CSE-234: Data Systems for Machine Learning, Winter 2025

2: Basics: Modern DL, computational graph, autodiff, frameworks

Lecturer: Hao Zhang   Scribe: Leduo Chen, Chao-Jung Lai, Ohm Rishabh, Raunak Sengupta, Vibha, Harshit, You Zhou, Kanaad, Jerry

1 Alpa Compiler: Hierarchical Optimization

To find the optimal strategy, the Alpa compiler employs a two-pass approach. Determining an optimal strategy in a single pass is too difficult, if not impossible. However, certain heuristics can be leveraged. Inter-op parallelism prefers low-bandwidth communication, while intra-op parallelism prefers high-bandwidth communication. The first pass is performed across nodes, and the second pass is executed within nodes using NVLink, leveraging hardware properties for better efficiency.

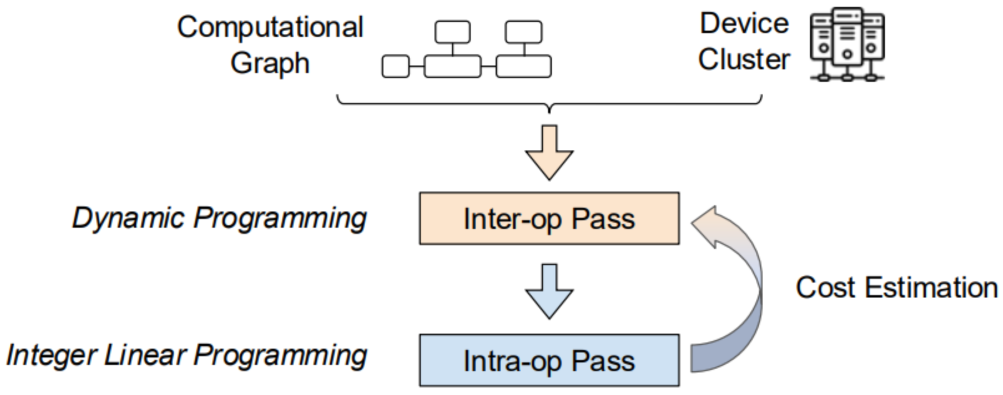


Figure 1: Inter and Intra Op Parallelism

1.1 Inter-Op Pass

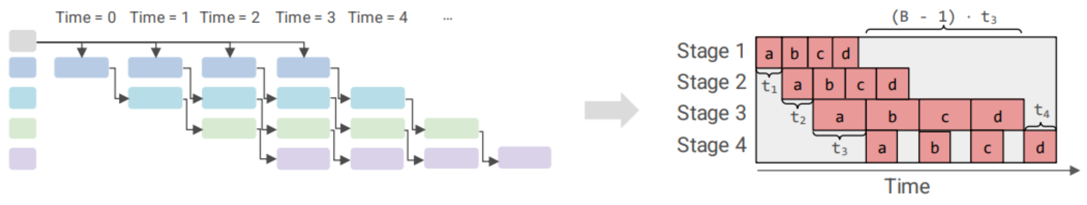
In the inter-op pass, the device cluster is split into sub-meshes, and the neural network is divided into different stages, mapping each stage to a corresponding mesh. After the mapping is complete, the inter-op pass optimizes the execution strategy given the assigned stage and mesh. The process continues by iterating over all possible mappings to find the best strategy.

A key question in this optimization is: how exactly do we solve this problem? In the inter-op pass, we aim to optimize pipeline parallelism by splitting the computation graph into different stages while minimizing pipeline execution latency. A realistic assumption is made that the pipeline schedule is already known.

When splitting a neural network into stages, each stage should take approximately the same time to execute. If any stage becomes a straggler, the entire pipeline will be slowed down. The network is divided into four stages (A, B, C, D), and each stage is assigned to a mesh so that they can execute simultaneously. The execution timeline involves T1, T2, and T3, where each represents a different phase of execution.

The entire pipeline latency is composed of two main terms:

- **Warmup Phase** – The initial phase where the pipeline is being filled.
- **Stable Phase** – The pipeline reaches a ready state, with an infinite stream of batches being processed. The overall latency depends on the longest stage in the pipeline.



$$T = \overset{\text{warmup phase}}{\sum_{i}^{S} t_i} + \overset{\text{stable phase}}{(B-1) \cdot \max_{1 \le j \le S}\{t_j\}}$$

Figure 2: Algorithm that is optimized with DP

The optimization problem can be solved using dynamic programming, where the goal is to minimize latency by balancing stage execution times. The process involves enumerating all possible stage splits and pruning the search space to find the optimal configuration.

Another assumption is that we already know the optimal latency of executing a given stage on its assigned mesh. This is necessary because each mesh consists of multiple devices, requiring further intra-op parallelization within that mesh. The overall strategy follows a hierarchical optimization approach:

- **The outer loop** assumes the optimal latency of running each stage.
- **The inner loop** optimizes the execution within the assigned mesh.

1.2 Intra-Op Pass

Once a stage and its corresponding mesh have been determined, the next step is to identify the best execution strategy within that mesh. The latency in intra-op optimization consists of two major components:

- **Node Cost** – The cost associated with choosing a particular partitioning strategy.
- **Edge Cost** – The additional resharding or communication cost incurred when two operators use different partitioning strategies and need to communicate.

1.3 Example: Matrix Multiplication Partitioning Strategies

Consider a simple dataflow graph with a single matrix multiplication (matmul) operation. There are multiple ways to partition the computation:

- **Algorithm 1: Partitioning along loop** i
  - The first matrix X is row-partitioned.
  - The weight matrix W_1 is replicated across devices.
  - No communication costs are incurred.
- **Algorithm 2: Partitioning along loop** j
  - Matrix X is replicated across devices.
  - The weight matrix W_1 is column-partitioned.
- **Algorithm 3: Partitioning along loop** k
  - Matrix X is row-partitioned.
  - The weight matrix W_1 is column-partitioned.
- **Algorithm 4: A more complex partially tiled strategy** This approach introduces additional complexity but may yield better computational efficiency in some cases.

Each algorithm is associated with a specific cost, which needs to be evaluated during execution.

1.4 Extended Example: Two Connected Matrix Multiplications

Now, consider a scenario where two matrix multiplications are connected and need to be parallelized. The goal is to evaluate different partitioning strategies and determine their communication overhead.

- **Possibility 1: Both operations use Algorithm 1**
  - Since both matrix multiplications use the same partitioning strategy, no layout conversion is required.
  - No resharding costs are incurred, so the cost is zero.
- **Possibility 2: First operation uses Algorithm 3, second uses Algorithm 2**
  - This configuration requires layout conversion from a partial sum to a replicated format.
  - The communication cost comes from an AllReduce operation.
- **Possibility 3: First operation uses Algorithm 1, second uses Algorithm 2**
  - This configuration requires an AllGather operation to distribute data across devices.
  - The AllGather cost adds overhead to execution.

To visualize these transitions, a graph representation can be used to show conversion paths between three states:

- Row-Partitioned
- Column-Partitioned
- Replicated

These states define how data is distributed across different devices and how communication overhead is introduced based on layout changes.

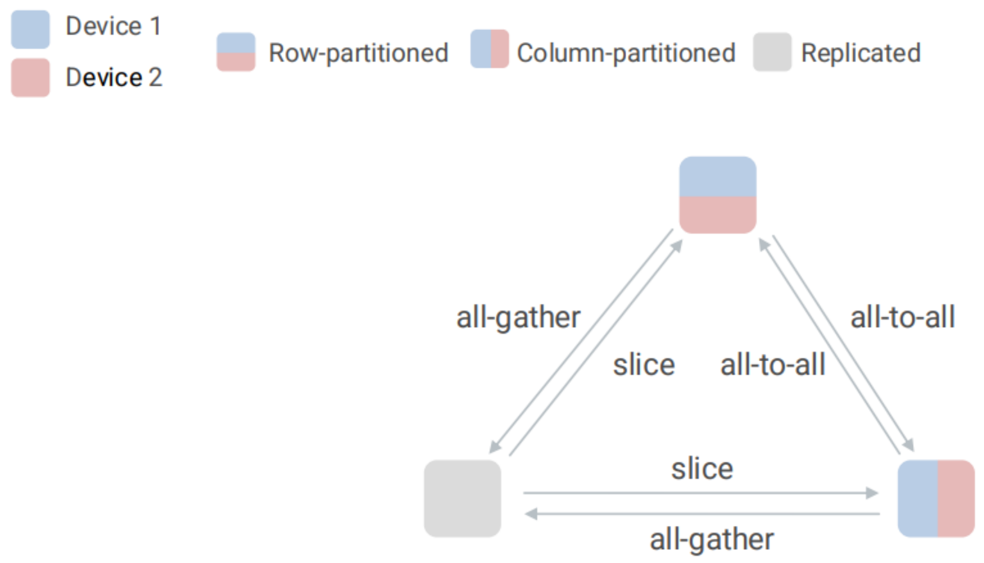


Figure 3: Conversion Paths

2 Hao's Ultimate Guide

Start by asking if your model can fit into a single GPU. If yes, then ask if the Job Completion Time (JCT) is acceptable. If JCT is acceptable, then you are good to go. If not, then data parallelism should be used until JCT becomes acceptable, since the model already fits into one GPU.

If the model does not fit into a single GPU, the next step is to check if memory optimizations can make it fit. Some common memory optimization techniques include checkpoint rematerialization, gradient accumulation, and micro-batching. However, swapping is avoided since it significantly slows down JCT.

If memory optimizations allow the model to fit into a single GPU, then check if JCT is still acceptable. If yes, then you are good to go, and there is no need for distributed systems. If JCT is still too high, or if none of the memory optimizations can make the model fit, then consider intra-op and inter-op parallelism without any memory optimizations.

Once you have determined that parallelism is necessary, the next step is to assess GPU connectivity and bandwidth.

3 Big Picture: Connecting Parallelization to LLM Training

At this stage, we understand all the foundational concepts, including dataflow graph optimization, automatic differentiation (autodiff), graph optimization, parallelization, runtime scheduling, memory management, operator optimization, and compilation. Now, we must connect these concepts to Large Language Models (LLMs).

The next lecture will focus on Transformers and attention mechanisms, exploring why large models are necessary and how scaling laws dictate model growth.

3.1 Connecting the Dots

Note: The professor will be reading the DeepSeek paper in class and covering certain hot topics if time permits.

3.1.1 Large Language Models: Next Token Predictioners

Large Language Models are basically next token predictioners; meaning, we have a prefix and we try to model the probability distribution, i.e, with a conditional prefix, we predict the next word. Now, once we have the probabilities whcih are possible, we are going to sample from this probability distribution of words.

$$P(next\ word|prefix)$$

For example, we have the sentences:

San Diego has very nice _	surfing	0.4
	weather	0.5
	snow	0.01
San Francisco is a city of _	innovation	0.6
	homeless	0.3

We can expand these sentences into factorization of conditional probabilities. With the condition "San Diego" (we consider San Diego as one word), we try to find the probability of the next word being "has". Then, we find the probability of the sequence by multiplying all the factors together.

Probability("San Diego has very nice weather")
$= P(\text{“}\textbf{San Diego}\text{”})P(\text{“}\textbf{has}\text{”}|\text{“}\textbf{San Diego}\text{”})P(\text{“}\textbf{very}\text{”}|\text{“}\textbf{San Diegohas}\text{”})P(\text{“}\textbf{city}\text{”}|\ldots)\ldots P(\text{“}\textbf{weather}\text{”}|\ldots)$

In reality, what we do is we *observe* a lot of these sentences, which we collect from various sources like the internet and books. We then use Maximum Likelihood Estimation (MLE) to estimate the parameters. The way the model is designed is it is capable of predicting the next token. We get probabilities and we try to maximize the likelihood of the observed data.

$$\underbrace{\text{Max}Prob(x_{1:T})}_{\text{MLE on observed data } x_{1:T},} = \underbrace{\prod_{t=1}^{T} P(x_{t+1}|x_{1\ldots t})}_{\text{This is next token prediction. Predicting using seq2seq NNs.}}$$

These conditional probabilities are modeled using Large Language Models, specifically transformers. We have a sequence of predictions with inputs x_1, x_2, x_3, x_4 and we try to predict the output.

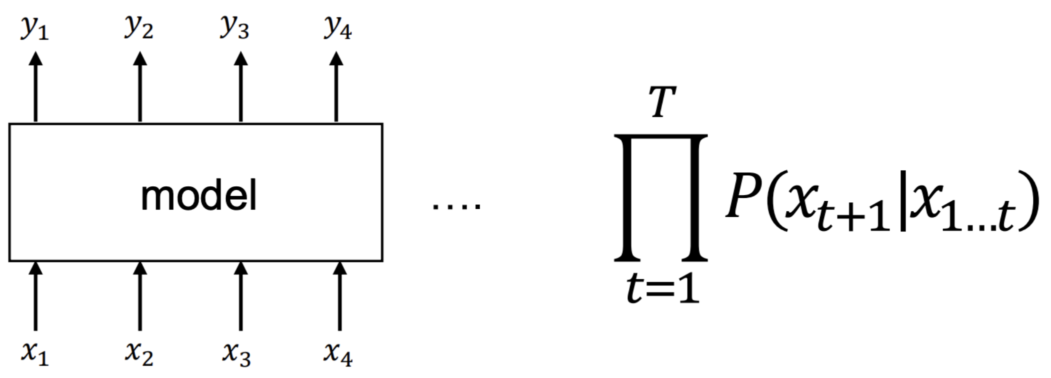
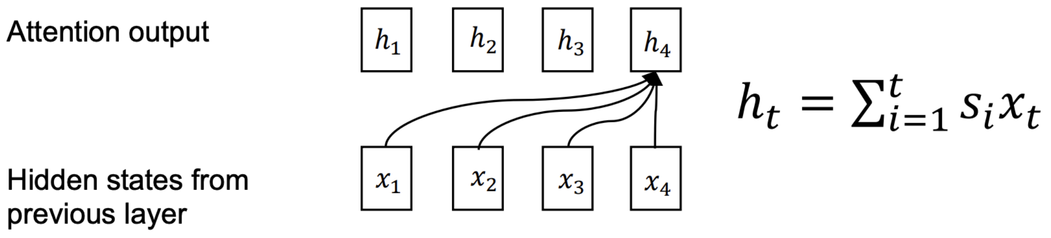


Figure 4: Sequence prediction

When we want to predict the next token, we usually take into account "causal relationships", where the preceding words "cause" the prediction of the following word.

There are many techniques for this; one method is the Recurrent Neural Network which does not work because (refer the first lecture) such a network cannot learn a very long sequence, it keeps "forgetting", and is hard to scale. Thus, we turn to *attention.*

3.1.2 Attention



Attention output
Hidden states from previous layer
h_1 h_2 h_3 h_4
x_1 x_2 x_3 x_4
$h_t = \sum_{i=1}^{t} s_i x_t$

Figure 5: The Input Layer and a Single Attention Layer

Attention generally refers to weighted combination of individual states. We observe input words, for example, embeddings. We usually stack multiple attention layers on top of this input, which are the hidden representations. Each state at the hidden token position *attends* to all previous layer states.

$$h_t = \sum_{i=1}^{t} s_i x_t$$

The hidden states are calculated (as shown in the above formula) as the weighted sum of the product of the observation from the previous layer and the attention score s_i. This attention score, s_i, computes how relevant the position i's input is to the current hidden output. There are different methods to decide how attention score is being computed, but the method we turn to is self-attention.

3.2 Self-Attention

Self-attention refers to a special form of attention mechanism. Given inputs $Q, K, V \in R^{T\times d}$ representing queries, keys, and values respectively, self-attention is defined as:

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V$$

3.2.1 Detailed Computation (Per Timestep)

At timestep t, with query q_t, the self-attention output h_t is computed in two steps:

1. **Compute pre-softmax attention scores**:

$$s_{ti} = \frac{1}{\sqrt{d}} q_t k_i^T$$

2. **Compute weighted average of values using softmax-normalized scores**:

$$h_t = \sum_i \text{softmax}(s_{t,:})_i v_i = \frac{\sum_i \exp(s_{ti}) v_i}{\sum_j \exp(s_{tj})}$$

Intuition: Attention scores s_{ti} quantify relevance of keys k_i to the query q_t. Then the values v_i are aggregated proportionally to their computed relevance.

3.2.2 Matrix vs. Decomposed Form

Matrix form (Compact):

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V$$

Decomposed form (Explicit): - Computes attention explicitly per query and key, then aggregates explicitly.

3.2.3 Multi-Head Attention

Multi-head attention introduces multiple attention heads $(Q^{(j)}, K^{(j)}, V^{(j)})$, computed as:

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V$$

Each attention head operates independently, potentially capturing different types of relationships. Outputs from all heads are concatenated to form the final representation.

Note: In some architectures (e.g., Group Query Attention, or GQA), keys (K) and values (V) may be shared across heads, while queries (Q) differ.

4 Transformer

4.1 Query, Key, and Value Vectors

For each position in the sequence with representation X, we derive three vectors:

$$Q = XW_Q \tag{1}$$
$$K = XW_K \tag{2}$$
$$V = XW_V \tag{3}$$

Where W_Q, W_K, and W_V are learned weight matrices. Here, X represents the input embeddings for the first layer, and for subsequent layers, X is the output from the previous layer. These vectors are computed through linear projections of the hidden states.

For multi-head attention, these projections are performed h times with different learned weight matrices, allowing the model to attend to different parts of the input space.

4.2 Transformer Block

A typical transformer block comprises two main sublayers: a self-attention mechanism followed by a position-wise feed-forward network, both wrapped with residual connections and layer normalization. The computations can be formalized as:

$$Z_1 = \text{SelfAttention}(XW_K, XW_Q, XW_V) \tag{4}$$
$$Z_2 = \text{LayerNorm}(X + Z_1) \tag{5}$$
$$H = \text{LayerNorm}(\text{ReLU}(Z_2 W_1) W_2 + Z_2) \tag{6}$$

Where:

- The first equation computes the self-attention mechanism, transforming the input using the query, key, and value projections
- The second equation applies a residual connection (adding the original input X) followed by layer normalization, producing a normalized representation Z_2

- The third equation represents a two-layer position-wise feed-forward network with ReLU activation, again followed by a residual connection and layer normalization
- The output H becomes the input for the next transformer block in the stack

4.3 Masked Self-Attention

In language modeling tasks, we aim to model the conditional probability of each token given its preceding tokens. This presents a fundamental challenge when training transformer models: while the entire sequence is available during training, the model must only access previous tokens when predicting the next token to avoid information leakage. This constraint ensures the model learns to predict based solely on past context, as would be required during generation.

To enforce this causal constraint, we implement masked self-attention, which prevents each position from attending to future positions by using an attention mask M:

$$\text{MaskedSelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}} - M\right)V \tag{7}$$

Where:

$$M_{ij} = \begin{cases} \infty, & \text{if } j > i \\ 0, & \text{if } j \leq i \end{cases} \tag{8}$$

This effectively prevents each position from attending to future positions by setting their attention scores to $-\infty$ before the softmax, resulting in attention weights of 0.

4.4 Summary

Transformer-based language models can be summarized as follows:

- **Decoder-Centric**: Decoder blocks are the fundamental components of language models, typically stacked in large numbers
- **Core Components**:
  - Attention mechanisms (masked self-attention)
  - Layer normalization
  - Multi-layer perceptrons (MLPs)
- **Input Processing**:
  - Token embeddings: Convert discrete tokens into continuous vector representations
  - Positional embeddings: Encode sequential information (e.g., rotary embeddings)
- **Training Objective**: Cross-entropy loss with softmax applied over the sequence

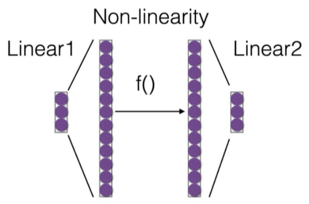
This simple architecture has proven remarkably effective for a wide range of natural language processing tasks, from text generation to translation and beyond.

4.5 Feedforward Layer

A feedforward network (FFN) is composed of two linear transformations separated by a nonlinear activation function. Given an input x, we compute:

$$\mathrm{FFN}(x; W_1, b_1, W_2, b_2) = f\big(xW_1 + b_1\big)\, W_2 + b_2,$$

where W_1 and W_2 are weight matrices, b_1 and b_2 are bias vectors, and f is a nonlinear activation function (for instance, ReLU).



4.6 Computing Components in LLMs

Large language models (LLMs) can be broken down into several components based on the types of operations they perform, each differing in computational cost. Element-wise operations such as layer normalization, residual connections, nonlinear activations, word embeddings, positional embeddings, and the loss function tend to be light and require relatively few FLOPs. In contrast, MLPs and self-attention mechanisms can be more computationally intensive, because they involve matrix multiplications, which typically demand a large number of FLOPs. (Techniques to speed up matrix multiplication were covered in the previous lecture.)

4.7 Original Transformer vs. Modern LLMs

Although Transformers have evolved, their most computationally intensive components have not changed significantly. For example, the earliest Transformer architectures and newer models like LLaMA differ primarily in the placement of the normalization layer (before vs. after attention), the type of normalization (LayerNorm vs. RMSNorm), the nonlinearity (ReLU vs. SiLU), and the positional encoding scheme (sinusoidal vs. RoPE).

5 Connecting the Dots: Key Characteristic in Large Language Models

In this section, we discuss the key characteristic when training large language models (LLMs):

1. **Compute**: Estimating the number of floating-point operations (FLOPs) required for training.

2. **Memory**: Ensuring that the memory usage does not exceed the GPU limits.

3. **Communication**: Optimizing parallelization strategies.

To analyze these three aspects, we need to consider the following:

1. **Parameter Calculation for LLMs**
    - **Memory**: Allocating sufficient memory to store model parameters.
    - **Communication**: When partitioning model weights, understanding their size helps determine an efficient partitioning strategy.
2. **FLOPs Estimation for Training**
    - **Compute**: Estimating the total computational cost required to complete the training process.
3. **Memory Estimation for Training**
    - **Memory**: Managing memory usage for storing parameters, intermediate activations, and optimizer states.
    - **Communication**: If activation partitioning is employed, communication overhead should be considered.

5.1 Parameter Counting in Large Language Models

To effectively analyze the computational and memory requirements of training large language models (LLMs), we first need to calculate the number of parameters in each component of the model. Below, we provide a breakdown of the parameter count using a standard Transformer decoder layer as an example refer to figure 6.

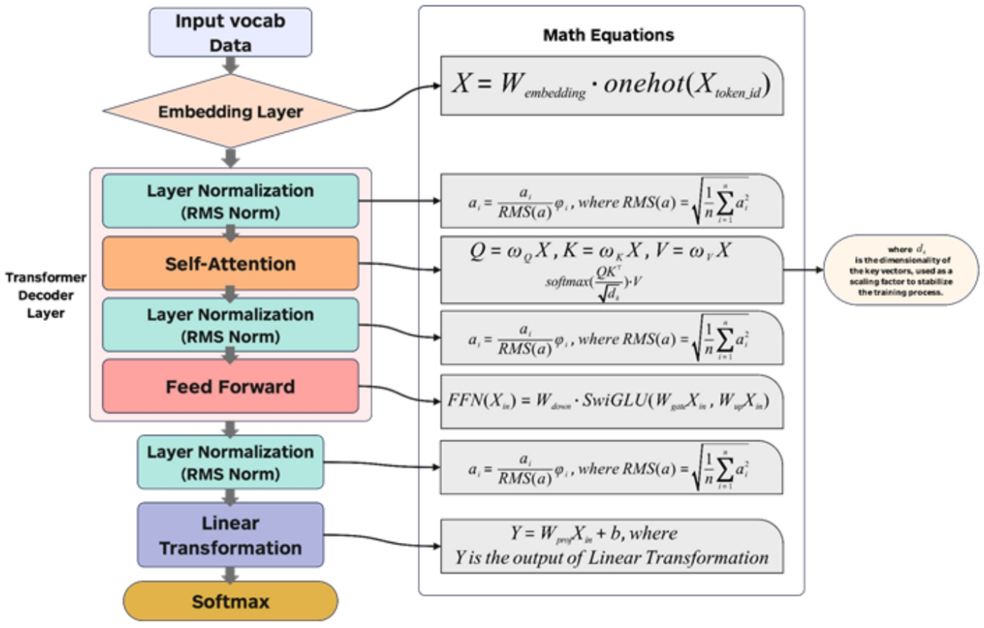

Input vocab Data
Embedding Layer
Transformer Decoder Layer
Layer Normalization (RMS Norm)
Self-Attention
Layer Normalization (RMS Norm)
Feed Forward
Layer Normalization (RMS Norm)
Linear Transformation
Softmax
Math Equations
$X = W_{embedding} \cdot onehot(X_{token\_id})$
$a_i = \frac{a_i}{RMS(a)} \varphi_i$, where $RMS(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$
$Q = \omega_Q X, K = \omega_K X, V = \omega_V X$
$softmax(\frac{QK^T}{\sqrt{d_k}}) \cdot V$
where d_k is the dimensionality of the key vectors, used as a scaling factor to stabilize the training process.
$a_i = \frac{a_i}{RMS(a)} \varphi_i$, where $RMS(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$
$FFN(X_{in}) = W_{down} \cdot SwiGLU(W_{gate} X_{in}, W_{up} X_{in})$
$a_i = \frac{a_i}{RMS(a)} \varphi_i$, where $RMS(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$
$Y = W_{proj} X_{in} + b$, where
Y is the output of Linear Transformation

Figure 6: Transformer Decoder Layer Parameter Shapes

Model Dimensions We define the following variables:

- b – Batch size
- s – Sequence length
- h – Hidden dimension
- d_{ff} – Intermediate size in the feed-forward network
- V – Vocabulary size

5.2 Component-wise Parameter Calculation

Embedding Layer
The embedding layer maps each token in the vocabulary to a dense representation:

$$\text{Parameters} = V \times h$$

Layer Normalization
Each layer normalization module contains learnable scale and bias parameters:

$$\text{Parameters per LayerNorm} = 2h$$

Since there are three normalization layers in a typical Transformer block, the total parameter count for normalization is:

$$\text{Total LayerNorm Parameters} = 3 \times (2h) = 6h$$

Self-Attention Mechanism
The self-attention mechanism consists of weight matrices for query, key, value, and output projection:

$$\text{Parameters} = 4h \times h = 4h^2$$

Feed-Forward Network (FFN)
The FFN consists of two linear transformations with an intermediate hidden size:

$$\text{Parameters} = 3 \times h \times d_{\text{ff}}$$

Final Linear Transformation and Softmax
The final linear transformation maps hidden states back to the vocabulary space:

$$\text{Parameters} = V \times h$$

Total Parameter Count
Summing up all components:

$$\begin{aligned}\text{Total Parameters} &= V \times h + 4h^2 + 3h \times d_{\text{ff}} + 6h + V \times h \\ &= 2Vh + 4h^2 + 3hd_{\text{ff}} + 6h\end{aligned}$$

This equation provides a structured way to estimate the total number of parameters in a Transformer decoder layer, which is crucial for understanding model size and resource requirements.

For a visual representation of the parameter breakdown, refer to the figure 7 below

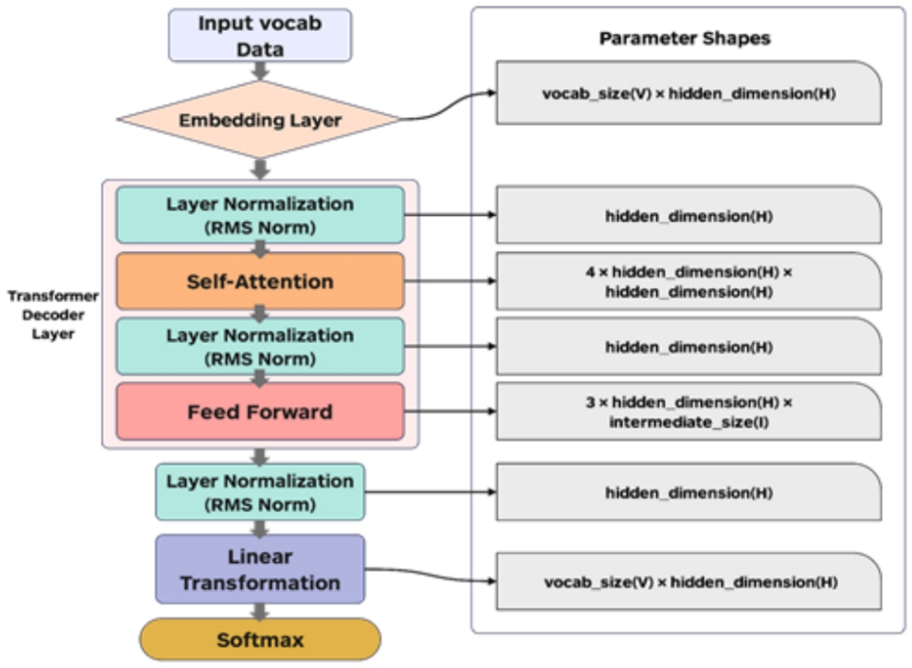

Input vocab Data
Embedding Layer
Layer Normalization (RMS Norm)
Self-Attention
Layer Normalization (RMS Norm)
Feed Forward
Transformer Decoder Layer
Layer Normalization (RMS Norm)
Linear Transformation
Softmax
Parameter Shapes
vocab_size(V) × hidden_dimension(H)
hidden_dimension(H)
4 × hidden_dimension(H) × hidden_dimension(H)
hidden_dimension(H)
3 × hidden_dimension(H) × intermediate_size(I)
hidden_dimension(H)
vocab_size(V) × hidden_dimension(H)

Figure 7: Mathematical Equations for Transformer Computations

These figures illustrate the relationship between different components in a Transformer decoder layer and their corresponding parameter sizes.

5.2.1 SwiGLU

SwiGLU is an advanced feed-forward activation mechanism used in Transformer models to enhance learning capabilities. The general formula for SwiGLU is:

$$\text{SwiGLU}(x) = \text{Swish}(xW_1 + b_1) \odot (xW_2 + b_2)$$

where $\odot$ represents element-wise multiplication.

Swish Activation Function Swish is a smooth activation function applied to one branch in the computation:

$$\text{Swish}(z) = z \cdot \sigma(z)$$

where $\sigma(z)$ is the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Advantages of SwiGLU

- SwiGLU helps the model capture more complex patterns by selectively gating information.
- The Swish activation function is smoother than traditional activation functions like ReLU, leading to better gradient flow.

5.3 Scaling Up: Where is the Potential Bottleneck?

As large language models (LLMs) scale up, certain components in the architecture can become computational or memory bottlenecks. One such critical component is Layer Normalization (RMS Norm), which appears multiple times within a Transformer block.

Why is Layer Normalization a Bottleneck?

Layer normalization is used to stabilize training and improve convergence, but it also introduces computational overhead due to the need for calculating the root mean square (RMS) statistics across each sequence. Below are the key reasons why it could become a bottleneck:

- **Frequent Computation:** Layer normalization is applied multiple times per layer—before self-attention, before feed-forward networks, and after key operations. This frequent execution increases latency.
- **Memory Overhead:** Each application of Layer Normalization requires storing temporary statistics (e.g., mean, variance) and applying scale and shift parameters, contributing to increased memory consumption.
- **Inefficient Parallelization:** Unlike matrix multiplications in self-attention or feed-forward networks, which can be highly parallelized on GPUs, layer normalization requires reductions across feature dimensions, which limits parallel efficiency.

Visual Representation of Potential Bottleneck

The following diagram illustrates the parameter allocation in a Transformer decoder layer. The frequent occurrences of Layer Normalization (RMS Norm) highlight its role as a potential computational bottleneck:

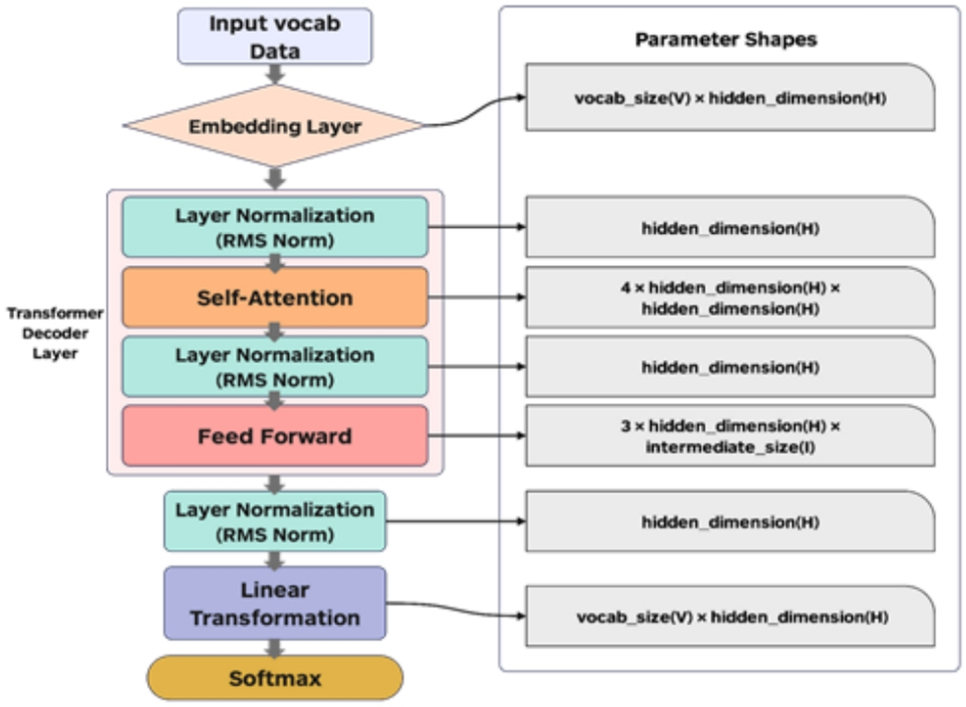

Input vocab Data
Embedding Layer
Transformer Decoder Layer
Layer Normalization (RMS Norm)
Self-Attention
Layer Normalization (RMS Norm)
Feed Forward
Layer Normalization (RMS Norm)
Linear Transformation
Softmax
Parameter Shapes
vocab_size(V) × hidden_dimension(H)
hidden_dimension(H)
4 × hidden_dimension(H) × hidden_dimension(H)
hidden_dimension(H)
3 × hidden_dimension(H) × intermediate_size(I)
hidden_dimension(H)
vocab_size(V) × hidden_dimension(H)

Figure 8: Potential Bottlenecks in Scaling Up LLMs: Layer Normalization is a frequently used component.

To address the potential bottleneck of Layer Normalization, researchers have proposed alternative normalization techniques:

- RMSNorm: A simplified variant of Layer Normalization that eliminates the need for computing mean statistics.
- Post-LayerNorm Variants: Modifying the Transformer architecture to apply normalization at different points in the computation to reduce its impact.
- Faster Normalization Layers: Implementing fused or hardware-optimized versions of normalization layers to improve efficiency.

Understanding bottlenecks in LLMs is crucial for optimizing their training and inference efficiency, particularly as models continue to grow in size.

6 FLOPS Compute

6.1 Estimate the Compute: FLOPs

To estimate the computational cost of matrix multiplications in large language models (LLMs), we analyze the number of floating-point operations per second (FLOPs) required for multiplying two matrices.

Matrix Multiplication FLOPs Calculation

Given two matrices of dimensions:

1. A of size $m \times n$

2. B of size $n \times h$

The resulting matrix $C = A \times B$ will have dimensions $m \times h$. Each element of C is computed as a dot product of a row from A and a column from B.

The FLOPs required for computing a single element in C is:

$$2n - 1$$

where:

- n multiplications are needed per row-column dot product.
- $(n-1)$ additions are required to sum the products.

Since C has $m \times h$ elements, the total number of FLOPs required for this matrix multiplication is:

$$\text{FLOPs} = m \times h \times (2n - 1)$$

For large-scale computation, we approximate this as:

$$\text{FLOPs} \approx 2m \times n \times h$$

The following figure illustrates matrix multiplication, showing how each element in the resulting matrix C is computed:

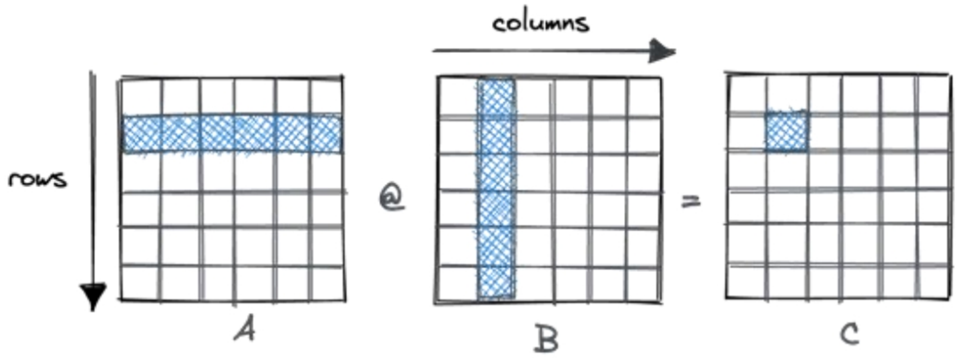


Figure 9: Illustration of matrix multiplication FLOPs computation.

The FLOPS for compute are done using only matmul as this dominates over all other operations. The number of FLOPS of matmul is as follows:

$$2 * m * h * n$$

where m,n,h are the dimensions of the matrices.

We will now go through how to compute FLOPS for LLama 2 7B. To do so we need the following parameters/hyperparameters:

- Batch Size: b
- Sequence length: s
- The number of attention heads: n
- Hidden state size of one head: d
- Hidden state size: h (h = n * d)
- SwiGLU proj dim: i (mostly 4 * h)
- Vocab size: v

Now let us go through the calculation step by step for forward calculation during training. For the first step we want to feed the input which only has embeddings operations which is negligible and are ignored.

Input	**Output Shape**	**FLOPs**
X	(b, s, h)	0

Table 1: Inputs

The next operation is self attention which follows the sequence shown below along with the corresponding FLOPS required for each of them. We see that the first step is Q,K,V multiplication which is matrix multiplication. RoPE is positional embedding calculation. The next part is softmax in which the first term is from QxK multiplication and the second term is for the softmax computation. Next is PxV which is another matmul. And next we outer projection. Here the terms with s^2 are very bad as this limits out ability to model longer sequences. Whereas parameters like h^2 are controllable by us even if they are square. Next is residual connections followed by the output.

Input	**Output Shape**	**FLOPs**
XW$_Q$, XW$_K$, XW$_V$	(b, s, h)	$3 \times 2bsh^2$
RoPE	(b, n, s, d)	$3bsnd$
P = Softmax(QK$^T/\sqrt{d}$)	(b, n, s, s)	$2bs^2nd + 3bs^2n$
PV	(b, n, s, d)	$2bs^2nd$
AW$_O$	(b, s, h)	$2bsh^2$
Residual Connection	(b, s, h)	bsh
Output from self-attention	(b, s, h)	0

Table 2: Self-Attention

This last table is for the computation that happens during the MLP operations which for Llama is SwiGLU. Here the 1st and 4th steps are matmuls and are also the most time consuming steps. They also have a square term as $i = 4 * h$ typically.

Input	**Output Shape**	**FLOPs**
XW$_{gater}$, XW$_{up}$	(b, s, i)	$2 \times 2bshi$
Swish Activation	(b, s, i)	$4bsi$
Element-wise *	(b, s, i)	bsi
XW$_{down}$	(b, s, h)	$2bshi$
RMS Norm	(b, s, h)	$4bsh + 2bs$

Table 3:

Putting everything together we get this equation for FLOPs for forward training:

$$\text{Total FLOPs} \approx \#\text{num\_layers} \times \left(6bsh^2 + 4bs^2h + 3bs^2n + 2bsh^2\right) + \#\text{num\_layers} \times (6bshi) + 2bshv$$

As an example if we substitute these values for training:

- Batch size: b=1
- Sequence length: s=4096
- The number of attention heads: n=32
- Hidden state size of one head: d=128
- Hidden state size: h =4096
- SwiGLU proj dim: i=11008
- Vocab size: v=32000
- The number of layers: N=32

we get the total FLOPs needed as 63 TFLOPs during a single forward pass.

Breaking it down into per layer for Llama training using 192.17 TFLOPs:

- Embedding Layer: 1.676
- Normalization: 0.007
- Residual: 0.003
- Attention: 41.276
- MLP (Multi-Layer Perceptron): 55.361
- Linear: 1.676

We see that the primary bottleneck is MLP especially when we want to increase by dimension. Whereas increasing by sequence length attention will be the bottleneck.

7 Scribe Contribution

1. Leduo Chen: section 5 and 6
2. Chao-Jung Lai: section 4.5
3. Ohm Rishabh: section 6
4. Raunak Sengupta: section 2
5. Vibha: section 1 and 2
6. Harshit: section 1
7. You Zhou: section 4
8. Kanaad: section 3
9. Jerry: Section 3