How Many GPUs Do You Need To Train a Transformer?

Li Ju

li.ju@it.uu.se

If you set out to train a 7 Billion parameter Transformer (like LLaMA or Mistral) from scratch in 2026, the first question you face is hardware: "How many GPUs do I need?" This note dives deep into the maths on how to evaluate the number of GPUs you need, and how can you reduce it.

1. Problem Setting

Assume we are training a classic transformer of 7B parameters, with a batch size of 128 using Adam(W). Following are the specifications of the transformer we are talking about:

Parameter	Value	Description
Layers (n_{layers})	32	The total number of transformer blocks.
Hidden Dimension (d_{model})	4,096	The size of the latent space (embedding vector size).
Attention Heads (n_{heads})	32	Number of query heads.
Head Dimension (d_{head})	128	Size of each individual head ($4096/32$).
Intermediate Size (d_{interm})	16,384	The hidden dimension of the MLP layers.
Vocabulary Size (V)	32,000	Total number of tokens in the tokenizer.
Context Length (s)	4,096	The maximum number of tokens in a single sequence.
Activation Function	GeLU	Instead of traditional ReLU.
Normalization Layer	LayerNorm	Layer-wise normalization plus linear projection

Let's verify that this model is indeed of 7B parameters:

1. Embedding layer: $V \times d_{\text{model}} = 32000 \times 4096 \approx 131M$.
2. Self-attention (per layer): Q, K, V and the output projection (from the layer norm) matrices $4 \times d_{\text{model}} \times d_{\text{model}} \approx 67M$
3. MLP (per layer): Dimention up and down back matrices $2 \times d_{\text{model}} \times d_{\text{interm}} = 2 \times 4096 \times 16384 \approx 134M$.
4. Final output layer: $d_{\text{model}} \times 4096 \times 32000 \approx 131M$.

Summing all up we have total parameters as

$$N_{\text{total}} = 131M + (67M + 134M) \times 32 + 131M \approx 6694M \approx 7B.$$

2. VRAM Estimation

To train a transformer, we need the GPU to store the following parts in its VRAM:

- Model weights
- Gradients
- Optimizer States
- Activations

Let's break it down and see how much does each part cost.

2.1 Model weights

In 2026, the model weights are generally stored in `FP16`, which takes 2 bytes each. Then we have

$$M_{\text{weights}} = 7B \times 2\text{b} = 14\text{GB}$$

2.2 Gradients

Gradients are used to update model weights and have exactly the same shape as model weights. We then have

$$M_{\text{gradients}} = 7B \times 2\text{b} = 14\text{GB}$$

2.3 Optimizer States

Since we are using Adam(W), we need to maintain a moving average of the first and second-order of the gradient momentum, which have exactly the same number of parameters as the gradient, 7B. However, for stability, both momentum are stored in `FP32`, and a copy of weights of high precision `FP32` is also stored in Adam(W). Then we have

$$M_{\text{opt}} = 3 \times 7B \times 4\text{b} = 84\text{GB}.$$

If you are not using Adam(W) but using momentum SGD, you can save 1/3 of the memory here, with the size going down to 56GB.

2.4 Activations

Activations are the intermediate results temporarily stored during the forward pass, so that they can be used to compute the gradients during the back propagation step. As intermediate results, they do not directly rely on the number of model weights. They scale linearly with batch size B and quadratically with sequence length s. It can be estimated as follows:

Within one transformer block, first we have the input $B \times s \times d_{\text{model}}$. Then the input will be passed into the self attention layer and a layer norm layer, then will be passed to the MLP and another layer norm layer. Let's break it down further.

2.4.1 Self attention layer

Self attention is done by

$$X_{\text{MHA}} = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_{\text{head}}}}\right)V^\top$$

where $Q = W_q X, K = W_k X$ and $V = W_v X$.

1. Here first of course we need to store the input X of size $B \times s \times d_{\text{model}}$.
2. Since we are using multi-head attention, we essentially have n_{head} Q, K and V matrices, each of which is of size $B \times s \times d_{\text{head}}$. They have a total size of $3 \times B \times s \times d_{\text{model}}$ anyway since $d_{\text{model}} = n_{\text{head}} \times d_{\text{head}}$,
3. Then for each head, we need to store the attention score matrix, of size $B \times s \times s$. Thus in total we have $n_{\text{head}} \times B \times s \times s$.
4. We don't need to store intermediate results for dividing the score matrix by $\sqrt{d_{\text{head}}}$ since it is a constant. But we do need to store the attention probability matrix, of size $n_{\text{head}} \times B \times s \times s$.
5. Also we need to store X_{MHA}, of size $n_{\text{head}} \times B \times s \times d_{\text{head}} = B \times s \times d_{\text{model}}$.
6. Finally in the layer norm layer, we normalize and rescale the output, so we need to store the result matrix of each step, with a total size $2 \times B \times s \times d_{\text{model}}$.

2.4.2 MLP layer

For the MLP layer, we have two linear layers followed by a layer norm:

1. The first layer maps the result up to d_{interm}, resulting the activation of size $B \times s \times d_{\text{interm}}$.
2. The activation function is applied element-wise, so we have activation of size $B \times s \times d_{\text{interm}}$ again.
3. The second layer maps the result down to d_{model}, resulting the activation of size $B \times s \times d_{\text{model}}$.
4. The last layer norm has a total results of size $2 \times B \times s \times d_{\text{model}}$.

Summing everything up, we have the total activations of each transformer block as

$$N_{\text{activation}} = \underbrace{\mathbf{9} \times B \times s \times d_{\text{model}} + 2 \times B \times s \times d_{\text{interm}}}_{\text{linear}} + \underbrace{2 \times n_{\text{head}} \times B \times s^2}_{\text{self attention}}.$$

Note here 9 (instead of $1 + 3 + 1 + 2 + 1 + 2 = 10$) is emphasized since the output of this transformer block is the input of the next one.

Injecting all numbers we have $B = 128, s = 4096, d_{\text{model}} = 4096, n_{\text{head}} = 32, d_{\text{interm}} = 16384$, we have $N_{\text{action}} = 173B$. If the activations are stored in `FP16` (2 bytes), roughly it takes

$$M_{\text{activation}} \approx \underbrace{68\text{GB}}_{\text{linear}} + \underbrace{275\text{GB}}_{\text{attention}} = 343\text{GB}.$$

Note: This is just activations for ONE transformer block. We have 32 in total, thus $343 \times 32\text{GB}$ in total.

2.5 Summing Up

To summarize all the parts, we have the following numbers:

- $M_{\text{weights}} = 14\text{GB}$.
- $M_{\text{gradient}} = 14\text{GB}$
- $M_{\text{opt}} = 84\text{GB}$.
- $M_{\text{activation}} = 343 \times 32\text{GB} \approx 11\text{TB}$

and we have

$$M = M_{\text{weights}} + M_{\text{gradient}} + M_{\text{opt}} + M_{\text{activation}} \approx 11\text{TB}$$

3. Practicality via Data Parallelism

All the calculations above is based on the assumption that: All the values should be stored in one single GPU with a huge VRAM (at least 11TB :-)). To practically to train the model, we need to split the training process into multiple GPUs. Let's take A100 (40GB) as the example. How many A100s do we need? Is it $\text{ceil}(11,000/40) = 275$? It is much more complicated than that.

3.1 Vanilla Data Parallelism

We see that out of the 11TB VRAM, most are taken by the activations, which scales linearly with the batch size. This part can be safely broken down into multiple GPUs safely, by saving activations of one mini batch within each GPU's VRAM. However, in order to get these activations via forward pass, backpropagate through the activations and update the model parameters, we need to keep a copy of model weights (14GB), gradients (14GB) and optimizer states (84GB) in **each** GPU.

This is **not possible**: Each A100 GPU has only 40GB VRAM, which does not even fit in $14 + 14 + 84$ GB, not even to mention any activations.

3.2 ZeRO: Zero Redundancy Optimizer

In the native data parallelization, each GPU needs to keep a replica of the model weights, gradients and optimizer states, making data parallelism a mission impossible. ZeRO sharding these values in to different GPUs makes data parallelization practical. Here are three stages of ZeRO

3.2.1 Stage 1: Sharding the optimizer states

The optimizer state is the biggest memory eater (84GB). It is not necessary for each GPU to keep a full copy of all of them. By slicing it into N_{gpu} shards, each GPU is only responsible for updating its own part.

This is done by following steps:

1. Each GPU do a forward pass using their own batch of data.
2. Each GPU computes the gradients of all parameters (which is based on their own batch of data).
3. Synchronize the gradients of all GPUs using global average.
4. Each GPU update the momentums and high precision weights of their own responsible subset of parameters.
5. Synchronize the parameters.

Now that it is possible, how many GPUs do we need? Simply calculation gives that

$$40\text{GB} = \frac{M_{\text{opt}} + M_{\text{activation}}}{N_{\text{gpu}}} + M_{\text{weights}} + M_{\text{gradient}}$$

we have $N = 924$. So we need 924 A100 to train a 7B transformer.

3.2.2 Stage2: Sharing the gradients

Further we can shard the gradients of the model. Checking Step 2 in ZeRO1, we compute all the gradients, and then synchronize them. But, gradient is computed layer by layer with the chain rule. We can synchronize the gradient of each layer once it is computed, instead of waiting the full gradient is computed and saved. In another world, ZeRO1 do the full back propagation getting all the gradients, and synchronize all the gradients after backpropagation. But ZeOR2 does back propagation and synchronization in a layer-wise interleaving manner:

1. Same as ZeRO Step 1
2. Calculate gradient for layer L. Synchronize the gradient for layer L.
3. Dropt Gradient for layer L if it is not your responsibility. $L := L = 1$. Go to Step 2.
4. Same as Zero Step 4, 5.

Then how many GPUs do we need? We now have

$$40\text{GB} = \frac{M_{\text{opt}} + M_{\text{activation}} + M_{\text{gradient}}}{N_{\text{gpu}}} + M_{\text{weights}},$$

and we have $N \approx 423$ GPUs.

3.2.3 Stage 3: Sharding the weights

The final step is to share the model weights across GPUs. Each layer will be sharded into N_{gpu} slices and stored into N_{gpu} GPUs. Now let's focus on Step 1 and 2:

Step 1:

1. For Layer L, for each GPU, collecting the missing $N_{\text{gpu}} - 1$ parts from other GPUs using `allgather` to reconstruct layer L on all GPUs.
2. Feed forward mini batch data, store the activations for all the parameters, and delete the other $N_{\text{gpu}} - 1$ shards of weights. $L := L + 1$ and go to Step 1.1.

Then to do back propagation for the calculation of gradients of layer L, we need to recover the full layer L again:

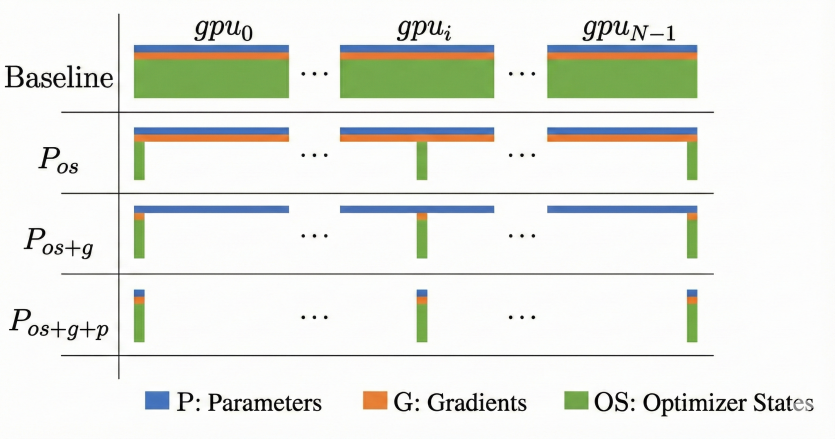
Step 2:

1. Collect the missing $N_{\text{gpu}} - 1$ shards and recover layer L.
2. Calculate the gradient, drop both gradients and parameters that the GPU is not responsible for.
3. $L := L + 1$ go to Step 2.1.

Now we have

$$40\text{GB} = \frac{M_{\text{opt}} + M_{\text{activation}} + M_{\text{gradient}} + M_{\text{weights}}}{N_{\text{gpu}}},$$

and we have $N \approx 275$.



Note: You may also see people using Fully Sharded Data Parallel (FSDP) too. Don't panic. FSDP is exactly stage 3 ZeRO of PyTorch's native implementation. One may choose from DeepSpeed (ZeRO) or FSDP. They are just different implementations.

4. Compressing Activations

Checking the components of VRAM demands, the activations take the majority of the VRAM (~99%). Although we made it possible to fit the training into GPUs, we can use two different ways to compress the VRAM demands from the activations, namely Gradient/Activation Checkpointing and FlashAttention.

4.1 Gradient/Activation Checkpointing

ZeRO trades practicality of large VRAM demand with inter-GPU communication, by gathering and reconstructing weights, gradients and optimizer states multiple times during one optimization step. Alternatively, we can trade practicality with computation using gradient/activation checkpointing too.

4.1.1 What is activation checkpointing

For now we save *every* intermediate results from forward pass to the end of the back propagation. However, the activations for each layer are only used when back propagating through that layer, which can be recomputed easily. Here is a concrete yet simple example of gradient checkpointing:

Standard training (No Checkpointing)

In this mode, all intermediate activations are stored in memory during the forward pass to be reused during the backward pass.

- **Forward Pass:**
  $x \xrightarrow{f_1,w_1} a_1 \xrightarrow{f_2,w_2} a_2 \xrightarrow{f_3,w_3} \cdots \xrightarrow{f_n,w_n} \mathcal{L}$
  Memory State: Stored $\{x, a_1, a_2, \ldots, a_{n-1}\}$
- **Backward Pass:**

To compute the gradient for w_i, the engine retrieves the stored activation a_{i-1}:
$\nabla_{w_i}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial a_i} \cdot \frac{\partial a_i}{\partial w_i}$
Result: High memory usage ($O(n)$), but faster computation.

Gradient Checkpointing training

This mode trades compute for memory by discarding intermediate activations and recomputing them only when needed.

- **Forward Pass:**
  $x \xrightarrow{f_1,w_1} a_1 \xrightarrow{f_2,w_2} a_2 \cdots \to \mathcal{L}$
  Memory State: Stored $\{x\}$ (Checkpoint). Activations $\{a_1, a_2, \ldots\}$ are **discarded** after their use in the forward step.
- **Backward Pass (Recomputation Phase):**
  When the gradient $\frac{\partial \mathcal{L}}{\partial a_2}$ is received, a_1 and a_2 are missing. The engine performs a local "sub-forward" pass:
  1. **Recompute:** $a_1 = f_1(x, w_1)$ and $a_2 = f_2(a_1, w_2)$
  2. **Differentiate:** $\nabla_{w_i}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial a_i} \cdot \frac{\partial a_i}{\partial w_i}$
  3. **Clean up:** a_2 is discarded again to free memory while the engine moves to w_1.

  Result: Low memory usage but higher computation time.

4.1.2 VRAM ReCalculation

Persistent VRAM Demand

Then back to our transformer block. For each block, we have

$$N_{\text{activation}} = \underbrace{\mathbf{9} \times B \times s \times d_{\text{model}} + 2 \times B \times s \times d_{\text{interm}}}_{\text{linear}} + \underbrace{2 \times n_{\text{head}} \times B \times s^2}_{\text{self attention}}$$

number of activations. However, if we discard *all other activations* but the input for the block, i.e., $N^{\text{GC}}_{\text{activation}} = B \times s \times d_{\text{model}}$, we can greatly reduce the VRAM demand at a cost of computation: We then need to go through the transformer block once during the forward pass stage to get the output, and once again during the backprogapation to get the activations. Then we have

$$N^{\text{GC}}_{\text{activation}} = B \times s \times d_{\text{model}}.$$

Storing in `FP16`, the needed VRAM size can be reduced to 4.3GB per block.

Then we can re-compute the VRAM we need by enabling Gradient Checkpointing:

$$\begin{aligned} M^{\text{GC}}_{\text{pers}} &= M_{\text{opt}} + M_{\text{gradient}} + M_{\text{weights}} + M^{GC}_{\text{activation}} \\ &= 14\text{GB} + 14\text{GB} + 84\text{GB} + 32 \times 4.3\text{GB} \approx 250\text{GB} \end{aligned}$$

Temporary VRAM Demand

But this is **NOT** the full picture, since we have only considered the VRAM that are persistently needed. There are still temporary values that need to be stored in VRAM too.

Gradient checkpointing frees us from storing activations of *all layers* at the same time, but still we need to store the activations of the layer we are working on temporarily. Think about it: When doing forward pass or back propagation enabling gradient checkpointing, layer by layer we compute the attention scores and the attention probabilities, use them and discard them. In our previous calculations, we simply remove them from the VRAM term but in reality, once they are computed, they do need to be stored in VRAM, though temporarily. This is still a massive number:

$$M_{\text{temp}} = \frac{M_{\text{activation}} - M_{\text{activation}}^{\text{GC}}}{n_{\text{layers}}} = 343\text{GB} - 4.3\text{GB} \approx 340\text{GB}.$$

So in total, we have

$$M^{\text{GC}} = M_{\text{pers.}}^{\text{GC}} + M_{\text{temp}}^{\text{GC}} = 590\text{GB}$$

which is *MUCH* smaller than 11 TB without gradient checkpointing. If combined with ZeRO-3, the training can be done only in 15 GPUs.

4.2 FlashAttention

Even till now, the activations are still taking the most of our VRAM storage, of which the activations for the attention scores and probabilities are the main contributors. The computation of the attention score and probability matrices, namely $QK^\top$ and $\text{softmax}(QK^\top)$ can be further optimized using FlashAttention.

Before checkout FlashAttention, we need first understand the process and problems of the classic implementation of the attention on GPU. Like CPU's hierachical memory system, GPU also has two VRAM of two tiers:

- High Bandwidth Memory (HBM): The main 40GB VRAM in our context, which is huge but slow.
- Static RAM (SRAM): The "cache" on the computing chip itself. It is tiny (19MB on A100 in our context) but fast.

Rethinking how the attention score and probability matrices are computed (the IO heavy way):

1. Read all Q and K from HBM/
2. Compute $S = QK^\top$ (Size $s \times s$)
3. Write S to HBM.
4. Read S from HBM to do $\text{softmax}(\cdot)$.
5. Write P to HBM (size $s \times s$).
6. Load P and V from HBM.
7. Compute $O = PV$ (size $s \times d_{\text{head}}$).
8. Write O to HBM.

This is clearly IO-bounded: For the computation of each value, the GPU needs to read and write huge matrices frequently between the HBM to SRAM: While reading matrices from HBM is inevitable for the computation, we do not need to write them to HBM multiple times, since they will be discarded any way.

Instead of computing the entire $s \times s$ matrices at once, FlashAttention breaks the Q, K, V matrices into small blocks that fit into the fast SRAM (e.g., 128×128).

1. Load a block of query matrix Q_i to SRAM.
2. Loop through all blocks of key and value matrices:
    1. Load K_j, V_j into SRAM
    2. Compute attention score for Q_i against K_j.
    3. Update a running sum and running max in SRAM.
    4. Compute partial probability matrix $P_{i,j}$ using the current score.
    5. Compute $P_{i,j}V_j^\top$.
    6. Accumulate this result into the running output O_i,
    7. Discard K_j, V_j from SRAM (not writing back to HBM).
3. Write O_i to HBM.

The computation in step 2.3 - 2.4 is essentially online softmax, while the step 2.5 - 2.6 is block matrix multiplication.

FlashAttention is efficient in terms of both VRAM requirement and memory IO:

- Now all intermediate results are dropped directly from SRAM without the need of writing back to HBM, we do not need temporary memory reservation for the attention activations.
- Though both the classic attention implementation and FlashAttention has a computation cost of $\mathcal{O}(s^2)$, the later has a lower memory IO cost of $\mathcal{O}(s)$ while the former has it of $\mathcal{O}(s^2)$.

Now with FlashAttention, without the need of storing the attention score and probability matrices, we have

$$\begin{aligned} M_{\text{tmp}}^{\text{GC, FA}} &= \frac{M_{\text{activation}} - M_{\text{activation}}^{\text{GC}} - M_{\text{activation}}(\text{attention})}{n_{\text{layers}}} \\ &= 343\text{GB} - 4.3\text{GB} - 275\text{GB} \approx 64\text{GB} \end{aligned}$$

and

$$M^{\text{GC, FA}} = M_{\text{pers}}^{\text{GC}} + M_{\text{tmp}}^{\text{GC, FA}} = 250\text{GB} + 64\text{GB} = 314\text{GB},$$

which can be trained using 8 GPUs.

5. Further Optimizations

While ZeRO, Gradient Checkpointing, and FlashAttention, we reduce the number of A100 for the training of a 7B transformer from 924 to 8, which is a much affordable cost. If you are interested in squeezing the last drop, several "less significant" techniques can be further explored:

- **8-bit Optimizers:** The current calculation assumes the Optimizer States consume 84 GB (12 bytes per parameter). Using 8-bit Adam (via libraries like `bitsandbytes`) reduces this to roughly 4 bytes per parameter, saving approximately 56 GB of persistent VRAM.
- **Activation Quantization:** While we calculated the MLP activations at FP16 (2 bytes), modern techniques can store these stashed activations in 8-bit (`INT8`) or even 4-bit formats. This would halve the 64 GB temporary memory spike derived in the FlashAttention section.