked by the other two
  - Trend spanning more than six orders of magnitude
- Universality of overfitting ?!?
  - Performance enters regime of diminishing returns if N or D held fixed while the other increases

SCALING LAWS

- Universality of training (see figure next slide) ?!?
  - Training curves follow predictable power laws
  - Their parameters are roughly independent of model size
  - It is possible to predict by extrapolating the early part of the training curve
- Transfer improves with test performance ?!?
  - When evaluating on text with different distribution from training text, results are strongly correlated to those on the validation set
  - Transfer to different distribution incurs a constant penalty but improves in line with performance on training set
- Sample efficiency ?!?
  - Large models are more sample-efficient than small models
  - They reach same performance with fewer optimization steps

POWER LAW (GPT3 PAPER)



SCALING LAWS

- Convergence is inefficient
  - When C is fixed but N and D are not, optimal performance is achieved by training very large models and stopping significantly short of convergence (QUESTION: why?)
- Optimal batch size ?!?
  - "Optimal batch size: The ideal batch size for training these models is roughly a power of the loss only, and continues to be determinable by measuring the gradient noise scale [MKAT18]; it is roughly 1-2 million tokens at convergence for the largest models we can train."
  - gradient noise scale = a measure of the signal-to-noise ratio of gradient across training examples

Larger language models will perform better and be more sample efficient than current models. ?!?

OPTIMAL BATCH SIZE

▸ Kaplan et al. (2018)

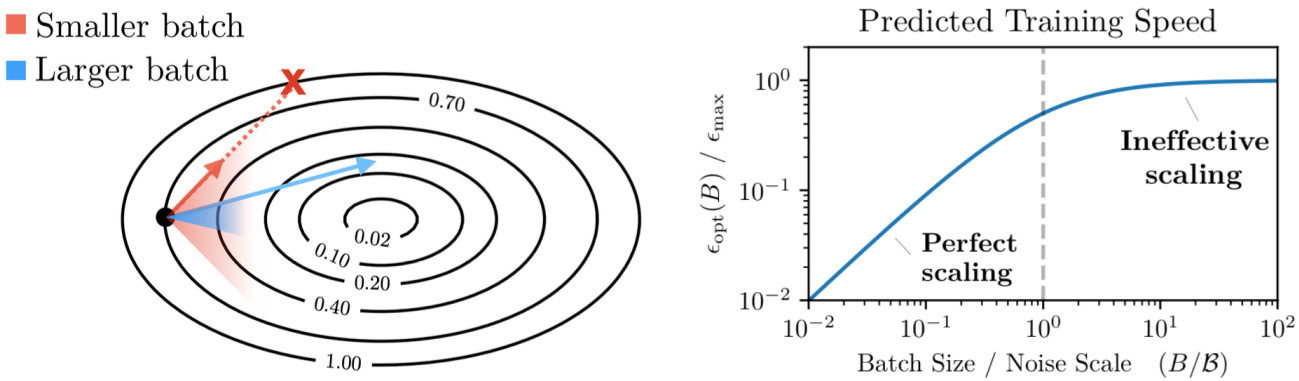


Figure 3: Larger batch sizes yield estimated gradients that are closer to the true gradient, on average. Larger step sizes can be used when the estimated gradient is closer to the true gradient, so more progress can be made per step. **Left:** A large step size used with a small batch size can lead to instability, as illustrated for a quadratic loss. **Right:** Equation 2.6 predicts that the 'turning point' after which larger batch sizes become less helpful is the noise scale $\mathcal{B}$, where the training speed drops to 50% of the maximum possible.

- Estimate gradient as accurately as possible $\rightarrow$ large batch
- Increase training speed as much as possible $\rightarrow$ large step size
- Based on the estimated gradient, choose a step size such that the cost of the landing position does not deviate too much from the cost of the ideal landing position $\rightarrow$ small step size

OPTIMAL BATCH SIZE

- Exploit stochasticity (epoch batches would not be a good thing even if we could compute them efficiently) $\rightarrow$ small step size

SCALING LAW FOR NEXT WORD PREDICTION

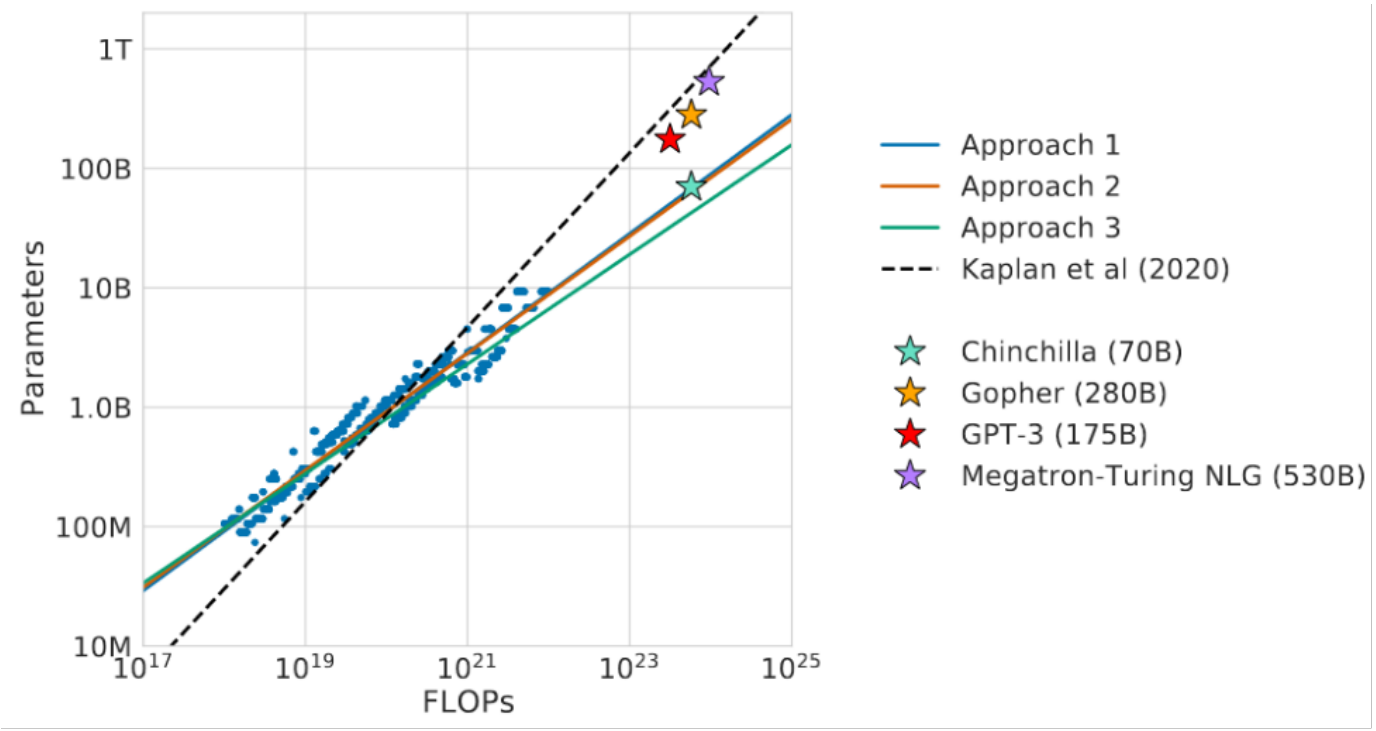
- $L(N, D) = 1.61 + \frac{406.4}{N^{0.34}} + \frac{410.7}{D^{0.28}}$
- $L(N, D)$ is cross entropy on new text.

COMPUTE-OPTIMAL LLMS

Given a fixed FLOPs budget, how should we trade off model size and text size to optimize performance? ▸ Hoffmann et al., 2022

- Find N and D so that $FLOPs(N, D) = C$ and $L(N, D)$ is minimal
- Empirically estimated N and D based on 400 models.
  - Ranging from 70 M to 16 B parameters
  - Trained on 5 B to 400 B tokens
- Different results from those of ▸ Kaplan et al., 2020
- Results verified using Chinchilla
  - Chinchilla has 70 B parameters and is trained on 1.4 T tokens
  - 4x less parameters and 4x more tokens than Gopher
  - Chinchilla outperforms Gopher and has reduced memory footprint and inference cost

COMPUTE-OPTIMAL LLMS: ARE GPT3 ETC TOO LARGE?



▸ Source: Hoffmann et al., 2022

COMPUTE-OPTIMAL LLMS (2)

Given a fixed FLOPs budget, how should one trade off model size and the number of training tokens? We find that all three methods predict that current large models should be substantially smaller and therefore trained much longer than is currently done. Based on our estimated compute-optimal frontier, we predict that for the compute budget used to train Gopher, an optimal model should be 4 times smaller, while being training on 4 times more tokens. We verify this by training a more compute-optimal 70B model, called Chinchilla, on 1.4 trillion tokens. Not only does Chinchilla outperform its much larger counterpart, Gopher, but its reduced model size reduces inference cost considerably and greatly facilitates downstream uses on smaller hardware. The energy cost of a large language model is amortized through its usage for inference and fine-tuning. The benefits of a more optimally trained smaller model, therefore, extend beyond the immediate benefits of its improved performance.

CHINCHILLA AND THE OTHER LLMs

Model	Size (# Parameters)	Training Tokens
LaMDA (Thoppilan et al., 2022)	137 Billion	168 Billion
GPT-3 (Brown et al., 2020)	175 Billion	300 Billion
Jurassic (Lieber et al., 2021)	178 Billion	300 Billion
Gopher (Rae et al., 2021)	280 Billion	300 Billion
MT-NLG 530B (Smith et al., 2022)	530 Billion	270 Billion
Chinchilla	70 Billion	1.4 Trillion

▸ Source: Hoffmann et al., 2022

Model	Layers	Number Heads	Key/Value Size	d_{model}	Max LR	Batch Size
Gopher 280B	80	128	128	16,384	4×10^{-5}	3M $\rightarrow$ 6M
Chinchilla 70B	80	64	128	8,192	1×10^{-4}	1.5M $\rightarrow$ 3M

▸ Source: Hoffmann et al., 2022

CHINCHILLA OUTPERFORMS OTHER LLMS: MMLU

Random	25.0%
Average human rater	34.5%
GPT-3 5-shot	43.9%
Gopher 5-shot	60.0%
***Chinchilla* 5-shot**	**67.6%**
Average human expert performance	*89.8%*
June 2022 Forecast	57.1%
June 2023 Forecast	63.4%

▸ Source: Hoffmann et al., 2022

CHINCHILLA OUTPERFORMS OTHER LLMS: QA

	Method	*Chinchilla*	*Gopher*	GPT-3	SOTA (open book)
Natural Questions (dev)	0-shot	16.6%	10.1%	14.6%	54.4%
	5-shot	31.5%	24.5%	-	
	64-shot	35.5%	28.2%	29.9%	
TriviaQA (unfiltered, test)	0-shot	67.0%	52.8%	64.3 %	-
	5-shot	73.2%	63.6%	-	
	64-shot	72.3%	61.3%	71.2%	
TriviaQA (filtered, dev)	0-shot	55.4%	43.5%	-	72.5%
	5-shot	64.1%	57.0%	-	
	64-shot	64.6%	57.2%	-	

▸ Source: Hoffmann et al., 2022