DATA MOVEMENT LIMITS TO FRONTIER MODEL TRAINING

Ege Erdil[*]
Epoch AI
ege@epochai.org

David Schneider-Joseph[*]
david@davidsj.com

ABSTRACT

We present a theoretical model of distributed training, and use it to analyze how far dense and sparse training runs can be scaled. Under our baseline assumptions, given a three month training duration, data movement bottlenecks begin to significantly lower hardware utilization for training runs exceeding about 10^{28} FLOP, two orders of magnitude above the largest training run to date, **suggesting the arrival of fundamental barriers to scaling in three years** given recent rates of growth. A training run exceeding about 10^{31} FLOP is infeasible even at low utilization. However, more aggressive batch size scaling and/or shorter and fatter model shapes, if achievable, have the potential to permit much larger training runs. An interactive version of our model will shortly be accessible here.

1 INTRODUCTION

Scaling up neural networks and training them on more examples is crucial for good task performance (Hestness et al., 2017; Kaplan et al., 2020; Hoffmann et al., 2022; Bi et al., 2024), with state-of-the-art models requiring tens of thousands of GPUs[1] to train in a reasonable duration. Previous work (Huang et al., 2019; Rajbhandari et al., 2020; Lepikhin et al., 2020; Narayanan et al., 2021; Jiang et al., 2024b) has developed practical techniques enabling the rapid scaling of the past decade (Sevilla et al., 2022).

In this work, we address unexamined fundamental questions about **limits to scaling in the future:**

Q1 Given present-day algorithms, GPUs, and interconnects, what is the biggest training run that can be performed within a fixed duration, before intra- and inter-GPU data movement starts to seriously worsen utilization or even render it impossible?

Q2 How far might this limit be extended, and what algorithmic or hardware progress can achieve that?

Answering these questions empirically would require millions of GPUs and large-scale engineering efforts, so we instead approach them theoretically. In doing so, we develop a simulator that can find optimal training run configurations accounting for the factors that we identify as fundamental. This gives us the answers:

A1 With most current technology, **GPU utilization starts to fall at $\approx 10^{28}$ floating point operations (FLOP), about three years away at recent trends** (Epoch AI, 2024) of $4.2\times$ growth per year.

[*] Equal contribution.

[1] We focus on GPUs, but our theoretical model and findings are broadly applicable to other accelerators, and even groups of accelerators.

Currently-available specialized high-bandwidth inter-node interconnects can permit training runs about two orders of magnitude larger ($\approx 10^{30}$ FLOP), at which point latencies begin to worsen utilization, until reaching **an absolute latency barrier at $\approx 10^{31}$ FLOP, about seven years away.**

A2 Improved hardware interconnects may buy no more than two orders of magnitude in training run size, assuming technology anything like the current paradigm. Beyond that, the critical innovations must come from machine learning algorithms: **The key challenge is transforming two serial dependencies — between batches and between layers — into opportunities for parallelism, by making batch sizes bigger (perhaps enabled by sparsity) and models wider and shallower.** Achieving these goals may be quite difficult in practice.

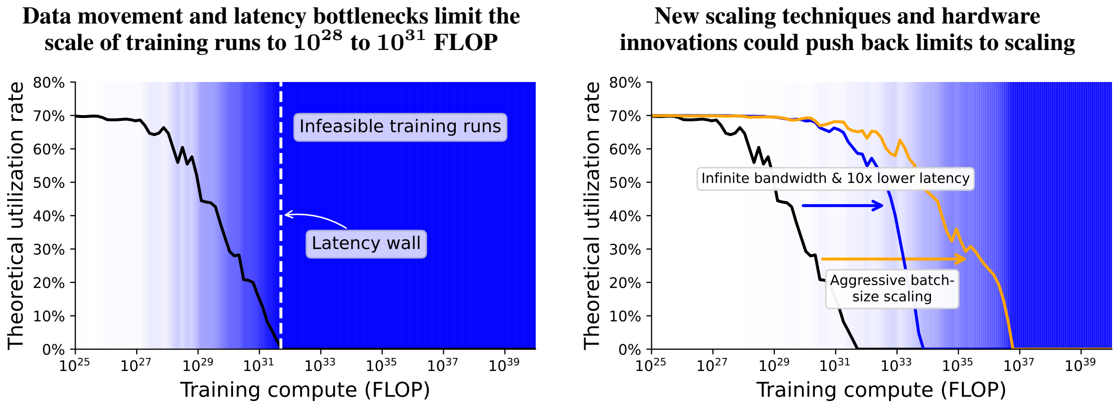


Figure 1: With current technology, such as the H100 GPU and current scaling techniques, data movement bottlenecks lower hardware utilization for training runs exceeding 10^{28} FLOP, and a "latency wall" renders surpassing 10^{31} FLOP infeasible (left). However, with innovations in scaling (such as techniques to enable much larger batch sizes) or dramatic increases in network bandwidth coupled with a $10\times$ reduction in inter- and intra-GPU latency, training runs can be at least a few orders of magnitude larger (right).

This work is organized as follows:

- Section 2 introduces a simplified model of a neural network consisting of stacked sparse linear multi-layer perceptrons that we use as the basis for our analysis throughout the paper.
- Section 3 provides an overview of the four main parallelism strategies employed in distributed training—data, tensor, pipeline, and expert parallelism—and summarizes their communication costs.
- Section 4 identifies the key factors that constrain distributed training, including data movement, critical batch size, and latency, and model depth.
- Section 5 derives closed-form expressions for the maximum training scale under this model.
- Section 6 presents the complete theoretical model accounting for all identified constraints and discusses simulation results on current hardware, demonstrating the limits to efficient scaling.

We conclude by discussing the implications of our findings, and posing key open technical questions whose answers will determine the limits to large-scale model training.

2 A TOY MODEL: STACKED SPARSE LINEAR MLP BLOCKS

We first carefully define the class of models to be considered. Since the Transformer (Vaswani et al., 2017), including its sparse varieties (Fedus et al., 2022), is the dominant architecture today for frontier models, it seems a natural baseline. As we will show momentarily, the great majority of computation when training a Transformer occurs in its linear layers, so we adopt a simplified model consisting of these elements. This approach has the advantage that such layers also constitute the bulk of computation in many alternative architectures (Peng et al. (2023) and Gu & Dao (2023) among others), so our model retains applicability to a wide variety of potential future algorithmic developments. However, the huge space of possible unexpected algorithmic developments precludes any watertight guarantees, should state-of-the-art architectures or learning algorithms change drastically.

The formal definitions we make are as follows: a **sparse linear multi-layer perceptron (MLP) block** ("MLP block" for short) consists of learnable weight matrices $W_1^e \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ and $W_2^e \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$ for **expert indices** e in the range $1 \leq e \leq E$, where E is the **sparsity factor** of the model.

Given a matrix of input vectors $X_{\text{in}} \in \mathbb{R}^{d_{\text{model}} \times b}$, where b is the batch size in units of tokens, a router $\rho : \mathbb{R}^{d_{\text{model}}} \to \{1, \ldots, E\}$ (assumed to be computationally inexpensive) chooses some expert index $\rho(X_{\text{in}}^k)$ for each token index $k \leq b$. The sparse linear MLP f then yields an output matrix $X_{\text{out}} = f(X_{\text{in}}) \in \mathbb{R}^{d_{\text{model}} \times b}$ by mapping each token's input column X_{in}^k to the corresponding output column X_{out}^k via the weight matrices corresponding to the chosen expert. If $^{(e)}$ indexes the set of tokens routed to expert e, then:

$$X_{\text{out}}^{(e)} = W_2^e \left(W_1^e X_{\text{in}}^{(e)} \right).$$

We optimistically assume *balanced routing* (Section 3.2.2): the same number b/E of tokens is routed to each of the E experts, so that during the forward pass every expert performs matrix multiplications of shapes $(d_{\text{ff}}, d_{\text{model}}) \times (d_{\text{model}}, b/E) \to (d_{\text{ff}}, b/E)$ and then $(d_{\text{model}}, d_{\text{ff}}) \times (d_{\text{ff}}, b/E) \to (d_{\text{model}}, b/E)$, each requiring $d_{\text{model}} d_{\text{ff}} b/E$ multiply-accumulate (MAC) operations. Across both linear layers of all E experts, the MLP block's total arithmetic cost is $2 d_{\text{model}} d_{\text{ff}} b$ MAC. We then assume L MLP blocks in total, so that the model's forward pass is their composition:

$$F(X) = f_L(f_{L-1} \cdots (f_1(X))).$$

Across these L blocks, the model has

$$N_{\text{p}} = 2LE d_{\text{model}} d_{\text{ff}} \tag{1}$$

parameters in total and a forward pass on a batch size of b requires $N_{\text{p}} b/E = 2L d_{\text{model}} d_{\text{ff}} b$ MAC.

During the backward pass, gradients of the loss $\mathcal{L}$ must be computed for the inputs (activations and weights) of each matrix multiplication. A matrix multiplication $C = AB$ of shape $(I, K) \times (K, J) \to (I, J)$ on the forward pass becomes two matrix multiplications,

$$\partial \mathcal{L} / \partial A = (\partial \mathcal{L} / \partial C) B^\top,$$
$$\partial \mathcal{L} / \partial B = A^\top (\partial \mathcal{L} / \partial C),$$

of shapes $(I, J) \times (J, K) \to (I, K)$ and $(K, I) \times (I, J) \to (K, J)$ on the backward pass, all requiring the same number IJK of MAC. Thus accounting for the backward pass, the number of MAC for our model F is tripled and becomes $6Ld_{\text{model}}d_{\text{ff}}b$.

An actual Transformer has linear layers not only in its MLP but also for attention queries, keys, values, and outputs. These can be fused with and computed in parallel with the MLP block (Wang & Komatsuzaki, 2021; Chowdhery et al., 2022), so that our simplified model encompasses these. However, there are also operations which we neglect:

- Element-wise operations such as nonlinear activation functions,[2] layer normalization (Ba et al., 2016), and residual accumulation (He et al., 2015). We treat these as negligible as they involve $O(d_{\text{model}})$ or $O(d_{\text{ff}})$ computations per token, as compared to the linear layers' $O(d_{\text{model}}d_{\text{ff}})$ computations per token. For large language models, d_{model} is typically on the order of 10^4, and d_{ff} a multiple thereof, so relative negligibility tends to hold in practice. Furthermore, these element-wise operations need not impose additional significant data movement of their own, as they can often be fused with matrix multiplication kernels.

- Embedding and unembedding projections $W_{\text{embed}} \in \mathbb{R}^{d_{\text{model}} \times V}, W_{\text{unembed}} \in \mathbb{R}^{V \times d_{\text{model}}}$. As these are computed only once per forward pass rather than once per Transformer block, and the vocabulary size V is typically not much larger than d_{ff}, we ignore these.

- Scaled dot-product attention scoring and the corresponding linear combination of the attention head value vectors. For a sequence length ℓ and typical attention width $n_{\text{heads}} \cdot d_{\text{head}} \approx d_{\text{model}}$, this requires $O(\ell d_{\text{model}} b)$ MAC per attention layer each batch. While not completely negligible, this cost is typically small compared to the MLP MAC $\approx 2d_{\text{model}}d_{\text{ff}}b$ above, as the sequence length ℓ is usually significantly smaller than the internal MLP width d_{ff} during most of training. (Dubey et al., 2024; Devlin et al., 2019)

We thus see that our toy model, despite being dramatically simpler to analyze and understand than an actual Transformer, encompasses most of its arithmetic work.

We have so far considered only arithmetic costs. To understand bottlenecks to distributed training, we must also investigate the costs of data movement, both within and between GPUs. Much of what follows will concern itself with this investigation.[3] To proceed, we must first understand how the work of a training run can be distributed across GPUs.

3 Methods of Parallelism

The workload of a training run can be distributed through a variety of methods (Weng & Brockman, 2022): data parallelism (DP), tensor parallelism (TP), pipeline parallelism (PP), and expert parallelism (EP), corre-

[2] Logically, dropping the nonlinearities makes a dense Transformer architecture trivial, as the end-to-end map F becomes linear and thus representable by a single matrix. We assume the nonlinearities are still present, but omit modeling them due to their relative computational insignificance.

[3] Beyond our toy model, element-wise operations need not impose additional significant data movement of their own, as they can often be fused with matrix multiplication kernels. Even when not, the volume tends to be small relative to weight matrices due to slicing along the batch dimension from data and pipeline parallelism into very small nanobatches (Section 5.1), necessitating many accesses of the same weight matrix per batch. Scaled dot-product attention can impose additional data movement, especially if the nanobatch size is significantly smaller than the sequence length, which will be one of several factors causing our results to err on the optimistic side. Actual achieved training run sizes and utilizations will likely be bounded above by our model.

sponding to the four problem dimensions: batch size, layer width, model depth, and sparsity, respectively.[4] This list of parallelism methods is also exhaustive, because a single matrix multiplication only allows for tensor and data parallelism. Further parallelism must come from assigning different matrix multiplications to different GPUs, which can be done vertically (pipeline parallelism) or horizontally (expert parallelism). In this section, we briefly overview these methods, organized by their characteristic communication patterns.

3.1 Tensor slicing and all-reduce: data and tensor parallelism

Consider a forward pass matrix multiplication $C = AB$ of shape $(I, K) \times (K, J) \to (I, J)$ and its backward pass computations $\partial\mathcal{L}/\partial A = (\partial\mathcal{L}/\partial C)B^\top$ and $\partial\mathcal{L}/\partial B = A^\top(\partial\mathcal{L}/\partial C)$ as described in Section 2. For example, A could be the weight matrix W_1^e for an expert's first layer, and B its input activations $X_{\text{in}}^{(e)}$, in which case the forward pass multiplication shape is $(d_{\text{ff}}, d_{\text{model}}) \times (d_{\text{model}}, b/E)$, as discussed earlier.

3.1.1 One-dimensional slicing

A natural idea is to partition the data (and associated work) along one of the dimensions of size I, J, or K, while replicating the other dimensions. When this is the expert's batch dimension of size b/E, we call this **data parallelism,** and when it's one of the layer shape dimensions d_{model} or d_{ff}, we call it **tensor parallelism.** Fundamentally they require the same communication pattern, which we now describe in general terms.

If the data is partitioned along the "internal" K-sized dimension into K'-sized chunks across $N_K = K/K'$ GPUs, and the computation partitioned accordingly, then the n^{th} GPU performs a matrix multiplication of shape $(I, K') \times (K', J) \to (I, J)$, defined by:

$$c_{ij}^n = \sum_{k=nK'}^{(n+1)K'-1} a_{ik} b_{kj}.$$

Each GPU thus has only a partially-reduced value for each component of C, but requires the fully-reduced component $c_{ij} = c_{ij}^0 + c_{ij}^1 + \ldots + c_{ij}^{N_K-1}$ so that it may proceed with the next computation step. This necessitates an **all-reduce** collective communication across the N_K GPUs. (Kumar et al., 1994)

We can derive the minimal inter-GPU data movement volume for this all-reduce, assuming that messages contain the single-word partial sum accumulated so far by the transmitting GPU. Given any total ordering of messages consistent with their causal partial ordering, the $(N_K - 1)^{\text{th}}$ message is the first whose receiver can possibly lie causally downstream of all other GPUs and therefore contain the fully-reduced component c_{ij}. The other $N_K - 1$ GPUs must then also receive at least one additional message, so there must be at least $2(N_K - 1)$ messages, and hence words, received in total.[5]

That is, to all-reduce a single word across N_K GPUs, each GPU must receive

$$2(N_K - 1)/N_K \approx 2 \tag{2}$$

words.

[4] For a Transformer, *sequence parallelism* is also available, partitioning work even across different tokens in the same sequence. This can be thought of as data parallelism with some additional communication for the attention layers. We omit discussion of this method because of our restricted focus on linear layers.

[5] Strictly speaking, not every word need be received by a *GPU* per se, but merely by some network device. For example, the network fabric can cut GPU all-reduce bandwidth requirements by about $2\times$. (Graham et al., 2016)

There are $I \cdot J$ such components, so this matrix multiplication's all-reduce must cost at least

$$2IJ(N_K - 1)$$

words of inter-GPU data movement.[6] This lower bound can in fact be achieved exactly by, for example, consecutively performing a reduce-scatter and an all-gather operation (Kumar et al., 1994).

On the backward pass matrix multiplications of shapes $(I, J) \times (J, K) \rightarrow (I, K)$ and $(K, I) \times (I, J) \rightarrow (K, J)$, the K-sized dimension is "external", hence requires no reduction, and therefore no inter-GPU communication.

If instead the data is partitioned across N_I GPUs along the I-sized dimension, then this dimension is "internal" (i.e. requiring reduction) only for the second backward pass matrix multiplication of shape $(K, I) \times (I, J) \rightarrow (K, J)$, imposing

$$2KJ(N_I - 1)$$

words of inter-GPU data movement in the backward pass, and none in the forward pass. Symmetrically, a partition across N_J GPUs along the J-sized dimension requires

$$2IK(N_J - 1)$$

words of inter-GPU data movement in the backward pass, and none in the forward pass.

3.1.2 Multi-dimensional slicing

In general, we may partition the data across $N = N_I N_J N_K$ GPUs, with each performing a forward pass multiplication of shape $(I', K') \times (K', J') \rightarrow (I', J')$, where as before we define $I' = I/N_I$, etc.

Because communication in the forward pass happens only along the "inner" K-sized dimension, this can be treated as $N_I N_J$ independent matrix multiplications, entailing (as derived above) $2I'J'(N_K - 1)$ words of inter-GPU data movement each. Across all independent multiplications, this works out to $2(N_I I')(N_J J')(N_K - 1) = 2IJ(N_K - 1)$ words of inter-GPU data movement in the forward pass, exactly as in the K-only parallelism case.

We find similarly for backward pass communication across the I and J dimensions, so that:

$$\text{total inter-GPU data movement } = 2[IJ(N_K - 1) + KJ(N_I - 1) + IK(N_J - 1)] \text{ words.} \tag{3}$$

Let us now consider intra-GPU data movement between main DRAM memory and the logic chip.[7] Assuming an ideal cache, each forward pass matrix multiplication loads $I'K' + K'J'$ words from memory and writes $I'J'$ words to memory, for

$$I'K' + K'J' + I'J'$$

[6] Here and throughout, we count only words received. Words transmitted will always either be equal, or (in a multicast setting) at least proportional.

[7] See Appendix 8.1 for a discussion of data movement between SRAM-based caches, shared memory, register banks, and execution units.

words accessed per GPU in the forward pass. The two backward pass multiplications behave similarly, tripling total memory IO, so that across all $N_I N_J N_K$ GPUs we have[8]

$$\text{total intra-GPU data movement} = 3(IJN_K + KJN_I + IKN_J) \text{ words.}$$

Note the similarity to the inter-GPU data movement formula Eq. 3 above, both having N_I, N_J, N_K-dependent term proportional to $IJN_K + KJN_I + IKN_J = IJK(1/I' + 1/J' + 1/K')$, or simply

$$1/I' + 1/J' + 1/K', \tag{4}$$

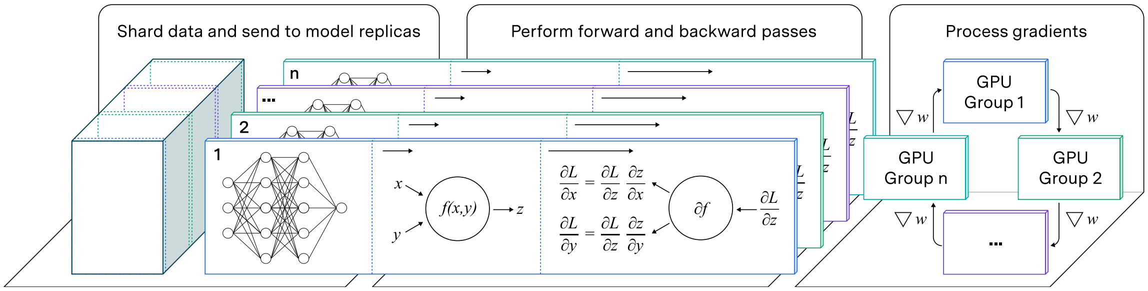
when we hold the problem dimensions I, J, K constant.

If we also hold the total number of GPUs N – and thus the per-GPU problem size $I'J'K' = IJK/N$ – constant, we find Eq. 4 is minimized when $I' = J' = K'$, that is when all three degrees of parallelism are sized so as to give each GPU a cube-shaped work unit. In practice the constraint that N_I, N_J, N_K be positive integers means this exact condition is rarely achieved, and when N is particularly small, the smallest dimensions go un-parallelized. As our concern is the limits to distributed training, we typically consider regimes where N is quite large and the condition approximately holds.

We now apply this analysis to the concrete cases of data and tensor parallelism.

3.1.3 Data parallelism

In this case, each expert's activations $X_{\text{in}}^{(e)}$ and $X_{\text{out}}^{(e)}$ (as well as intermediate states $W_1^e X_{\text{in}}^{(e)}$) are sliced N_{DP}-way along the b/E-sized batch dimension. This becomes a reduction dimension in the backward pass when computing gradients with respect to the (replicated) weight matrices W_1^e and W_2^e (via multiplications of shapes $(d_{\text{ff}}, b/E) \times (b/E, d_{\text{model}})$ and $(d_{\text{model}}, b/E) \times (b/E, d_{\text{ff}})$), so *data parallelism necessitates an all-reduce of weight gradients across the batch dimension*, with inter-GPU data movement per weight matrix of $2d_{\text{ff}}d_{\text{model}}(N_{\text{DP}} - 1)$ words per batch, as can be seen by applying Eq. 3 to this case.



Shard data and send to model replicas
Perform forward and backward passes
Process gradients
n
...
2
1
x
y
f(x,y)
z
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
∂f
$\frac{\partial L}{\partial z}$
GPU Group 1
GPU Group 2
GPU Group n
...
∇w
∇w
∇w
∇w

Figure 2: Data parallelism. The input data is divided into shards and processed independently by multiple model replicas. Each replica computes gradients for its local shard, which are then all-reduced across all replicas to obtain the full batch gradient. This full gradient finally updates the model parameters on each replica.

[8] In fact, caching is imperfect, increasing required IO, while some opportunities for cache hits across matrix multiplications can decrease this IO. We treat this as a first-order approximation in the regime where the matrices on each GPU are large enough that they would not fit into SRAM – our full model also considers the scenario where the matrices are sufficiently small that we can cut out DRAM reads and writes altogether.

Aggregating across the L MLP blocks, E experts per block, and two layers per expert, this amounts to a total inter-GPU data movement of $4LEd_{\text{ff}}d_{\text{model}}(N_{\text{DP}}-1)$ words per batch, i.e.

$$2N_{\text{p}}(N_{\text{DP}}-1)/N_{\text{DP}}$$

words per batch per data-parallel worker, where $N_{\text{p}} = 2LEd_{\text{ff}}d_{\text{model}}$ is the total number of model parameters.

Across those E experts, each data-parallel worker sees b/N_{DP} tokens. Naively implemented, each stores a full copy of the N_{p} model parameters and associated optimizer state. However, the optimizer state can be sharded by inserting the optimizer step *in between* a reduce-scatter/all-gather all-reduce implementation: first the gradients are reduce-scattered, then each worker optimizes the weights corresponding to its gradient partition, then the new weights are all-gathered across workers. (Rajbhandari et al., 2020)

3.1.4 Tensor parallelism

In this case, an expert's activations and weights are sliced into $N_{\text{TP}} = N_{\text{TP, ff}} \times N_{\text{TP, model}}$ partitions, along the d_{ff} and d_{model} dimensions. The d_{model} dimension is internal in the first layer's forward pass of shape $(d_{\text{ff}}, d_{\text{model}}) \times (d_{\text{model}}, b/E)$, as well as the second layer's activation gradient computation of shape $(d_{\text{ff}}, d_{\text{model}}) \times (d_{\text{model}}, b/E)$. Symmetrically, the d_{ff} dimension is internal in the second layer's forward pass and the first layer's activation gradient computation. Applying Eq. 3, we see $2(b/E)[d_{\text{ff}}(N_{\text{TP, model}}-1) + d_{\text{model}}(N_{\text{TP, ff}}-1)]$ words of inter-GPU data movement for each of an expert's two layers. Aggregating across the L MLP blocks, E experts per block, and two layers per expert, we have total inter-GPU data movement of

$$4Lb[d_{\text{ff}}(N_{\text{TP, model}}-1) + d_{\text{model}}(N_{\text{TP, ff}}-1)]/N_{\text{TP}}$$

words per batch per tensor-parallel worker.

From our general analysis above, we see that when N_{TP} is small (the norm today), the smaller dimension d_{model} goes un-partitioned ($N_{\text{TP, model}} = 1$), yielding **1D tensor parallelism**, but that otherwise (as for the much larger training runs which are our focus), **2D tensor parallelism** becomes optimal, with $d_{\text{ff}}/N_{\text{TP, ff}} \approx d_{\text{model}}/N_{\text{TP, model}}$ (i.e. roughly square weight partitions). Solving this yields approximately

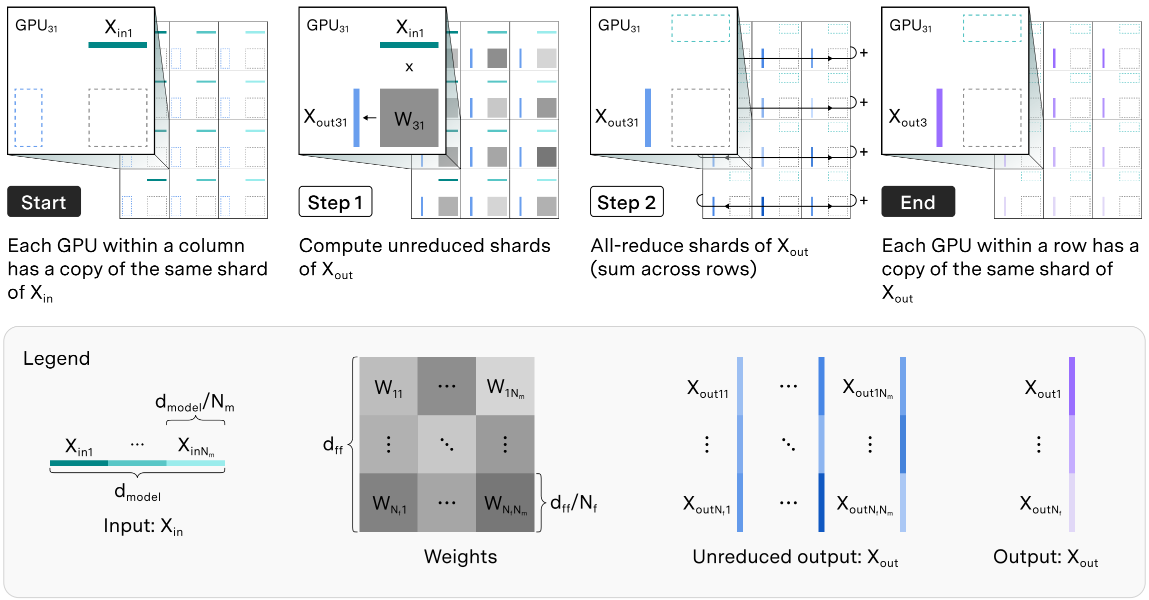
$$8Lb\sqrt{d_{\text{model}}d_{\text{ff}}/N_{\text{TP}}}$$

words of inter-GPU data movement per batch per tensor-parallel worker.

3.2 Point-to-point: pipeline and expert parallelism

We have so far examined partitioning the work along the per-expert batch dimension of size b/E (data parallelism) and the two weight matrix dimensions of size d_{model} and d_{ff} (tensor parallelism). Two problem dimensions remain: we can also partition depth-wise across the L MLP blocks with **pipeline parallelism,** and/or across the E experts with **expert parallelism.** In this case, a given token at a given layer is processed by the GPU (or GPUs, if combined with tensor parallelism) corresponding to that layer's **pipeline stage** and that token's routed expert. As the token moves through the layers during the forward and backward passes, its activations are transferred in a simple point-to-point communication of d_{model} words whenever crossing pipeline- *or* expert-parallel ranks.[9] Because of the disjunctive nature of this condition, the point-to-point communication cannot always be attributed solely to one or the other method of parallelism, and a joint analysis is warranted.

[9] To avoid redundant communication due to replication across tensor-parallel ranks, tensor-parallel all-reduces can be split into two phases: a reduce-scatter *before* the point-to-point communication, followed by an all-gather on the new set of tensor-parallel peers *after* the point-to-point communication. (Narayanan et al., 2021)



Each GPU within a column has a copy of the same shard of X_{in}

Compute unreduced shards of X_{out}

All-reduce shards of X_{out} (sum across rows)

Each GPU within a row has a copy of the same shard of X_{out}

Figure 3: 2D tensor parallelism for an MLP block expert's first layer weight matrix W. Start: Input vector X_{in} is scattered "horizontally" (across $N_m = N_{\text{TP, model}}$ GPUs) and duplicated "vertically" (across $N_f = N_{\text{TP, ff}}$ GPUs), and weight matrix shards W_{11} through $W_{N_f N_m}$ are scattered along d_{ff} (rows) and d_{model} (columns) in the same set of GPUs. Step 1: Each GPU computes a partially-reduced shard of output matrix X_{out}. Step 2: All-reduce operation across GPU rows computes fully-reduced shards of X_{out}. End: Each GPU on a given row has an identical, fully-reduced copy of the X_{out} shard corresponding to that row.

Indeed, when the expert-parallel degree N_{EP} is even moderately large and (as we shall assume) expert routing is stochastically independent of token and layer, then expert-parallel communication *usually* necessitates a point-to-point communication after each MLP block, and so pipeline-parallel communication comes along nearly "for free". However, pipeline parallelism comes with its own challenges, which we now examine.

3.2.1 Pipeline parallelism

In a naive pipeline-parallel configuration, the first pipeline stage is assigned the first L/N_{PP} layers, the second is assigned the next L/N_{PP} layers, *et cetera*. Consequently, in each forward and backward pass, a vector of size d_{model} has to be communicated a total of $N_{\text{PP}} - 1$ times. The total data movement cost across all GPUs is simply $d_{\text{model}}(N_{\text{PP}} - 1)$ activations per forward or backward pass per token.

Because model layers are inherently serial, pipeline parallelism imposes a unique challenge not encountered by the other forms – data, tensor, and expert – of parallelism: minimizing the so-called **pipeline bubble**, the time GPUs spend idle waiting for activations from other pipeline stages. If the entire batch were sent through the pipeline all together, then at any given moment only one of the N_{PP} pipeline stages would ever be active, and the **bubble fraction** (proportion of time spent idle) would equal $1 - 1/N_{\text{PP}}$. So instead, the batch must be sliced further than whatever is imposed by data parallelism alone, into smaller pieces called **microbatches,** which travel individually through the pipeline according to a **pipeline schedule** in which, ideally, the large majority of pipeline stages are active most of the time. Narayanan et al. (2021) notes that a naive pipelining strategy (Fig. 4) with m microbatches and N_{PP}-way pipeline parallelism incurs a bubble

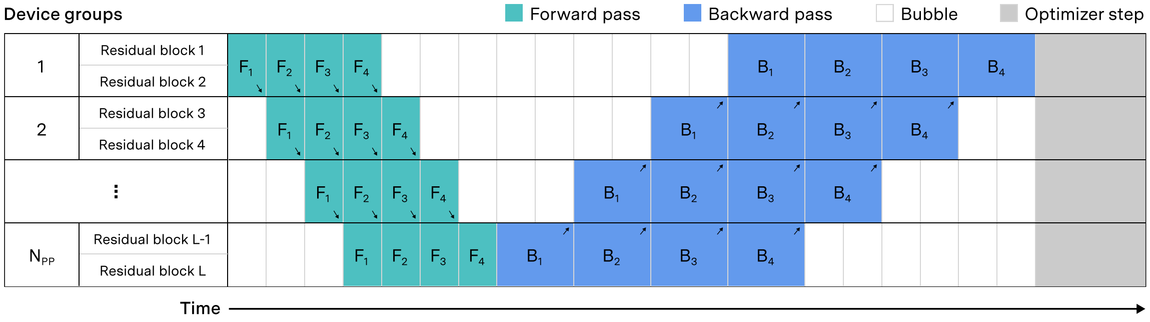


Figure 4: Pipeline parallelism. This diagram depicts the sequential execution flow of microbatches F_1 to F_4 during the training of a deep learning model using a GPipe pipeline schedule (Huang et al., 2019). Device groups, each containing one or more GPUs, are allocated distinct sets of model layers, shown here as residual blocks, and they process the microbatches in a staggered fashion. Pipeline bubbles, the white areas, indicate periods of GPU inactivity due to dependency waits. The optimizer step, highlighted in gray, follows the backward passes and is where model parameter updates occur.

fraction of

$$B_{\text{pp}} = \frac{N_{\text{PP}} - 1}{N_{\text{PP}} - 1 + m}, \tag{5}$$

or a maximum arithmetic utilization of

$$U_{\text{pp}} = 1 - B_{\text{pp}} = \frac{m}{N_{\text{PP}} - 1 + m}.$$

This can be seen most easily from the perspective of the last stage: on the forward pass it spends $N_{\text{PP}} - 1$ bubble steps waiting for that same number of previous stages, then has m steps of work to perform; the backward pass looks the same in reverse. Since all stages have the same amount of work to do, they must also have this same bubble fraction.

Therefore, it's crucial to ensure that the number of microbatches m is large relative to $N_{\text{PP}} - 1$, the number of inter-stage interfaces in the pipeline. But this lowers the arithmetic intensity of the computations done by the GPUs by slicing the data matrix further along the per-expert batch dimension b/E, increasing data movement costs *internal to the GPUs,* as the weights and their accumulated gradients must be accessed *once for each microbatch.* Increasing the batch size b is a natural answer to this conundrum, but this too has limits, which we discuss in Section 4.2.

Pipeline interleaving. Another solution described by Narayanan et al. (2021) is *pipeline interleaving.* By assigning non-adjacent layers of our model to the same pipeline stage, each microbatch travels through the pipeline *multiple times,* for a different set of model layers each time. This increases the effective microbatch count, shortens the initial (and terminal) serial wait time for work to reach (and clear) all stages, and thereby reduces the bubble fraction B_{pp}. This comes at the expense of increased network communication, but this trade-off is usually quite favorable, as the point-to-point communication volume imposed by pipeline parallelism (and shared with expert parallelism) is not large.

We leave a detailed discussion of pipeline interleaving to Appendix 8.2. Here, we note only that if the number of microbatches m is at least equal to the number of pipeline stages N_{PP}, we can pick any divisor i of L/N_{PP} and change the pipeline schedule to achieve the following two outcomes:

1. The bubble fraction becomes $B_{\text{interleaved PP}}(i) = (N_{\text{PP}} - 1)/((N_{\text{PP}} - 1) + im)$: the effective number of microbatches determining the bubble fraction goes up by a factor of i.

2. The total communication cost per forward pass becomes $d_{\text{model}}(N_{\text{PP}} \cdot i - 1)$: the effective number of pipeline stages determining network communication costs goes up by this same factor i.

Zero bubble pipeline parallelism. Qi et al. (2024) recently observed that only *activation* gradients are serial dependencies in the backward pass for previous pipeline stages, and hence the computation of *weight* gradients can be deferred to otherwise idle time in order that those earlier pipeline stages may sooner have work. In this way, their ZB-H2 schedule completely eliminates bubbles. Though this schedule has approximately twice the memory footprint of the interleaved schedule from the previous section, this is a negligible cost in large clusters in which device memory is abundant.

The ZB-H2 schedule requires at least $2N_{\text{PP}} - 1$ microbatches to achieve the zero bubble condition, but this is small compared to the much larger number of microbatches required even to approximate this in a traditional schedule.

3.2.2 Expert parallelism

Expert parallelism involves slicing each layer along the expert dimension E: each pair of weight matrices W_1^j and W_2^j for $1 \leq j \leq E$ is stored on one of N_{EP} expert-parallel ranks, with activation vectors routed to the appropriate rank for their routed expert. Even without any expert parallelism, the sparsity factor E itself already harms the arithmetic intensity of matrix multiplications by shrinking the per-multiplication batch size, and expert parallelism does not in general make matters worse. However, scaling this sparsity factor could potentially increase the largest batch size which can efficiently be used in training (Section 4.2).

In our toy model (Section 2), we consider only settings where each activation vector is routed to a single expert, in which case the point-to-point communication pattern is low-bandwidth and can coincide with that already needed for pipeline parallelism. If tokens are instead routed to multiple experts (Shazeer et al., 2017; Patel & Wong, 2023), with a linear combination taken of their outputs, then a tensor-parallel-style all-reduce across experts is required, potentially across unpredictable levels of the network hierarchy depending on expert placement. As our objective in this paper is to shed light on the limits to distributed training, we make optimistic assumptions when possible. In line with this, we treat the “fine-grained” case as an extension of tensor parallelism and do not model it separately, simplifying our analysis greatly.

Continuing in this spirit, we assume *balanced routing* between experts, independent of token or layer, to ensure uniform workloads across workers. (Lepikhin et al., 2020; Lewis et al., 2021; Zhou et al., 2022)

3.3 5D parallelism

5D parallelism involves combining all of the parallelism methods we’ve discussed so far: data, tensor (both dimensions), pipeline, and expert parallelism. Table 1 summarizes the network communication costs of these methods on an equal footing: how much communication they require per gradient step taken.

To see why combining multiple parallelism methods is more efficient than using any one method of parallelism by itself, consider the isolated problem of minimizing the total network bandwidth cost for a cluster given a fixed cluster size $N_{\text{GPU}} = N_{\text{TP}} N_{\text{PP}} N_{\text{DP}}$ for a dense training run (so that $E, N_{\text{EP}} = 1$). The total bandwidth cost for $i = 1$ can be expressed as

$$\approx 2N_{\text{p}}(N_{\text{DP}} - 1) + 2bd_{\text{model}}(N_{\text{PP}} - 1) + 8bL\sqrt{d_{\text{ff}} d_{\text{model}} N_{\text{TP}}} = c_{\text{DP}} N_{\text{DP}} + c_{\text{PP}} N_{\text{PP}} + c_{\text{TP}}\sqrt{N_{\text{TP}}} - d,$$

	Network bandwidth per gradient step	Slices along...	Communications can coincide with...
Data parallelism	$2N_\text{p}(N_\text{DP} - 1)$	b	Nothing
Tensor parallelism	$\approx 8bL\sqrt{d_\text{ff} d_\text{model} N_\text{TP}}$ (for large N_TP)	$d_\text{model}, d_\text{ff}$	Nothing
Pipeline parallelism	$2bd_\text{model}(N_\text{PP} \cdot i - 1)$	b, L	Expert parallelism
Expert parallelism	$2bd_\text{model}(L-1) \cdot (N_\text{EP} - 1)/N_\text{EP}$	E	Pipeline parallelism

Table 1: The different parallelism methods available along with their network bandwidth costs. Here, N_p is the number of parameters, b the batch size (in tokens), L the number of MLP blocks, d_model and d_ff the model widths, and i the pipeline interleaving factor.

where $c_\text{DP}, c_\text{PP}, c_\text{TP}, d$ are strictly positive constants depending on the model dimensions and batch size. In this problem, for N_GPU sufficiently large[10], it is optimal to scale all parallelism methods at the same time with the total cluster size N_GPU, specifically as

$$N_\text{DP}, N_\text{PP} \propto N_\text{GPU}^{1/4},\ N_\text{TP} \propto N_\text{GPU}^{1/2}.$$

This recalls our conclusion in Section 3.1.2 that slicing along multiple problem dimensions is more efficient than slicing along any single one.

In practice, network bandwidth is not the only constraint on distributed training: arithmetic intensity (i.e. memory bandwidth), communication latency, whether that communication can be overlapped with computation, bubble size management for pipeline parallelism, *et cetera* are all relevant factors. These factors can alter the exact quantitative conclusions of the above argument, but the basic intuition that scaling all parallelism methods together is superior to scaling one in isolation generally remains true.

Table 1 also offers insight into some factors that limit the methods from scaling arbitrarily far. Each needs to slice either on one of the two model width dimensions $d_\text{model}, d_\text{ff}$ or the batch dimension b. A guideline is that if the amount of computation needed to train a model scales faster than $d_\text{ff} d_\text{model} b$ (the inherently *parallel* problem volume, in contrast to the inherently *serial* dimensions of layer count and optimizer steps) as a model is scaled up, distributed training can hit a bottleneck. We make this guideline quantitative in Section 5.

4 What constrains distributed training?

Having discussed the available parallelism methods, we now turn to the main subject of our paper: the obstacles we face if we try to scale some combination of these methods arbitrarily far. Because all these methods incur data movement costs, the arithmetic utilization of the GPUs can fall if a training run becomes bottlenecked by this data movement. We reiterate our key questions:

[10] When N_GPU is small, the constraint $N_\text{TP}, N_\text{DP}, N_\text{PP} \geq 1$ will bind on the more expensive parallelism methods, so e.g. the optimal solution is likely to have $N_\text{TP} = 1$ over a wide range of N_GPU values. This is a big reason why this argument is too simplistic, but it still illustrates the general principle at work.

Q1 Given present-day algorithms, GPUs, and interconnects, what is the biggest training run that can be performed within a fixed duration, before intra- and inter-GPU data movement starts to seriously worsen utilization or even render it impossible?

Q2 How far might this limit be extended, and what algorithmic or hardware progress can achieve that?

In this section, we give a separate account of each fundamental constraint that we identify and model.

4.1 Data movement

Distributed training can run up against data movement limits for two reasons: because we're moving too much data *inside an individual GPU* or *between different GPUs*. We consider each in turn.

4.1.1 Intra-GPU data Movement

A typical GPU with tensor cores can compute much faster than it can move data to and from DRAM: for example, an NVIDIA H100 SXM (NVIDIA, 2022) can perform a theoretical maximum of 2×10^{15} FLOP/s (1×10^{15} multiply-accumulates (MAC)/s) at 8-bit precision during dense matrix multiplications, but has a DRAM memory bandwidth of only 3.35 TB/s, for an **arithmetic intensity** (the ratio of the two) of ≈ 299 MAC/byte. An 8-bit matrix multiplication of shape $(M, K) \times (K, N) \to (M \times N)$ must perform MKN MAC and, under ideal caching, read the two input matrices and write the output matrix once each, for $MK + KN + MN$ bytes accessed total. For the H100's computational resources to be balanced, we thus must have $1/299 \approx (MK + KN + MN)/MNK = 1/M + 1/N + 1/K$. In the square case,

$$M = N = K \approx 896.$$

If the dimensions are substantially smaller, then the tensor cores must go underutilized as they will be bottlenecked on data movement to and from DRAM. Similar considerations apply at lower levels of the memory hierarchy as well. We discuss this further in Section 5.1 and Appendix 8.1.1.

Distributed training across more GPUs or experts requires splitting the problem and hence reducing at least one of M, N, K, worsening arithmetic intensity.

The main way to counter the degradation in arithmetic intensity is by increasing the dimensions of the matrix multiplications: at least one of d_{model}, d_{ff}, b (Section 2), either making the model fatter and shorter, or increasing the batch size. Scaling b runs into the critical batch size limit, and it's not clear how far scaling model width d_{model} and d_{ff} at the expense of model depth L can go while remaining near the compute-optimal frontier. We discuss these matters in greater detail in Sections 4.2 and 4.4. The broad conclusion is there may be no free lunch for arithmetic intensity.

4.1.2 Inter-GPU data movement

In addition to data movement inside individual GPUs, distributed training requires the movement of data between them. We quantified this cost in Section 3.

Each GPU has an upper bound to the rate of information it can receive or transmit per unit time. This means that asymptotically, the scaling of the total network bandwidth of a cluster can at most be linear in cluster size, even if the interconnect switches, wires, *et cetera* are costless. In practice, scaling is often *sublinear*. For instance, a high-bandwidth NVLink region may be limited to a single node: GPT-4 (OpenAI (2023)) used clusters of A100s (Patel & Wong, 2023) with an NVLink node size of 8. Increasing the tensor-parallel degree N_{TP} for such a cluster thus demands some fraction of bandwidth-hungry all-reduce communication over slower interconnects such as InfiniBand.

Even in an optimistic linear scaling regime where we do not have to rely in part on slower connections at other levels of the network hierarchy, scaling a cluster increases *both* the arithmetic throughput *and* the available network bandwidth proportionally, but the data movement *per GPU* increases as we scale the degrees of parallelism, as seen in Section 3. Thus if we hold the model and batch size constant and simply scale up the training cluster, arithmetic intensity with regard to the network must eventually shrink to the point that this communication becomes a bottleneck.

Even if we scale these dimensions, the question remains whether they scale fast enough relative to total training compute of the model, as N_{GPU} must be at least proportional to the training run compute given constant utilization and training duration. The answer depends on factors discussed in Sections 4.2 and 4.4.

4.2 The critical batch size

Because data and pipeline parallelism both involve slicing the data matrix along the batch dimension, increasing the batch size b helps with scaling up these two methods of parallelism without damaging arithmetic intensity. This is always doable, but only useful to the extent it reduces noise in the gradient estimate; at some scale, the estimate is precise enough that further noise reductions are not useful. (Shallue et al., 2019)

In an ideal scaling regime, n gradient steps with a batch size b and learning rate η should reduce loss equivalently to taking a single gradient step with a batch size $n \cdot b$ and learning rate $n \cdot \eta$ (McCandlish et al., 2018), so increasing the batch size effectively parallelizes the otherwise serial gradient descent steps. However, at some point, called the *critical batch size*, this favorable scaling rapidly hits diminishing returns.

McCandlish et al. (2018) conjecture, based on a second-order approximation, that a gradient step at batch size b should ideally reduce loss by

$$\Delta L = \frac{\Delta L_{\text{max}}}{1 + b_{\text{noise}}/b},$$

where ΔL_{max} is the infinite batch size limit, and b_{noise} is a "noise scale" at which the dependence of ΔL on b goes from linear to sub-linear. In this model, the critical batch size b_{crit} is equal to the noise scale b_{noise}.

They also conduct experiments and observe that models with smaller loss tend to have larger b_{noise} values, but holding loss constant, there is no impact of model size on b_{noise}. As model loss decreases throughout training, this also means that b_{noise} increases throughout training, allowing larger batch sizes to be used later on in a training run compared to what is efficient at initialization.

The intuition for this is straightforward when we're using cross-entropy loss with respect to a ground truth distribution: a model with a smaller Kullback-Leibler divergence with the ground truth distribution can only be distinguished from ground truth by drawing more samples. Knowing which direction to go in for a useful gradient step certainly requires detecting that the model is not already optimal, so models with a smaller reducible loss per token can accommodate larger batch sizes (in units of tokens). This information-theoretic argument also makes the quantitative prediction that the noise scale b_{noise} should vary inversely proportionally with the reducible loss of the model, i.e. the Kullback-Leibler divergence of the model with the ground truth distribution.

If this conjecture is correct, and the dependence of the critical batch size on a dense model's properties factors through its reducible loss, then we can leverage the Chinchilla scaling law from Hoffmann et al. (2022) to predict how the critical batch size should scale with the training compute of a dense model. As the reducible loss of a Chinchilla-optimal dense model falls off like $\propto 1/T^{\alpha}$ where T is the model's training compute and $\alpha \approx 1/6$, we would predict that the critical batch size should scale approximately with $T^{1/6}$.

On the other hand, Bi et al. (2024) claim to find the scaling relationship $b_{\text{crit}} \propto T^{0.33}$, although many of their models are overtrained relative to the Chinchilla law. For a fixed total training compute and batch size, an overtrained model must be *smaller* (posing less opportunity for tensor parallelism at high arithmetic intensity) and *train on more tokens*, hence use more serial steps, exacerbating latency bottlenecks (Section 4.3). If batch sizes can be scaled this aggressively even at the Chinchilla-optimal frontier, then opportunities for parallelism abound. Our baseline results employ the more conservative scaling assumption of $b_{\text{crit}} \propto T^{1/6}$, though we analyze robustness to this assumption in Appendix 8.3.

The above discussion relies on theoretical and empirical analysis of first-order optimization methods. While second-order methods have been investigated for years (Dauphin et al., 2014; Martens & Grosse, 2015; Yao et al., 2020; Liu et al., 2023), they pose significant scalability challenges: Full curvature information is normally intractable to calculate, so approximate and/or infrequent estimates must be used instead. Furthermore, the curvature (and its empirical estimate) can be very small or even negative and vary across the loss landscape, so that damping or trust-region approaches are required for stable learning. For these reasons, it remains unclear if second-order methods will ever be practical for frontier training runs. If they are, it's conceivable that critical batch sizes could greatly increase.

4.3 Latency

The training of a neural network involves some irreducible number of inherently serial operations, across model layers ($2L$ matrix multiplications in each direction in our toy model) and optimizer steps (with parallelism limits due to the critical batch size, as explained in Section 4.2).

If lower bounds exist on the latency of communication between GPUs and matrix multiplication inside a GPU, then the sequential bottleneck in neural network training sets a lower bound for the duration of the entire training run, per Amdahl's law. In practice, bandwidth limits often become binding before latency limits, but latency limits are nevertheless significant because they are more difficult to overcome.

For example, one can always increase the total amount of bandwidth in a cluster by obtaining more GPUs and more network interconnects: in the bandwidth-bound regime this will reduce the arithmetic utilization of the entire cluster, but overall throughput will still increase. A typical result here might be a decay of $\propto 1/N_{\text{GPU}}^{1/3}$ or $\propto 1/N_{\text{GPU}}^{1/4}$ in the overall utilization rate, in a 3D- or 4D-parallel regime respectively (Section 3). In contrast, if the training process is latency-bound, further cluster scaling will not yield any additional benefits at all and the utilization rate will decay as $\propto 1/N_{\text{GPU}}$.

The two important sources of latency relevant during distributed training are:

1. **Network latency:** Interconnects used in high-performance computing are optimized to have low latency, but even low-latency interconnects such as InfiniBand still require on the order of $1\ \mu$s for a single point-to-point communication. Within a single node, the interconnect of choice for NVIDIA GPUs is NVLink, and a typical latency of an all-reduce operation that uses NVLink within a single node of 8 GPUs is on the order $\approx 10\ \mu$s (Jeauguey (2018)).

2. **Intra-GPU latency:** Even when working with a single GPU, there are sources of latency that constrain how quickly matrix multiplications may be performed on a single device. We say more about this in Section 5.2, but to summarize, we empirically measure a floor on matrix multiplication latency on an A100 using cuBLAS at around $4.5\ \mu$s.

These latencies are much too small to matter for current training runs, as a typical duration for one matrix multiplication in a single GPU during the training of a current large language model is on the order of

1 to 10 milliseconds. For instance, Falcon-180B (Almazrouei et al., 2023) used $d_{\text{model}} = 12288$, $d_{\text{ff}} = 4d_{\text{model}}$,$N_{\text{TP, ff}} = 8$, $b = 2^{22}$ and $N_{\text{DP}} \cdot N_{\text{PP}} = 512$. If we assume e.g. 4 times as many microbatches as pipeline stages, their feedforward matrix multiplications on individual GPUs were of the shape $12288 \times 6144 \times 2048$. On an A100, such a matrix multiplication takes ≈ 1 ms at full utilization.

However, as models become larger they need to be trained on more tokens (Hoffmann et al., 2022) and the critical batch size might not increase proportionally (Section 4.2). In this case, the time taken per gradient step has to fall in order that the entire training run still complete within an acceptable duration, and for very large models it could in principle fall below the timescales at which latency constraints would become binding. In Section 5.2 we conclude that we have ≈ 4 OOM more room to scale the training compute of state-of-the-art models before latencies begin to hurt utilization.

4.4 Scaling the number of layers

The need to increase the number of sequential operations in a neural network as the network is scaled up can have adverse effects on how parallelizable the training run of such a model can be, as model layers are inherently serialized. The effect of this is mitigated to some extent by techniques that reduce pipeline bubble sizes (Section 3.2.1) and enable models to be effectively parallelized across the layer dimension. However, when network or GPU latency constraints become binding, avoiding additional serial steps in the forward and backward pass is critical for ensuring good utilization even when these techniques are used.

Unfortunately, there is little good research we've been able to find in the literature about what we call *shape scaling laws*: how the performance of a model depends on the balance between its widths (as measured by $d_{\text{model}}, d_{\text{ff}}$) and its depth (as measured by the number of layers L) (Kaplan et al., 2020; Alabdulmohsin et al., 2024). The best we can do is back out implicit laws used in other work, and due to the clean nature of their data and the large number of data points we pick Hoffmann et al. (2022) as our reference in this section. Figure 10 shows the results we obtain by analyzing the information about layer scaling in the models trained in Hoffmann et al. (2022), with the best-fitting scaling law $L \approx 3.67 \cdot (N_{\text{p}}/10^6)^{0.27}$ where L is the number of layers and N_{p} is the number of model parameters.

While this might be nothing more than a rule of thumb, we believe it is nevertheless an important finding because scaling a model by increasing the number of layers generally allows fewer opportunities for parallelism than increasing d_{model} or d_{ff}. This is because arithmetic intensity constrains the minimum dimensions of the individual matrix multiplications performed by a single device, and without scaling at least one of $d_{\text{model}}, d_{\text{ff}}$, or b the arithmetic intensity cannot effectively be improved. We will see the significance of this in Section 5.1.

5 Closed-form expressions for the biggest training run

In this section and the next, we provide answers to the questions raised in Section 4. We make some simplifications to derive analytic bounds here. After this preliminary analysis, we will introduce a complete model in Section 6 which takes into account all of the constraints that we have previously discussed and closely matches our analytic results.

5.1 The utilization cliff

Here we derive a closed-form expression upper-bounding the largest achievable training run, before data movement bottlenecks lower utilization substantially.

With even a modest degree of tensor parallelism, its bandwidth consumption greatly exceeds (Table 1) the maximum possible point-to-point communication from pipeline or expert parallelism (e.g. the case $N_{\text{PP}} \cdot i = L$), so we shall neglect their communication cost and take these at their maximum possible degrees

$N_{\text{PP}} = L$, $N_{\text{EP}} = E$, restricting ourselves to considering only the all-reduce communication from data and tensor parallelism, across $N_{\text{DP}} \times N_{\text{TP}}$ groups ("workers") of $N_{\text{PP}} \times N_{\text{EP}} = LE$ GPUs each.

Recalling our discussion of multi-dimensional slicing in Section 3.1.2, and taking $I = d_{\text{ff}}, K = d_{\text{model}}, J = b/E$ in Eq. 3, the all-reduce communication so imposed is equalized across all three dimensions and minimized when each worker has a cube-shaped unit of work $(d, d) \times (d, d) \to (d, d)$ for each matrix multiplication, with some side length

$$d = d_{\text{ff}}/N_{\text{TP, ff}} = d_{\text{model}}/N_{\text{TP, model}} = (b/E)/N_{\text{DP}}. \tag{6}$$

However, given our $N_{\text{PP}} = L$ pipeline stages, we may assume at least $m \approx 2N_{\text{PP}} = 2L$ microbatches, as this is the minimum to achieve the zero bubble condition under a ZB-H2 schedule (Section 3.2.1). This means each actual "physical" matrix multiplication onboard a single GPU corresponding to a single expert and pipeline stage uses a much smaller **nanobatch** size of[11]

$$b' = (b/E)/(N_{\text{DP}} \cdot m) = (b/E)/(N_{\text{DP}} \cdot 2L), \tag{7}$$

which is smaller than d by the factor $2L$ if Eq. 6 holds. Under normal circumstances, this is already a problem for intra-GPU data movement to and from high-bandwidth memory (Section 4.1.1), as this necessitates loading all of an expert's weights, and accumulating their gradients, once for each tiny nanobatch. However, as we are examining the limits to distributed training, we must consider the possibility that each of the N_{DP} model replicas contains enough individual GPUs such that these weights, along with their gradients, can fit entirely in their aggregate SRAM, perhaps even in register banks. In such a case, their movement is required only once per batch to all-reduce gradients across model replicas, rather than once per nanobatch.

Even so, we cannot normally make the data-parallel degree N_{DP} large enough to achieve the ideal of Eq. 6, but instead just large enough that Eq. 7 equals some hardware-specific lower bound b', which we will estimate later using two separate methods. Then our data-parallel degree is

$$N_{\text{DP}} = \frac{b}{b'} \cdot \frac{1}{2LE}. \tag{8}$$

Meanwhile, there is some minimum utilization-preserving weight submatrix size $d' \times d'$, which we will also estimate later. This makes our tensor-parallel degree $N_{\text{TP}} = N_{\text{TP, ff}} \times N_{\text{TP, model}}$ equal to

$$N_{\text{TP}} = \frac{d_{\text{ff}} d_{\text{model}}}{d'^2}. \tag{9}$$

As discussed above, we assume the maximum possible pipeline- and expert-parallel degrees $N_{\text{PP}} = L$, $N_{\text{EP}} = E$, and take N_{DP} and N_{TP} as in Eqs. 8 and 9 so as to achieve the critical computation volume $d'^2 b'$ per single-GPU matrix multiplication, with a total number of GPUs

$$N_{\text{critical}} = N_{\text{PP}} \cdot N_{\text{EP}} \cdot N_{\text{DP}} \cdot N_{\text{TP}} = \frac{1}{4LE} \cdot \frac{bN_{\text{p}}}{d'^2 b'}, \tag{10}$$

[11] The nanobatch size $b' = b/(E \cdot N_{\text{DP}} \cdot m)$ is the number of tokens seen at once by an individual GPU matrix multiplication kernel, taking into account routing into E separate experts (whether or not on separate GPUs), data-parallel sharding of degree N_{DP}, and sharding into m microbatches for pipeline paralellism.

where we have made use of the parameter count formula $N_\text{p} = 2LEd_\text{model}d_\text{ff}$ (Eq. 1).

Assuming that a cluster of the critical size N_critical can run at close to full utilization by overlapping data movement with computation, its compute output over a time period t_train is simply

$$T_\text{critical} = N_\text{critical} C t_\text{train}, \tag{11}$$

where C is the arithmetic throughput (MAC/second) of a single GPU at perfect utilization.

Bringing scaling laws into the argument, we now calculate the size of the model that can be trained in a given duration (e.g. 3 months) under the condition that utilization not seriously drop off. Chinchilla scaling laws (Hoffmann et al., 2022) give us a compute-optimal training dataset size of roughly $20N_\text{p}$ tokens, with each training token requiring 3 MAC per parameter, so that the total compute required to train a model with N_p parameters and sparsity factor E is roughly

$$T_\text{critical} = (60\,\text{MAC}) \cdot N_\text{p}^2/E. \tag{12}$$

Setting Eqs. 11 and 12 equal and substituting in Eq. 10 lets us solve for number of parameters:

$$N_\text{p} = \frac{b}{L} \cdot \frac{Ct_\text{train}}{(240\,\text{MAC}) \cdot d'^2 b'}.$$

Taking this formula for N_p in Eq. 12,

$$T_\text{critical} = \frac{1}{(960\,\text{MAC}) \cdot E} \left(\frac{b}{L} \cdot \frac{Ct_\text{train}}{d'^2 b'} \right)^2 \tag{13}$$

is thus an upper bound on the critical training compute at which a model with sparsity factor E, a batch size of b, and a depth of L MLP blocks can be trained over duration t_train on a cluster of GPUs whose specifications are held fixed, without loss of utilization. This formula is quadratically sensitive to most of its variables, and even quartically sensitive to the critical weight submatrix shape d', so any significant uncertainties can shift our estimate substantially. However, we can use it to place the utilization cliff within an order of magnitude or two.

In practice, we (weakly) expect the batch size b and number of MLP blocks L to scale similarly with training run size: specifically, batch size roughly as $b \propto T^{1/6} \propto N_\text{p}^{1/3}$ (Section 4.2) and layer count roughly as $L \propto N_\text{p}^{0.27}$ (Section 4.4). Since Eq. 13 depends on b and L only through their ratio, an approximation that they are constant may suffice. We will usually assume $b = 4 \times 10^6$ and $L = 100$, but explore variations on this assumption in Appendix 8.3, concluding that under more optimistic, but speculative, assumptions, the critical compute threshold T_critical might increase by about three orders of magnitude.

It is not clear how the sparsity factor E should be scaled. The most recent paper on scaling laws for mixture-of-experts (MoE) models known to us, (Krajewski et al., 2024), is agnostic on the question: the scaling law they derive does not incorporate E as a variable, instead using a fixed value $E = 64$ following recommendations from Clark et al. (2022). It's also unknown to what degree doing so also permits further compute-efficient scaling of the batch size b. It is apparent from Eq. 13 that if greater sparsity can enable larger batch sizes at a rate exceeding the square root of the sparsity factor, then sparsity permits larger training runs.

We'll now pursue two methods to estimate the critical nanobatch size b' and weight submatrix size $d' \times d'$, the first from latency bottlenecks, the second from bandwidth bottlenecks.

5.1.1 From latencies

Definitionally, b' is the number of tokens in flight at any stage of its processing on a single GPU. So one lower bound for b' and d' comes from the fact that a GPU's unit of work $d'^2 b'$ MAC for each of a nanobatch's matrix multiplications must take at least as long as the irreducible latencies from kernel launches and model- (i.e. tensor-, pipeline-, or expert-) parallel communication. If t_L is the timescale of this latency, we therefore have:

$$\frac{d'^2 b'}{C} \geq \frac{t_L}{\text{MAC}}.$$

Substituting into Eq. 13,

$$T \leq \frac{1\,\text{MAC}}{960 \cdot E} \left(\frac{b}{L} \cdot \frac{t_{\text{train}}}{t_L} \right)^2. \tag{14}$$

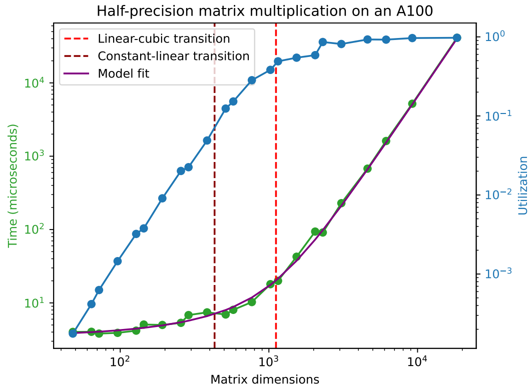


Figure 5: Default cuBLAS GEMM kernel performance on an A100 GPU. The green and blue curves show observed wall clock time and utilization, respectively. The purple curve shows a cubic polynomial fit to wall clock time, with dashed lines at the boundaries between a constant $\approx 4.5\ \mu$s latency-bound regime, a linear (likely occupancy-bound) regime, and a cubic compute-bound regime.

Figure 5 displays empirically observed matrix multiplication latencies on an A100, showing a $\approx 4.5\ \mu$s timescale below which the matrix multiplications cannot be accelerated due to various overheads of off-the-shelf cuBLAS kernels. Other GPUs are similar. Network latencies are also on the order of microseconds (Section 4.3) for an optimized all-reduce in which latency does not scale with the number of participants (e.g. a mesh network topology), so even a more optimized matrix multiplication kernel won't change our bottom line too much.[12] To account for both factors, we take $t_L = 9\ \mu$s. Then given a training duration t_{train} of 3 months, and typical values of $b = 4 \times 10^6$, $L = 100$, Eq. 14 for a dense model ($E = 1$) becomes

$$\boxed{\mathbf{T}_{\text{critical}} = \mathbf{3 \times 10^{30}}\ \text{FLOP.}} \tag{15}$$

We emphasize again that the quadratic sensitivity to uncertainty makes this an approximate estimate (within one or two orders of magnitude).

A discussion of possible methods to reduce t_L is beyond our scope, but we note that a reduction by one order of magnitude, to 900 ns, would put it at only several times typical DRAM access latencies, despite the need for an all-reduce operation across inter-node interconnects, so this is likely a generous lower bound given anything like present interconnect technology. Given the quadratic role of t_L in Eq. 14, improved interconnects may therefore permit at most two orders of magnitude of further training compute.

[12] This does assume some network latency is incurred on a large fraction of MLP blocks, which is true so long as any tensor or expert parallelism is employed, or any significant degree of pipeline parallelism is employed. A purely data-parallel cluster would fail to fit model weights in DRAM, unless fully-sharded (Rajbhandari et al., 2020), which would re-introduce network latencies at most layers.

5.1.2 From bandwidth

Alternatively, we can consider utilization-preserving minimums imposed on b' and d' by network and memory bandwidth. Each GPU, for each nanobatch, performs two physical matrix multiplications on the forward pass of shape $(d', d') \times (d', b') \to (d', b')$, one for each layer of the expert and MLP block for which it's responsible. This entails $d'^2 b'$ MAC twice in the forward pass and four times in the backward pass (Section 2), for $6d'^2 b'$ MAC in total arithmetic. Furthermore, it participates in a tensor-parallel all-reduce across $N_{\text{TP, ff}}$ or $N_{\text{TP, model}}$ peers of the $d' \times b'$ activation matrix (or its gradient) after each such multiplication, in both the forward and backward pass, receiving approximately $2d'b'$ words (Eq. 2) for each of four such all-reduces, or $8d'b'$ words total. This is balanced when

$$\frac{6d'^2 b' \text{ MAC}}{C} = \frac{8d'b'}{B_{\text{net}}},$$

where C is the GPU's arithmetic throughput in MAC per second, and B_{net} its unidirectional network bandwidth in words per second. This is solved by

$$d' = \frac{4C}{3B_{\text{net}}} \cdot \frac{1}{\text{MAC}}. \tag{16}$$

How large is the critical nanobatch size b'? If the weight submatrices (and their gradients) fit in SRAM, the usual reason to require a large nanobatch, i.e. controlling DRAM bandwidth repeatedly accessing weights and their gradients (Section 4.1.1), does not apply. This can potentially occur when the aggregate SRAM of a single model replica exceeds twice the model size:

$$\text{weights in SRAM} \implies \frac{N_{\text{GPU}}}{N_{\text{DP}}} \cdot S \geq 2N_p,$$

where S is the single-GPU SRAM word capacity. Using the substitution $N_{\text{GPU}}/N_{\text{DP}} = N_{\text{PP}} \cdot N_{\text{EP}} \cdot N_{\text{TP}}$, our choices above $N_{\text{PP}} = L$, $N_{\text{EP}} = E$, the tensor-parallel bandwidth bottleneck (Eq. 9) $N_{\text{TP}} = d_{\text{ff}} d_{\text{model}}/d'^2$, and the parameter count formula (Eq. 1) $N_{\text{p}} = 2LEd_{\text{model}}d_{\text{ff}}$, we cancel N_{p} from both sides to get:

$$\text{weights in SRAM} \implies \frac{S}{d'^2} \geq 4.$$

In this case, we set an aggressive lower bound of

$$b' = 16, \qquad \text{(if weights in SRAM)}$$

as tensor core instructions typically require each input dimension to be at least 8, and we assume at least twice as many tokens per nanobatch to support double-buffered data movement, ensuring that the tensor cores are well fed.

Otherwise, we make use of the fact that a typical weight gradient accumulation step entails $b'd'^2$ MAC and d'^2 DRAM accesses in both directions: once to read the previously-accumulated gradient, and once to write the new gradient. Canceling d'^2 from both sides, the DRAM bottleneck implies:

$$b' = \frac{C}{B_{\text{DRAM}}} \cdot \frac{1}{\text{MAC}}, \qquad \text{(if weights in DRAM)}$$

where B_{DRAM} is the "unidirectional" DRAM bandwidth (half the total bandwidth) in words per second.

Applying Eq. 13 with these formulas, we now estimate the largest possible utilization-preserving training run for recent NVIDIA systems, considering only bandwidth bottlenecks. We do the analysis with a twist: at the *node* level, treating the whole node as a single "GPU". This allows us to identify the tightest constraints since the inter-node bandwidth bottleneck (InfiniBand except in the case of an H100 SuperPOD) is most severe. The results are in Table 2. In most cases, the T_{critical} calculated by this method is two to three orders of magnitude below that given by the latency method, making bandwidth the operative constraint on utilization, and leaving only about two orders of magnitude of headroom above today's largest training runs (Epoch AI, 2024).

The increased inter-node bandwidth in a SuperPOD allows for a factor of ≈ 20 increase in tensor parallelism, which in turn allows model replicas to be large enough to fit weights and their gradients in SRAM, bypassing DRAM bandwidth bottlenecks and greatly reducing the nanobatch size. As such, the operative constraint in this case is latency (Eq. 15).

5.2 The absolute limit

In the previous subsection, we pursued one approach to answering the first question from Section 4, identifying the limits to scale imposed by latencies and bandwidth, assuming good utilization. Aggregate bandwidth and arithmetic throughput can in principle be scaled arbitrarily far, though perhaps with large decreases in utilization. But latency imposes hard limits to scale regardless of utilization. We consider those limits here.

If t_L is the latency of one matrix multiplication as in Section 5.1.1, then a model forward and backward pass must take at least $4Lt_L$ time no matter how many GPUs we have in the training cluster, for the two matrix multiplications required per each of L MLP blocks in each direction. For a batch size b and Chinchilla-optimal dataset size $D = 20N_{\text{p}}$, this means the training duration must be at least $4Lt_L \cdot (D/b) = 80N_{\text{p}}Lt_L/b$. Setting this equal to the training timescale t_{train} and solving for N_{p},

$$N_{\text{p}} = \frac{b}{L} \cdot \frac{t_{\text{train}}}{80t_L} \tag{17}$$

is the greatest number of parameters that can be trained over duration t_{train}.

	Arithmetic	**Unidir. bandwidth (words/s)**		**SRAM**			$\mathbf{T}_{\text{critical}}$
GPU	**(MAC/s)**	**Network**	**DRAM**	**(words)**	d'	b'	**(FLOP)**
DGX-1 (V100)	5.00×10^{14}	2.5×10^{10}	1.8×10^{12}	151M	26.7k	278	1×10^{27}
DGX A100	1.25×10^{15}	1.0×10^{11}	3.1×10^{12}	366M	16.7k	401	3×10^{28}
DGX H100	3.96×10^{15}	2.0×10^{11}	6.7×10^{12}	487M	26.4k	591	2×10^{28}
" in SuperPOD	"	9.0×10^{11}	"	"	5.9k	16	1×10^{34}

Table 2: FP16 specs of recent NVIDIA systems (NVIDIA, 2019; 2020; 2022), along with their corresponding critical weight matrix shape $d' \times d'$ as from Eq. 16 and critical nanobatch size b' (depending on whether weights and their gradients might fit in SRAM), and maximum dense ($E = 1$) three-month training run size T_{critical} as from Eq. 13, assuming $b = 4 \cdot 10^6$, $L = 100$ and considering only bandwidth bottlenecks.

Employing once again the Chinchilla equation 12 for the total training compute, and substituting Eq. 17,

$$T_{\text{limit}} = \frac{3\,\text{MAC}}{320 \cdot E} \left(\frac{b}{L} \cdot \frac{t_{\text{train}}}{t_L} \right)^2 \tag{18}$$

is the total compute for this latency-limited training run. Interestingly, this is exactly nine times as large as Eq. 14, the latency-imposed limit on *utilization-preserving* training runs. This factor of nine is attributable to the replacement of an *efficient* pipeline schedule with a *latency-minimizing* pipeline schedule.

Taking our usual values $b = 4 \times 10^6$, $L = 100$, $t_{\text{train}} = 3$ months, $t_L = 9\ \mu$s gives

$$\boxed{\begin{aligned} \mathbf{N}_{\text{p}} &= \mathbf{4 \times 10^{14}}, \\ \mathbf{T}_{\text{limit}} &= \mathbf{2 \times 10^{31}}\ \text{FLOP} \end{aligned}}$$

as the number of parameters and training compute of the largest dense model that can be trained over t_{train} duration, even at low utilization. We emphasize yet again that the quadratic sensitivity to uncertainty makes this an approximate estimate.

As in Section 5.1, this estimate depends on b and L through their ratio. In Appendix 8.3, we consider an optimistic scenario where the batch size can be scaled much faster than the layer count, in which T_{limit} could be as high as 3×10^{36} FLOP, which is so large that a training run of that size would exceed annual world primary energy consumption. Whether latencies are of practical relevance or not, therefore, may heavily depend on how aggressively batch sizes can be scaled relative to the layer count of large models.

6 Complete model and results

6.1 Model specification

Our model has the following pieces:

1. Assumptions about how the critical batch size b, the training dataset size D, and the model dimensions $d_{\text{model}}, d_{\text{ff}}, E, L$ should scale as models are increased in size.

2. A model of matrix multiplications on a single GPU. This model incorporates HBM, L2, and L1 memory bandwidth bottlenecks, and the resulting under-utilization of streaming multiprocessors on NVIDIA GPUs when the matrix multiplications become unusually small.

3. A hierarchical network description. We ignore network topology for the most part but take into account that different levels in a network hierarchy (e.g. intra-node and inter-node) will have different bandwidths and latencies at which they can facilitate communication across GPUs. For instance, such a description could look like "we have an NVLink node size of 8, the NVLink bandwidth is 600 GB/s bidirectional per GPU with 10 μs one-way latency, and inter-node communication takes place at 50 GB/s bidirectional per GPU with 5 μs one-way latency."

4. A method that calculates how long a training run of a specific model parallelized in a specific way across a cluster of N_{GPU} GPUs will take. Here, we take into account parameters such as the pipeline interleaving factor (Section 3.2.1), whether zero-bubble pipeline parallelism is used (Section 3.2.1), and how the 5D parallelism setup is divided across different levels of the network hierarchy.

5. An optimizer that searches over all possible ways of parallelizing the training of a model across a fixed number N_{GPU} of GPUs and finds the one that takes the least time. If multiple strategies take minimal time, we break the symmetry by picking the one that has the least network communication time, even if this time can be hidden behind arithmetic.

6. A search function that takes as input the training compute cost of a model, uses the scaling relations from (1) to infer its shape and critical batch size, looks for the smallest possible cluster size N_{GPU} satisfying certain divisibility constraints that (5) thinks can train this model in less than a given time t_{train}. Usually, we take t_{train} to be 3 months.

The full model therefore enables us to answer questions such as "if we wished to train a 10^{27} FLOP model in 4 months or less, what's the smallest number of GPUs we could accomplish this with, and what would be the parallelism setup minimizing training time and network communication costs?"

There are also effects that we omit:

- GPU and node failures are ignored. We assume a cluster of idealized GPUs that never fail and that can always do computations and communication at constant rates. This biases our conclusions in an optimistic direction, as in practice the challenges of managing GPU failures in clusters at the million GPU scale or above are nontrivial but likely manageable.

- Activation recomputation is assumed unnecessary, as very large training runs typically require large clusters with more than enough memory to avoid it.

For a fuller description of our model, see Appendix 8.1. The code we use to simulate it can also be found at this GitHub repository.

6.2 Results for current hardware and algorithms

We first simulate what we call our "baseline scenario", considering DGX-1 (V100), DGX A100, and DGX H100 nodes under their datasheet configurations (NVIDIA, 2019; 2020; 2022), but with GPU clocks adjusted for expected thermal throttling based on empirical experiments with each GPU. Above the single-node NVLink network, there is a flat InfiniBand network.

The scaling relationships we use for these runs can be found in Appendix 8.1.4. The exact relations have a large influence on our results, as we've already seen in the closed-form results from Section 5.

	For dense models	**For sparse models**
V100 SXM	3×10^{27} FLOP	2×10^{27} FLOP
A100 SXM	3×10^{28} FLOP	2×10^{29} FLOP
H100 SXM	2×10^{28} FLOP	7×10^{28} FLOP

Table 3: A table summarizing when linear scaling of FLOP throughput with three month training run size stops in different cases, defined to be the point when model FLOP utilization (MFU; Chowdhery et al. (2022)) falls below 80% of the utilization that a single GPU can achieve in sustained use.

Table 3 and Figure 6 show the results of these simulations, which closely match our analytic results in Table 2. Overall, **linear scaling breaks down around the 10^{27} to 10^{29} FLOP scale**. In addition, as calculated

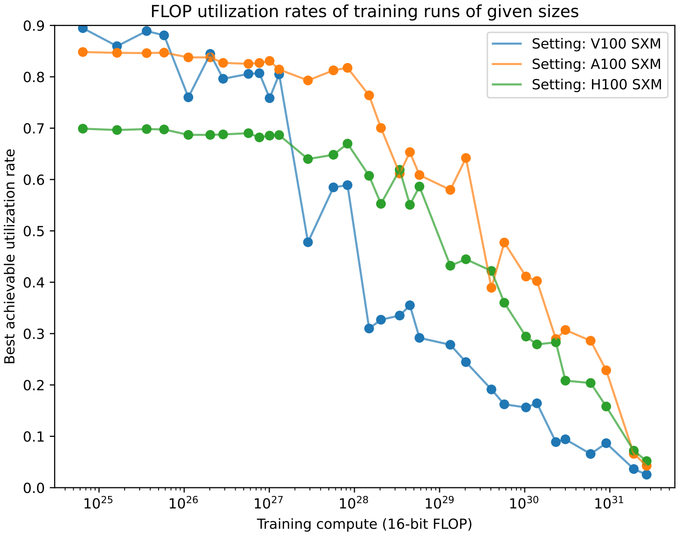


Figure 6: The MFU that's achieved by dense three-month training runs of different sizes when the cluster used to train them is made up of V100, A100, or H100 DGX nodes.

in Section 5.2, **training runs past the 2×10^{31} FLOP scale are impossible due to latency constraints**, which is why the curves stop at that point.

Figure 7 examines how training runs at different scales are parallelized. In general, as explained in Section 3.3, it turns out to be optimal to scale all parallelism degrees together for large-scale training runs.

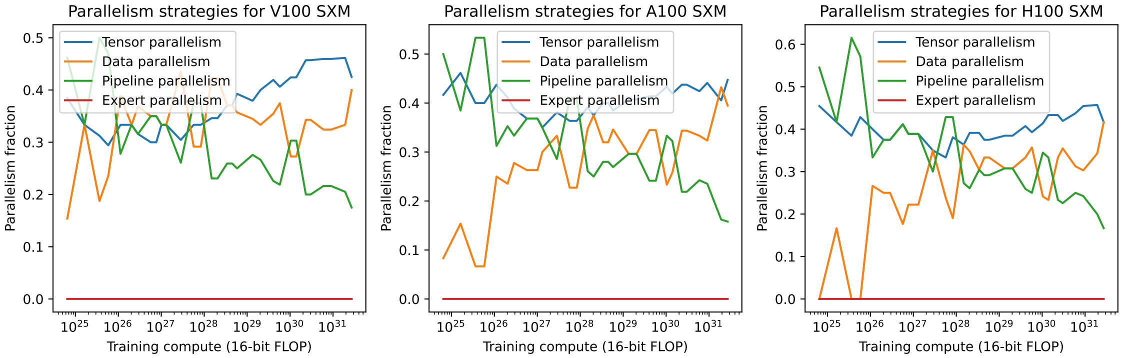


Figure 7: The fraction of overall parallelism that's dedicated to each parallelism strategy, for dense three-month training runs. The fraction for method X is computed as $\log(N_X)/\log(N_{\text{GPU}})$, i.e. the base N_{GPU} logarithm of N_X. The fractions add up to 1 because the different parallelism degrees must multiply to the overall cluster size.

A few important phenomena that we've observed in simulations:

- When memory bandwidth is the bottleneck, tensor parallelism is advantaged over data and pipeline parallelism due to its ability to slice along two dimensions at once.

- When network bandwidth is the bottleneck, pipeline parallelism is advantaged at small scales because it requires the least communication per parallelism degree. However, pipeline parallelism is costly due to pipeline bubbles unless a zero bubble scheme is used. At large scales, tensor parallelism is advantaged instead, because of the square root type scaling of communication costs derived in Section 3.1.4.

- When network latency is the bottleneck, data parallelism is advantaged over tensor and pipeline parallelism because it incurs the network latency cost twice per batch instead of multiple times per layer or once per pipeline stage. In addition, 2D tensor parallelism is often worse than 1D tensor parallelism because it incurs twice the network latency cost.

6.3 Extending the linear scaling regime

We've seen in Section 6.2 that using current hardware, it's not possible to extend the linear scaling regime past the 10^{29} FLOP scale under our baseline scaling assumptions for dense models. This answers the first question we raised in the introduction. Now, we turn to answering the second question: what technological improvements are needed to extend the linear scaling regime past this scale?

There are two ways this can be achieved: by improving the hardware, or by improving the software (e.g. by finding ways to switch to more favorable scaling relations). We consider each in turn.

6.3.1 Improving hardware

Because the 10^{31} FLOP scale is a limit set by latency constraints, going past this scale requires improving intra-GPU and network latency. Simply improving memory and network bandwidth is insufficient. However, there's still 3 OOM of room between 10^{28} FLOP and 10^{31} FLOP, and the linear scaling regime can be extended in this limited window by increases in network or memory bandwidth, without any improvements to latency.

Figure 8 shows the effect of relaxing different bottlenecks on utilization. Here, the 5 lines shown in the plot correspond to the following 5 sets of assumptions:

1. **H100 SXM:** The baseline DGX H100 setup that consists of nodes of 8 GPUs connected over NVLink and different nodes connected over InfiniBand.

2. **H100 SXM Low Latency:** Same as (1), except with all sources of latency (network and intra-GPU) divided by ten, which is perhaps the maximum reduction achievable without a major technological paradigm shift, as this puts inter-GPU latency at only several times the scale of a DRAM memory access.

3. **H100 SXM Global NVLink:** A hypothetical network technology in which arbitrarily many H100s can be linked together at NVLink bandwidths.

4. **H100 SXM Global NVLink and LL:** Same as (3), except with all sources of latency divided by ten.

5. **H100 SXM Infinite Network and LL:** Same as (4), except network bandwidth is assumed infinite.

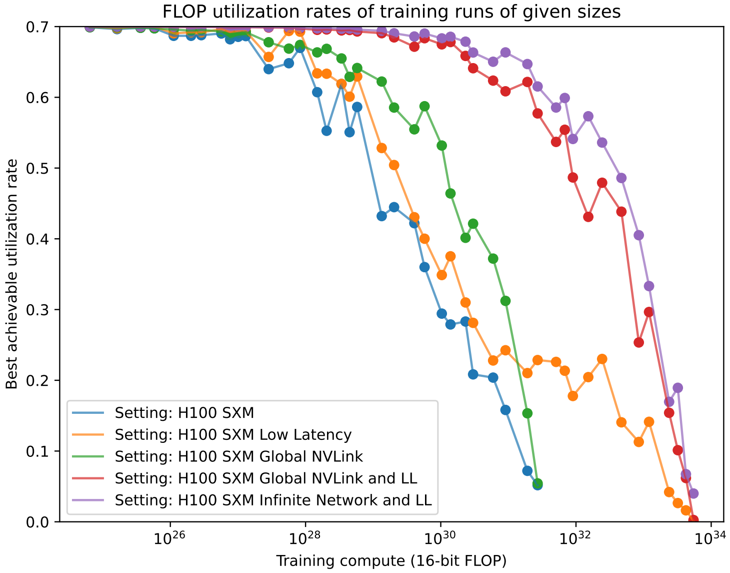


Figure 8: Utilization achieved by DGX H100s equipped with different hardware technologies.

The end of linear scaling for each case is listed in Table 4, which also includes this information for sparse training runs. As usual, we emphasize that real-world results will differ somewhat as they are sensitive to exact achieved latencies and bandwidth, as well as assumptions about model depth and batch size scaling.

	For dense models	**For sparse models**
H100 SXM	2×10^{28} FLOP	7×10^{28} FLOP
H100 SXM Low Latency	1×10^{29} FLOP	7×10^{28} FLOP
H100 SXM Global NVLink	4×10^{29} FLOP	7×10^{29} FLOP
H100 SXM Global NVLink and LL	5×10^{31} FLOP	1×10^{32} FLOP
H100 SXM Infinite Network and LL	9×10^{31} FLOP	6×10^{32} FLOP

Table 4: The training run scale at which linear scaling of FLOP throughput with cluster size stops under different assumptions, defined to be the point at which the hardware utilization achieved in a training run falls below 80% of the utilization that a GPU can achieve in sustained use.

Figure 8 illustrates that a significant expansion of the linear scaling regime cannot be achieved solely by a reduction in latency or relative increase in network bandwidth, but requires their combination. With the (perhaps unrealistically) optimistic assumption of a $10\times$ reduction in latency and global NVLink-like bandwidth, the linear scaling regime is pushed out just over three orders of magnitude, to around 5×10^{31} FLOP.

Even 10^{30} FLOP is a very substantial compute budget: at September 2024 rental prices of around 3.5 USD per hour for the H100 SXM5 (Nebius, 2024), such a training run would cost over a trillion dollars. Our results can therefore also be interpreted in a more favorable light for the prospects of continued scaling: it may be possible to reach the 10^{30} FLOP scale at good utilization, where economic rather than data movement considerations may dominate.

6.3.2 Improving algorithms

Algorithmic improvements offer another way to keep scaling past 10^{28} FLOP. In Section 5, we saw that the achievable training compute scales proportionally to $(b/L)^2$ where b is the global batch size (in tokens) and L is the number of MLP blocks in the model. While we assume a constant b/L ratio in that section, Appendix 8.3 considers the variable case and notes that changing the default batch size scaling law to the one from Bi et al. (2024) while holding the layer count scaling law fixed raises the achievable training compute by about three orders of magnitude.

This result is slightly more pronounced when we consider the full range of effects in our complete model, as shown in Figure 9: quantitatively, the shift from default scaling to DeepSeek scaling raises the point at which linear scaling stops by 5.2 OOM, from a training compute of 2×10^{28} FLOP to 3×10^{33} FLOP. The "no scaling" option in the plot corresponds to a fixed global batch size of 2^{22} tokens.

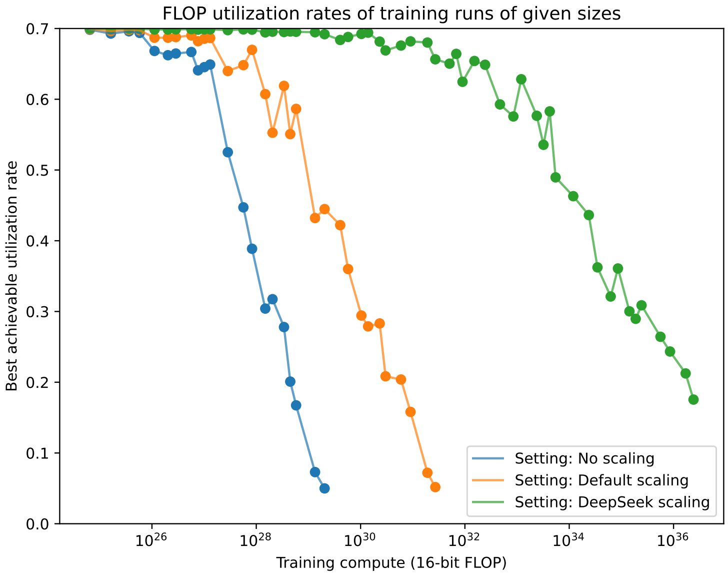


Figure 9: The H100 SXM utilization achieved by three-month dense training runs, under different critical batch size scaling laws.

7 Conclusion

We now return to the two guiding questions we've set out to answer in this work. We repeat them once again for the sake of convenience:

Q1 Given present-day algorithms, GPUs, and interconnects, what is the biggest training run that can be performed within a fixed duration, before intra- and inter-GPU data movement starts to seriously worsen utilization or even render it impossible?

Q2 How far might this limit be extended, and what algorithmic or hardware progress can achieve that?

Briefly summarizing our answers to these questions:

1. Our best guess is $\approx 2 \times 10^{28}$ FLOP. With aggressive batch size scaling (Bi et al., 2024) this number could go up to $\approx 3 \times 10^{33}$ FLOP, but we consider this to be rather optimistic. We think how quickly critical batch sizes can be scaled along with model sizes is a crucial question that has received little attention compared to its importance for continued scaling.

2. In the future, improved interconnect bandwidth and latency might extend the linear scaling regime by at most two orders of magnitude. However, improvements in machine learning algorithms that allow for faster batch size scaling or shorter, fatter models have the potential to allow many more orders of magnitude of improvement.

The recent turn of leading AI labs toward secrecy has made it difficult to acquire reliable information about algorithmic developments, which means much of the discussion of algorithmic issues in this paper has been speculative by necessity. It should be possible to substantially improve on our analysis with better information about:

1. **Critical batch size scaling.** A better understanding of critical batch sizes, discussed in McCandlish et al. (2018), and how they scale with model size or reducible loss is essential. This is because making the batch size larger creates more opportunities for parallelism and reduces the number of gradient steps that must be taken in a single epoch during training. Compared to its importance, this question has received very little interest from researchers.

2. **Model depth scaling and aspect ratio scaling laws.** Much of our work is based on the Chinchilla scaling law pioneered by Hoffmann et al. (2022), which has proven very useful in understanding the limits to distributed training better. However, despite its utility, this scaling law does not make any predictions about how model depth ought to be scaled with model size and how much of a performance loss results from a sub-optimal choice. Because the model depth controls the number of sequential operations in a single forward pass, it has a big influence on how easy distributed training is at large scales. Consequently, we think the nature of the trade-off between model depth and model width deserves a more thorough investigation than we've been able to find in the literature.

3. **Sparse model scaling laws.** Another shortcoming of the Chinchilla scaling law from Hoffmann et al. (2022) is that it only applies to dense models. With the growing popularity of mixture-of-experts models such as GPT-4 (OpenAI, 2023) and Mixtral (Jiang et al., 2024a), it has become important to understand the scaling laws of these models better as well. A useful start would be to check if the training compute-optimal dataset size of a sparse model scales approximately linearly with the total number of model parameters or not. There is also the possibility for interaction between sparsity and critical batch size.

Acknowledgments

We are grateful to Hailey Schoelkopf, Quentin Anthony, Daniel Ziegler, Deepak Narayanan, and Finbarr Timbers for their comments on an earlier draft of this paper; Robert Sandler for his assistance with illustrations; Jaime Sevilla and Tamay Besiroglu for comments on the paper and administrative assistance; Edu Roldan and Andrew Lucas for their work on the interactive tool; Maria de la Lama and Simon Jarvis for their assistance with publication; Mauricio Baker, Nicholas Brown, Felipe Calero Forero, Gabriel Kulp, Ashley Lin, and Aris Richardson for comments on a late draft; and Sihao Huang for his insights on hardware.

David Schneider-Joseph is very thankful to Lennart Heim for his mentorship and collaboration, as well as comments on this work; the Astra Fellowship under which some of this work took place; and the many great people at Constellation for thought-provoking discussions.

REFERENCES

Ibrahim Alabdulmohsin, Xiaohua Zhai, Alexander Kolesnikov, and Lucas Beyer. Getting ViT in shape: Scaling laws for compute-optimal model design, 2024. URL https://arxiv.org/abs/2305.13035.

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. The Falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, A. X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. DeepSeek LLM: Scaling open-source language models with longtermism, 2024.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with pathways, 2022.

Aidan Clark, Diego de las Casas, Aurelia Guy, Arthur Mensch, Michela Paganini, Jordan Hoffmann, Bogdan Damoc, Blake Hechtman, Trevor Cai, Sebastian Borgeaud, George van den Driessche, Eliza Rutherford, Tom Hennigan, Matthew Johnson, Katie Millican, Albin Cassirer, Chris Jones, Elena Buchatskaya, David Budden, Laurent Sifre, Simon Osindero, Oriol Vinyals, Jack Rae, Erich Elsen, Koray Kavukcuoglu, and Karen Simonyan. Unified scaling laws for routed language models, 2022. URL https://arxiv.org/abs/2202.01169.

Yann N. Dauphin, Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR*, abs/1406.2572, 2014. URL http://arxiv.org/abs/1406.2572.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 2019. URL https://arxiv.org/abs/1810.04805.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Epoch AI. Data on notable AI models, 2024. URL https://epochai.org/data/notable-ai-models. Accessed: 2024-10-09.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022. URL https://arxiv.org/abs/2101.03961.

Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pp. 1–10. IEEE, 2016.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory F. Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *CoRR*, abs/1712.00409, 2017. URL http://arxiv.org/abs/1712.00409.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

Sylvain Jeauguey. Multi-GPU training with NCCL. URL: https://on-demand.gputechconf.com/gtc/2018/presentation/s8462-multi-gpu-training-with-nccl.pdf, 2018. Presented at NVIDIA GTC 2018.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024a.

Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs, 2024b.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Jakub Krajewski, Jan Ludziejewski, Kamil Adamczewski, Maciej Pióro, Michał Krutul, Szymon Antoniak, Kamil Ciebiera, Krystian Król, Tomasz Odrzygóźdź, Piotr Sankowski, et al. Scaling laws for fine-grained mixture of experts. *arXiv preprint arXiv:2402.07871*, 2024.

Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. BASE layers: Simplifying training of large, sparse models. *CoRR*, abs/2103.16716, 2021. URL https://arxiv.org/abs/2103.16716.

Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. *CoRR*, abs/2305.14342, 2023. doi: 10.48550/ARXIV.2305.14342. URL https://doi.org/10.48550/arXiv.2305.14342.

James Martens and Roger B. Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. *CoRR*, abs/1503.05671, 2015. URL http://arxiv.org/abs/1503.05671.

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training, 2018.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

Nebius. GPU prices: Nvidia H100, A100, 2024. URL https://web.archive.org/web/20240915193540/https://nebius.ai/prices. Accessed: 21 October 2024.

NVIDIA. Nvidia DGX-1: The essential instrument of AI research. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-1-rhel-datasheet-nvidia-us-808336-r3-web.pdf, 2019. Accessed: 2024-10-18.

NVIDIA. NVIDIA DGX A100, 2020. URL https://resources.nvidia.com/en-us-dgx-systems/dgx-ai.

NVIDIA. NVIDIA DGX H100, 2022. URL https://resources.nvidia.com/en-us-dgx-systems/ai-enterprise-dgx.

NVIDIA. NVIDIA H100 tensor core GPU architecture. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper, 2022. Accessed: 2024-10-09.

NVIDIA. CUTLASS documentation. https://github.com/NVIDIA/cutlass/wiki/Documentation, 2024. Accessed: 2024-10-09.

OpenAI. GPT-4 technical report, 2023.

Dylan Patel and Gerald Wong. GPT-4 architecture, infrastructure, training dataset, costs, vision, MoE, 7 2023. URL https://www.semianalysis.com/p/gpt-4-architecture-infrastructure?utm_campaign=post&utm_medium=web.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, et al. RWKV: Reinventing RNNs for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism. *CoRR*, abs/2401.10241, 2024. doi: 10.48550/ARXIV.2401.10241. URL https://doi.org/10.48550/arXiv.2401.10241.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning, 2022.

Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training, 2019. URL https://arxiv.org/abs/1811.03600.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 billion parameter autoregressive language model, 2021.

Lilian Weng and Greg Brockman. Techniques for training large neural networks, 2022. URL https://openai.com/research/techniques-for-training-large-neural-networks. Accessed: 2023-12-01.

Zhewei Yao, Amir Gholami, Sheng Shen, Kurt Keutzer, and Michael W. Mahoney. ADAHESSIAN: an adaptive second order optimizer for machine learning. *CoRR*, abs/2006.00719, 2020. URL https://arxiv.org/abs/2006.00719.

Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Y. Zhao, Andrew M. Dai, Zhifeng Chen, Quoc Le, and James Laudon. Mixture-of-experts with expert choice routing. *CoRR*, abs/2202.09368, 2022. URL https://arxiv.org/abs/2202.09368.

8 Appendices

8.1 Appendix: Detailed model description

8.1.1 Matrix multiplication on a single device

A naive model of matrix multiplication on one device is to assume perfect utilization, in which case the time taken for a matrix multiplication of dimensions $m \times k \times n$ is mkn/C where C is the MAC per second that the device can perform. Our model improves on this in three ways:

1. We consider the memory bandwidth required for the matrix multiplication at different levels of the memory hierarchy. A big matrix needs to be read from HBM to L2, from L2 to shared memory (physically the same SRAM as L1 cache), and from shared memory into the registers. If communication is perfectly overlapped with computation inside a single device, the time taken for the matrix multiplication will be the maximum of the time taken for the arithmetic and the time taken for the data movement at all levels.

2. There is a minimum latency to each matrix multiplication due to setup and teardown of the CUDA kernel along with memory access latency.

3. Due to thermal throttling, devices often fall short of the peak boosted clock speeds reported in official data sheets. We correct this by performing empirical benchmarks on various GPUs to observe what clock speeds devices can sustain during realistic workloads.

The complicated part of the model is (1). To understand how we model this, consider the memory hierarchy levels involved in performing a matrix multiplication on an NVIDIA GPU using a typical kernel design, in which responsibility for calculating the output matrix is successively tiled into smaller sub-rectangles at the streaming multiprocessor (SM) and processing block (executing one or more "warps" of 32 threads each) level. Because of the correspondingly smaller output matrix size at each smaller level, the IO intensity of the input reads is increased. In detail (NVIDIA, 2024):

1. The input matrices must be read from HBM into L2.

2. The necessary pieces of the input matrices for an SM's output tile are loaded from L2 (or distributed shared memory in later GPU generations (NVIDIA, 2022) if another SM has already loaded the relevant data) into that SM's shared memory. This and other data movement typically happens asynchronously and double-buffered so as to overlap with computation.

3. Each of the four processing blocks on an SM performs coalesced reads of the input data from the SM's shared memory into its threads' registers.

4. Small chunks of input data move from the processing blocks' register banks to the tensor core, where the arithmetic of a warp-level matrix multiply-accumulate operation is performed, and the result accumulated in registers in that processing block which have been pre-allocated for the corresponding tile of the output matrix.

5. After many repetitions of the above steps, all arithmetic has been performed and the output tiles fully accumulated. The result is finally written back through the memory hierarchy and into HBM.

In the above sequence, each step has an input-output intensity which depends on the extent to which the tiled matrix has to be tiled at that level in the hierarchy. Other sequences are possible, based on different tiling strategies. We always assume tiling by the weight (or weight gradient) matrix dimensions, even if this is not the output matrix, as this usually allows for the smallest memory intensity. We calculate this intensity at the L2, SM, and warp level of the hierarchy, which tells us how much data movement is required at each level for the amount of computation involved in the matrix multiplication. We divide these by associated memory bandwidths obtained from official documentation or micro-benchmarks to estimate how much time is required for communication at each level. Taking a maximum of these values and the arithmetic time, then adding kernel setup/teardown latency, tells us how long the matrix multiplication takes.

8.1.2 Network communication

A straightforward communication model for a network with a single level of hierarchy proceeds by taking the quantitative estimates from Section 3. For a training step taken on a single batch, we can estimate the following network communication costs:

- **Data parallelism:** N_{p} words all-reduced across N_{DP} ranks per batch.
- **Tensor parallelism:** d_{ff} words all-reduced across $N_{\text{TP, model}}$ ranks and d_{model} words all-reduced across $N_{\text{TP, ff}}$ ranks twice per layer per token, once each for the forward and the backward pass.
- **Pipeline parallelism:** d_{model} words communicated point-to-point $2(N_{\text{PP}} \cdot i - 1)$ times per token, where i is the pipeline interleaving factor, and the factor of two counts both the forward and the backward pass.
- **Expert parallelism:** d_{model} additional words communicated point-to-point an average of $2 \cdot (1 - 1/N_{\text{EP}}) \cdot (L - N_{\text{PP}} \cdot i)$ times per token. The factor of $1 - 1/N_{\text{EP}}$ is the probability that a request must be routed to a different expert than the current one, while $(L - N_{\text{PP}} \cdot i)$ is the number of layer boundaries that are not pipeline communication boundaries.

Letting the bidirectional data movement costs per batch be denoted by $M_{\text{DP}}, M_{\text{TP}}, M_{\text{PP}}, M_{\text{EP}}$ respectively, and assuming that we have a per-GPU bidirectional network bandwidth of B_{network}, we can calculate the total time necessary for inter-GPU data movement as

$$t_{\text{network}} = t_{\text{network, DP}} + t_{\text{network, TP}} + t_{\text{network, PP}} + t_{\text{network, EP}} = \frac{M_{\text{DP}} + M_{\text{TP}} + M_{\text{PP}} + M_{\text{EP}}}{N_{\text{GPU}} \cdot B_{\text{network}}}$$

We assume that this communication, except for that imposed by data parallelism or during pipeline startup and clearing, can overlap with GPU arithmetic.

When there are several different levels of network hierarchy (e.g. nodes of 8 GPUs with $B_{\text{network}} = 900\,\text{GB/s}$ per GPU, superpods of 32 nodes with $B_{\text{network}} = 450\,\text{GB/s}$ per GPU and InfiniBand across superpods achieving $B_{\text{network}} = 100\,\text{GB/s}$ per GPU), matters become more complicated:

1. We must decide how different degrees of parallelism are partitioned across different levels in the network hierarchy. For example, because tensor parallelism is more communication-intensive than data and pipeline parallelism, it usually makes sense to do tensor parallelism at lower levels of the network hierarchy (e.g. inside nodes using NVLink for internal communication) and data, pipeline, or expert parallelism at higher levels.
2. All-reduces and point-to-point communications will have to be implemented hierarchically to make efficient use of the available network bandwidth.

3. Pipeline parallelism and expert parallelism interact in a nontrivial way when multiple levels of network hierarchy are taken into account. Specifically, having to send requests to an expert outside of a lower level of the network hierarchy can cause information to be routed across slower connections even if pipeline parallelism is present at lower levels of the network hierarchy. This is taken into account in our model.

We address each of these points in order. First, if our network has H levels of hierarchy enumerated as $h \in \{1, 2, \ldots, H\}$, and each level h of the hierarchy has a per-GPU bandwidth of $B_{\text{network}}(h)$ where levels with smaller h have faster bandwidth, we assume there's some partition of the parallelism degrees across the H levels. Specifically, if X is a dimension of parallelism, we assume there is a factorization $N_X = N_X(1) \cdot N_X(2) \cdot \ldots \cdot N_X(H)$.

In such a setup, we can perform a hierarchical all-reduce of d words across dimension X as follows:

- Reduce d words across $N_X(1)$ participants $\prod_{h=2}^{H} N_X(h)$ times using a per-GPU bandwidth of $B_{\text{network}}(1)$.
- Reduce d words across $N_X(2)$ participants $\prod_{h=3}^{H} N_X(h)$ times using a per-GPU bandwidth of $B_{\text{network}}(2)$.
- $\ldots$
- Reduce d words across $N_X(H)$ participants once using a per-GPU bandwidth of $B_{\text{network}}(H)$.

This explains how to modify the calculation for data and tensor parallelism: each all-reduce operation is replaced by H all-reduce operations in ascending order across the network hierarchy, with bandwidth, though not the latency, of these all-reduces overlapped.

For pipeline and expert parallelism, we must think more carefully because of the interaction between these two modes of parallelism: a single point-to-point communication can simultaneously do the job for both. We consider each combination $(h_{\text{PP}}, h_{\text{EP}}) \in \{0, \ldots, H\}^2$ of network hierarchy levels possibly requiring communication from pipeline and expert parallelism (with level 0 corresponding to the case where no communication is required), determine the frequency of that combination, and then tally a communication on the highest level of the two $h' = \max\{h_{\text{PP}}, h_{\text{EP}}\}$ with that frequency.

Pipeline-parallel communication frequencies. Define $P_{\text{PP}}(h)$ to be the probability that an inter-layer interface (i.e., between two MLP blocks) requires pipeline-parallel communication at level h of the network hierarchy.

If $N_{\text{PP}} = 1$ (i.e. no pipeline parallelism is utilized), then clearly $P_{\text{PP}}(0) = 1$ and $P_{\text{PP}}(h) = 0$ for $h > 0$, since $h = 0$ corresponds to the case that no pipeline-parallel communication is required.

If $N_{\text{PP}} > 1$ (i.e. any pipeline parallelism at all is utilized), let $h^* = \max\{h : N_{\text{PP}}(h) > 1\}$ be the highest level of the network hierarchy at which pipeline parallelism ever takes place. For a given level h, there are

$$\left(i \prod_{k=h}^{h^*} N_{\text{PP}}(k) \right) - 1 \tag{19}$$

pipeline stage interfaces at level h or above, where i is the interleaving factor as in Section 3.2.1.

When $h = h^*$, this is just $i \cdot N_{\text{PP}}(h^*) - 1$, and hence the frequency that one of the $L-1$ inter-layer interfaces is an h^*-level pipeline stage interface is just

$$P_{\text{PP}}(h^*) = \frac{i \cdot N_{\text{PP}}(h^*) - 1}{L - 1}.$$

Otherwise when $1 \leq h < h^*$, we employ Eq. 19 twice to see that there are

$$\left(i \prod_{k=h}^{h^*} N_{\text{PP}}(k) \right) - \left(i \prod_{k=h+1}^{h^*} N_{\text{PP}}(k) \right) = \left(i \prod_{k=h+1}^{h^*} N_{\text{PP}}(k) \right) \left(N_{\text{PP}}(h) - 1 \right)$$

pipeline stage interfaces at exactly level h, yielding the frequency

$$P_{\text{PP}}(h) = \left(i \prod_{k=h+1}^{h^*} N_{\text{PP}}(k) \right) \cdot \frac{N_{\text{PP}}(h) - 1}{L - 1} \qquad (1 \leq h < h^*)$$

that an inter-layer interface requires pipeline-parallel communication at this level of the network hierarchy.

Employing Eq. 19 one last time with $h = 1$ yields the total number $iN_{\text{PP}} - 1$ of pipeline stage interfaces across all levels of the network hierarchy. Subtracting from $L-1$ gives us the number of inter-layer interfaces that do *not* require pipeline communication, with frequency

$$P_{\text{PP}}(0) = \frac{L - iN_{\text{PP}}}{L - 1}.$$

Expert-parallel communication frequencies. Now similarly define $P_{\text{EP}}(h)$ to be the probability that one of the $L-1$ inter-layer interfaces requires expert-parallel communication at level h of the network hierarchy.

We assume expert routing is uniform, so the probability is

$$\left(\prod_{k=h}^{H} N_{\text{EP}}(k) \right)^{-1} \tag{20}$$

that the current and next expert lie at the same rank on all levels of the network hierarchy at or above h. Employing Eq. 20 twice, the probability is

$$P_{\text{EP}}(h) = \left(\prod_{k=h+1}^{H} N_{\text{EP}}(k) \right)^{-1} - \left(\prod_{k=h}^{H} N_{\text{EP}}(k) \right)^{-1} = \frac{N_{\text{EP}}(h) - 1}{\prod_{k=h}^{H} N_{\text{EP}}(k)} \qquad (h \geq 1)$$

that the current and next expert lie at the same rank on all levels of the network hierarchy above h but not at level h, and hence that expert-parallel communication must occur on exactly level h. As a sanity check, observe that the sum

$$\sum_{h=1}^{H} P_{\text{EP}}(h)$$

telescopes to $1 - 1/N_{\text{EP}}$, where the leftover probability of $1/N_{\text{EP}}$ corresponds to the chance that no expert-parallel communication will be needed, as the next expert is stored in the same rank as the current expert on all levels of the network hierarchy. Thus $P_{\text{EP}}(0) = 1/N_{\text{EP}}$.

Joint communication frequencies. Activations (and their gradients, on the backward pass) must be communicated at the inter-layer interface between MLP blocks on the level of the network hierarchy corresponding to the maximum of that required for pipeline parallelism and that required for expert parallelism. Since we assume expert routing is independent of layer and token, the probability that communication is required on network level h' is:

$$P_{\text{P2P}}(h') = \sum_{\substack{0 \le h_{\text{PP}}, h_{\text{EP}} \le H \\ \max\{h_{\text{PP}}, h_{\text{EP}}\} = h'}} P_{\text{PP}}(h_{\text{PP}}) P_{\text{EP}}(h_{\text{EP}}).$$

Latencies and overlap. Two final wrinkles: we will need to make assumptions about to what extent network communication is overlapped with computation, and also account for network latencies. To this end, we decompose the network communication time as

$$t_{\text{network}} = t_{\text{overlapped network}} + t_{\text{nonoverlapped network}}$$

and assume the first term can be overlapped with computations while the second term cannot be. In the ideal case we have $t_{\text{nonoverlapped network}} = 0$, but we also find it useful to consider cases where overlapping is not quite perfect.

On top of the above calculations regarding network bandwidth, we must also track network latency. We do this by considering the number of times that each all-reduce or point-to-point communication operation must occur serially (as opposed to in parallel) and multiplying this number by a network latency parameter at that particular level of the network hierarchy. $t_{\text{network latency}}$ is then defined as the sum of all of these latency timescales. Furthermore, for the latency (as opposed to bandwidth) of peer-to-peer communication, we always assume the worst-case for expert routing (i.e. $P_{\text{EP}}(h^*) = 1$ where h^* is the highest level of the network hierarchy employing expert parallelism, and $P_{\text{EP}}(h) = 0$ for $h \neq h^*$), as all tokens in a microbatch must wait on the slowest one.

8.1.3 Pipeline bubbles

We take pipeline bubbles into account by following Section 3.2.1 and assuming either a 1F1B interleaving schedule (Narayanan et al., 2021) or a ZB-H2 schedule (Qi et al., 2024). For the 1F1B schedule, the bubble fraction f_b is computed following Equation 22:

$$f_b = \frac{(N_{\text{PP}} - 1) + z}{(N_{\text{PP}} - 1) + z + im},$$
$$z = (i - 1) \cdot \max(0, N_{\text{PP}} - m),$$

where i is the pipeline interleaving factor and m is the number of microbatches in the pipeline. For the ZB-H2 schedule, we assume $f_b = 0$ but require that the number of microbatches $m \geq 2N_{\text{PP}} - 1$, as this is the condition required to achieve no pipeline bubble.

Data-parallel all-reduce communication time is assumed exempt from the pipeline bubble as this normally occurs in a separate phase.

8.1.4 Scaling assumptions

We assume the following scaling relationship for the baseline scenario:

- **Critical batch size:** $b = 2^{22} \cdot E^{1/2} \cdot \left(\frac{T}{3 \cdot 10^{23}\,\text{FLOP}}\right)^{1/6}$ tokens, where T is the training compute of a compute-optimal model
- **Number of MLP blocks:** $L = 0.10056 \cdot (d_{\text{model}} d_{\text{ff}})^{0.3751}$
- **Sparsity factor:** $E = 8 \cdot \left(\frac{d_{\text{model}} d_{\text{ff}}}{4 \cdot 12288^2}\right)^{1/2}$
- **Feedforward dimension:** $d_{\text{ff}} = 4 \cdot d_{\text{model}}$
- **Dataset size:** $D = 20 N_{\text{p}}$

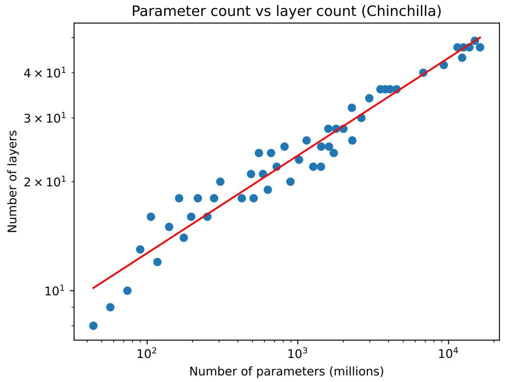


Figure 10: Parameter vs. layer count for models trained in Hoffmann et al. (2022). The red line corresponding to the best power law fit is given by $L = 3.67 \cdot (N_{\text{p}}/10^6)^{0.27}$ and achieves an R^2 of 94.7%.

The scaling relation for the number of MLP blocks is informed by the scaling done in Hoffmann et al. (2022). Figure 10 shows the results we obtain by analyzing the information about layer scaling in the models trained in Hoffmann et al. (2022), with the best-fitting scaling law $L \approx 3.67 \cdot (N_{\text{p}}/10^6)^{0.27}$ where L is the number of layers and N_{p} is the number of model parameters.[13]

These are not as well-justified as we would like them to be due to a lack of solid empirical evidence. We choose them to match cases where we have explicit information when we can (e.g. a sparsity factor of 8 for GPT-4, a batch size of 2^{22} tokens for GPT-3, *et cetera*) and we base the scaling exponents on the discussions in relevant parts of Section 4. Our framework allows for the input of alternative scaling assumptions.

8.1.5 Time taken for a training run

Putting together the considerations from Sections 8.1.1, 8.1.2 and 8.1.3, we compute the time taken per gradient step as

$$t_{\text{step}} = t_{\text{network latency}} + t_{\text{DP},n} + \max\left(t_{\text{DP},y}, \frac{\max(t_{\text{matmul}}, t_{\text{n-DP},y}) + t_{\text{n-DP},n}}{1 - f_b}\right) \tag{21}$$

Here, the subscripts DP and n-DP of t refer to data parallel and non-data parallel communication time respectively, while the subscripts y and n (standing for ”yes” and ”no”, respectively) refer to whether said communication can be overlapped with computation or other communication or not. In the Zero Bubble case only data-parallel communication counts toward $t_{\text{network latency}}$, as the rest can be hidden.

[13] If we directly substitute $N_{\text{p}} = 2Ld_{\text{model}}d_{\text{ff}}$, these two scaling laws appear inconsistent with each other. The discrepancy is explained by the fact that the models trained by Hoffmann et al. (2022) have attention layers, while our toy model implying this expression for the parameter count only considers MLP blocks and ignores the QKV and post-attention projections, which in practice also contribute to the parameter count.

We combine this with the scaling relationships from 8.1.4 to compute the time taken for the entire training run as

$$t_{\text{run}} = n_{\text{steps}} \cdot t_{\text{step}} = \frac{D \cdot t_{\text{step}}}{b},$$

where D is the training dataset size and b is the critical batch size. This is the final output of our model.

8.2 Appendix: Effect of pipeline interleaving

Suppose our model has $L = 12$ MLP blocks (numbered $0 \ldots 11$), and our pipeline has $N_{\text{PP}} = 3$ stages (numbered $0 \ldots 2$). In the non-interleaved ($i = 1$) case, then the MLP block $\rightarrow$ pipeline stage mapping is:

MLP Block	0	1	2	3	4	5	6	7	8	9	10	11
Pipeline Stage	0				1				2			

If instead we employ an interleaving factor i (restricted so that $N_{\text{PP}} \cdot i$ divides L), then on every forward and backward pass, each of the m microbatches passes through the pipeline i times rather than once, each time seeing only L/i consecutive MLP blocks. On each pass through the pipeline, the N_{PP} pipeline stages are encountered in order, each responsible for $L/(N_{\text{PP}} \cdot i)$ consecutive MLP blocks per pass.

So in the concrete example above, if we take $i = 2$, the mapping becomes:

MLP Block	0	1	2	3	4	5	6	7	8	9	10	11
Pipeline Stage	0		1		2		0		1		2	

As there are now $N_{\text{PP}} \cdot i$ "virtual" pipeline stages, the number of inter-stage interfaces has increased from $N_{\text{PP}} - 1$ (in the interleaving-free case) to $N_{\text{PP}} \cdot i - 1$. Therefore activations must be communicated approximately i times as frequently. In exchange, devices spend less time idle in the pipeline bubble, as the pipeline fills up and clears out i times as fast at the beginning and end of each batch.

Let us compute the bubble fraction precisely, which as usual is easiest from the perspective of the last stage. It spends $N_{\text{PP}} - 1$ "bubble" steps waiting for that same number of previous stages before it sees its first microbatch, then sees all m microbatches, for m steps of work.

At this point, the microbatches are recycled through the pipeline for the next interleaving cycle. If $m < N_{\text{PP}}$, then there are insufficient microbatches to fill the pipeline, and the last stage must wait $N_{\text{PP}} - m$ steps for the first microbatch to reach the end again, at which point it has m more steps of work. This recycling process repeats $i - 1$ times.

Tabulating, we have $N_{\text{PP}} - 1$ initial bubble steps, $z := (i - 1) \cdot \max(0, N_{\text{PP}} - m)$ inter-cycle bubble steps, and im work steps, so our last stage's forward pass bubble fraction is

$$B_{\text{interleaved PP}} = \frac{N_{\text{PP}} - 1 + z}{N_{\text{PP}} - 1 + z + im}. \tag{22}$$

The situation looks the same in reverse on the backward pass. Furthermore, since all stages have the same amount of work to do, they must also have this same bubble fraction.

In the typical case when the number of microbatches m can fill the pipeline depth N_{PP}, then this is simply Equation 5, with im substituted for m. So we have multiplied the "effective" microbatch count by i, reducing the bubble accordingly, at the cost of increasing pipeline communication costs by roughly this factor as well.

8.3 Appendix: Allowing for layer count and batch size scaling

In Sections 5.1 and 5.2 we assumed a ratio between the critical batch size b and the number of MLP blocks L that does not scale with the size of the training run, which we justified with some (weak) empirical evidence. However, our analysis can be generalized. Suppose instead the power law scaling relationships $b = b_0(T/T_0)^{\alpha_b}$, $L = L_0(T/T_0)^{\alpha_L}$, where T is the total training compute, and b_0, L_0 are the batch size and depth for a reference training run of T_0 compute. Then defining $\alpha := \alpha_b - \alpha_L$ and solving for T_{critical}, the bandwidth bottleneck formula Eq. 13 becomes

$$T_{\text{critical}} = \left[\frac{1}{(960\,\text{MAC}) \cdot E}\left(\frac{b_0}{L_0} \cdot \frac{1}{{T_0}^{\alpha}} \cdot \frac{Ct_{\text{train}}}{d'^2 b'}\right)^2\right]^{\frac{1}{1-2\alpha}},$$

while solving for T_{limit}, the latency limit formula Eq. 18 becomes

$$T_{\text{limit}} = \left[\frac{3\,\text{MAC}}{320 \cdot E}\left(\frac{b_0}{L_0} \cdot \frac{1}{{T_0}^{\alpha}} \cdot \frac{t_{\text{train}}}{t_L}\right)^2\right]^{\frac{1}{1-2\alpha}}.$$

We plot the effects for dense training runs in Figure 11, taking as our reference training run $T_0 = 3 \times 10^{23}$ FLOP, with $b_0 = 4 \times 10^6$ tokens per batch and $L_0 = 100$ layers, assuming a 9 μs lower bound on the matrix multiplication timescale, accounting for both intra- and inter-GPU latencies.

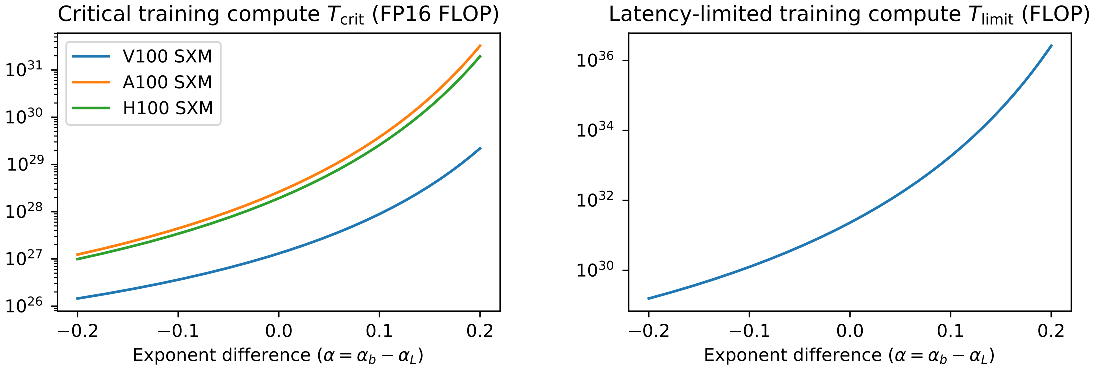


Figure 11: The sensitivity of T_{critical} and T_{limit} for dense training runs, to the exponent difference $\alpha_b - \alpha_L$, starting from a reference training run of 3×10^{23} FLOP for a model having 100 MLP blocks, using a batch size of 4M tokens. In Sections 5.1 and 5.2 we assume $\alpha_b = \alpha_L$, in which case the effects of batch and layer scaling cancel out.

If correct, the results in Bi et al. (2024) imply an aggressive $\alpha_b = 0.3271$, yielding $\alpha = \alpha_b - \alpha_L \approx 0.2$. The impact is about three orders of magnitude of training compute, highlighting the importance of better understanding the scaling of the optimal batch size and model depth. At this scale, the relevant constraint becomes economic: acquiring a sufficiently large cluster and the energy to operate it.