Understanding the Performance of Transformer Inference

by

Anne Ouyang

S.B., Computer Science and Engineering,
Massachusetts Institute of Technology (2023)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by:	Anne Ouyang Department of Electrical Engineering and Computer Science May 12, 2023
Certified by:	Jonathan Ragan-Kelley Esther & Harold E. Edgerton Assistant Professor Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Understanding the Performance of Transformer Inference

by

Anne Ouyang

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The state of the art results in natural language processing tasks have been obtained by scaling up transformer-based machine learning models, which can have more than a hundred billion parameters. Training and deploying these models can be difficult and extremely expensive, and performance engineering efforts to improve the latency and throughput of these models are crucial in enabling widespread applications.

We developed an analytical model for studying the performance of transformer inference and combined it with empirical studies using existing frameworks to gain insights into the performance characteristics of transformers and efficiency of existing implementations. The findings revealed the contribution of the different operations to the total parameter count, floating-point operations count, activation memory. A comparison between prefilling and generation stages highlighted differences in performance characteristics, with generation being slower due to low arithmetic intensity operations. Empirical studies with existing implementations on single GPUs showed that the implementation has a high roofline utilization but low FLOPs utilization during the generation stage, which indicates that implementation is reasonably efficient, but the low arithmetic operations during autoregressive generation is an inherent limitation of transformer-based architectures.

We also experimented with various parallelism strategies for different inference workloads and distilled our observations as recommendations for effectively using parallelism. We found that the best parallelism strategy depends on the specific workloads (batch size and input and output sequence lengths). We also found that model parallelism can be useful for reasons beyond fitting the model in the GPU memory — for example, in the case where a model fits in a single GPU, in the generation stage, tensor parallelism can decrease the latency for small batch settings.

We hope that a comprehensive understanding of the performance characteristics and trade-offs can serve as a guide for researchers to optimize hardware resource utilization and enhance the efficiency of large language models.

Thesis Supervisor: Jonathan Ragan-Kelley
Title: Esther & Harold E. Edgerton Assistant Professor

Acknowledgments

I am deeply grateful to my thesis advisors, Professor Jonathan Ragan-Kelley and William Brandon, for their guidance, expertise, and continuous support throughout this research journey. Their active participation in discussions, wealth of knowledge, and inspiring ideas have been instrumental in shaping this work. I also deeply appreciate the valuable professional advice and interesting discussions in the fields of machine learning systems and compilers.

I would like to extend my heartfelt thanks to Aniruddha Nrusimha and Rameswar Panda for their helpful discussions, which have provided valuable insights and broadened my understanding of the subject matter. I am also immensely grateful to Rameswar and the MIT-IBM Watson AI Lab for generously providing GPU resources for this project.

I extend my gratitude to all the professors and teaching assistants at MIT whose interesting classes and engaging lectures have taught me a lot and made me greatly enjoy the field of computer science.

I would like to thank my friends who have made my time at MIT enjoyable and memorable through their support, companionship, and our shared adventures. Their presence has made this journey more meaningful, and I look forward to more adventures together.

I am grateful to my family for their unconditional love and unwavering support. To my parents, thank you for not only supporting my education but also inspiring my passion in computer science. I am immensely grateful for the sacrifices you have made to ensure that I had the best opportunities to learn and grow — you have always gone above and beyond to support my aspirations. I am grateful for the countless conversations we have had, the guidance you have provided, and the unwavering faith you have shown in my abilities.

To my grandparents, thank you for all the wonderful memories we have shared during my childhood in Beijing. Those years were filled with joy, laughter, and countless precious moments that have left an indelible mark on my heart. From exploring the vibrant streets of Beijing to savoring delicious meals as a family, every moment was infused with love and warmth.

To my brother Dougy Ouyang. Your fun-loving nature and vibrant personality make you an absolute joy to be around, and your sense of humor has kept me motivated during the most challenging times — your ability to make light of even the most stressful moments is truly a blessing. I cherish the relationship we share. Thank you for always being there for me and for making life brighter.

Lastly, I would like to express my appreciation to Gustav Mahler, whose music has been a constant source of inspiration, solace, and motivation throughout my years at MIT. Thank you, Gustav Mahler, for sharing your gift with the world and enriching our lives through your music.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem and Motivation

State-of-the-art results on a wide variety of natural language processing tasks have been obtained by scaling transformer-based language models; however, these models that grow rapidly in size over the years have resulted in a high cost in computation resources. Table 1.1 includes examples of models, their sizes in the number of parameters, and time of release.

Table 1.1: Examples of known Large Language Models, their parameter counts, and the time of release.

Name	Number of Parameters	Time of Release
BERT Large [19]	340M	October 2018
GPT-2 [22]	1.5B	February 2019
Megatron-LM [20]	8.3B	September 2019
T5 [23]	11B	October 2019
Turing-NLG [17]	17B	February 2020
GPT3 [18]	175B	May 2020
Megatron-Turing NLG [25]	530B	February 2022
OPT-175B [27]	175B	May 2022
BLOOM [2]	176B	June 2022

As the sizes of language models increase, deploying these models have become difficult and expensive, and performance engineering efforts to improve the latency and throughput of these models are crucial in enabling applications. It is important to have not only efficient implementations that result in high hardware utilization on a single accelerator but also effective parallelization across multiple accelerators.

For our project, we benchmarked existing transformer libraries such as HuggingFace Transformers and FasterTransformer from Nvidia for their inference latency for different batch sizes and input and output sequence lengths. We also developed an analytical performance model based on maximum hardware utilization as a comparison baseline to analyze the efficiency of these libraries.

Most machine learning models in the past fit in the memory of a single GPU without issues; however, as the models become more complex and larger, a single GPU is no longer sufficient, and computation must move into the distributed space. The opportunity to parallelize work across multiple machines enables performance boosts, and, parallelization, combined with intelligent partitioning, allows us to use large models that would otherwise be impossible to fit on a single GPU. Furthermore, parallelizing models that fit in a single GPU can also offer performance benefits. We used the analytical performance model along with multi-GPU experiments to explore different parallelization strategies and developed recommendations for efficiently parallelizing transformer-based large language models for inference.

1.2 Project Setup

1.2.1 Architecture

We focused our project on GPT-style [22] decoder-only model architectures. Since we do not have access to the GPT weights, we instead used the class of OPT (Open Pre-trained Transformers) [27] models for our experiments since they come in a variety of sizes ranging from 125 million to 175 billion parameters, are publicly available, and many existing implementations of transformer libraries support these model architectures.

1.2.2 Compute Resources

We used the MIT Satori cluster [13] for running our experiments. Each node on Satori has 4 V100 32GB memory GPU cards connected by an NVLink2 network that supports 200GB/s bi-directional transfer, and the nodes are connected by a 100GB/s Infiniband network with microsecond user space latency.

1.3 Contributions

We developed an analytical model based on peak hardware utilization to analyze the performance of transformer inference for GPT-like architectures. The analytical model was used in conjunction with empirical studies with existing frameworks to understand the state-of-the-art performance. Through a combination of these, we were able to provide a comprehensive understanding of the performance characteristics and trade-offs involved and develop recommendations for effective parallelization of large language models. We hope that this thesis can guide researchers and practitioners in optimizing the utilization of hardware resources and improving the efficiency of large language models.

1.3.1 Main Takeaways

In this section, we describe some of the main observations and takeaways from this work.

Parameter Counts

- In a transformer block, MLP parameters account for around 66% of the total, and attention parameters account for around 33%.
- As model sizes get larger, the percentage of embedding parameters becomes smaller since the parameters from the repeated transformer blocks will dominate.

FLOP Counts

- Most operations scale linearly with sequence length and quadratically with the hidden dimension. There are a few operations in the attention mechanism (QK, multV, softmax) that scale quadratically with sequence length.

- The total number of FLOPs scales roughly linearly with the sequence length, which suggests that the operations that scale quadratically with the sequence length do not account for a significant part of the total FLOPs.

- the MLP operation contributes to most of the FLOPs.

- The KQV computations in the attention layers dominate most of the time, followed by attention_out.

- The non-KQV computations, which scale quadratically with sequence length, do not account for a significant number of FLOPs. In smaller models, they account for around 10% at most and are negligible in larger models.

- The number of FLOPs spent on embeddings becomes a less significant fraction as the model gets larger.

Prefilling vs. Generation

- For generation, the number of FLOPs for generation grows quadratically as the sequence length increases, and it can be more than 2 orders of magnitude greater than prefilling for the same sequence length.

- When using a KV cache, the number of FLOPs for generation becomes roughly the same as the number of FLOPs for prefilling for a given sequence length. However, in practice, we will see that generation is still a lot slower than prefilling for the same sequence length even if we are using a KV cache due to low arithmetic intensity operations.

Hardware Utilization with Huggingface Transformers and Nvidia Faster-Transformer – Single GPU

Prefilling

- The shapes of the analytical utilization and FLOPS utilization curves look very similar, and for each (batch size, sequence length) data point, the FLOPs utilization is only lower than the analytical utilization by a few percent. This means that the prefilling stage workloads generally have high arithmetic intensity.
- As the sequence length increases, the analytical utilization for all batch sizes converges to around 46%, and the FLOPs utilization for all batch sizes converges to around 42%.
- For almost all batch sizes (except batch size 1), both the FLOPs and analytical utilization peak at a sequence length of around 128 - 256, which is about 10% - 20% of the maximum context length. The utilization decreases as the sequence lengths grow longer than that range.
- The highest FLOPs utilization we observed for prefilling is 78% for batch size 1024 and input sequence length 16. The highest analytical utilization we observed for prefilling is 81% also for batch size 1024 and input sequence length 16. This indicates that the implementation of Nvidia FasterTransformer is reasonably efficient at utilizing the hardware.
- For most of the different (batch size, input sequence length) configurations, the ratio of the HFT latency to the FT latency is about 1.5, which indicates that Huggingface Transformer is 1.5x slower than Nvidia FasterTransformer for most of the prefilling workloads.
- Workloads with small batch sizes and short sequence lengths have a higher ratio since they have fewer compute-bound matrix multiplication operations. The latency of the matrix multiplication operations should not be too different for Huggingface Transformers and Nvidia Transformer, since they should both be

using optimized CUDA kernels for matrix multiplication, which is a well-studied problem.

- Huggingface Transformers runs out of memory earlier than Nvidia FasterTransformer, as indicated by "N/A"s in Table 3.1. This indicates that FT is more efficient at memory usage than HFT.

Generation

- Unlike the prefilling stage, the shapes of the FLOPs and analytical utilization curves are very different. The FLOPs utilization is quite low while the analytical utilization is reasonably high. This is consistent with our analysis that the operations in the generation stage are mostly memory bound (compute bound operations would show similar FLOPs and analytical utilization). From this disparity in FLOPs vs analytical utilization, we can also conclude that the implementation of Nvidia FasterTransformer is also reasonably efficient at utilizing the hardware, but we are limited by the memory-bound operations to achieve a high FLOPs utilization.

- The FLOPs utilization decreases for all batch sizes as the sequence length increases. This is consistent with our analysis that the autoregressive generation steps are memory bound. With more generation steps, the FLOPs utilization decreases as there are more memory bound operations. Overall, the generation FLOPs utilization is quite low.

- The analytical utilization becomes higher as the sequence length grows longer for all batch sizes.

- The ratio HFT / FT Latency ratio decreases as we increase the batch size. This is consistent with our expectation that operations with higher arithmetic intensity will show a smaller gap between the latency of the different implementations.

- With the same batch size, the HFT / FT Latency ratio is similar.

- For the different (batch size, input sequence length) configurations, the ratio of the HFT latency to the FT latency does not stay fairly constant as in the prefilling case. It is around 3 for most of the workloads, indicating that HFT is less efficient than FT at the generation stage.

- Compared to prefilling, both HFT and FT are able to support larger batch size and sequence length combinations. We think this is because in the attention operation when using a KV cache during generation, we save on some of the matrices with sequence length as dimensions since some of them will become 1 with the KV cache.

Multi-GPU Inference

Model Fits in a Single GPU

- There is no reason to use model parallelism during the prefilling stage to minimize latency if the model fits in a single GPU. Data parallelism is useful for decreasing latency.

- In the generation stage, using tensor parallelism in small batch settings is beneficial for decreasing latency; however, this doesn't scale infinitely. As the communication costs of tensor parallelism grow quickly, we will hit diminishing returns for using GPUs for tensor parallelism until eventually the communication overhead exceeds the computation savings. In large batch settings, data parallelism by replicating the model is best for minimizing latency.

Model Doesn't Fit in a Single GPU

- For prefilling, to minimize latency, tensor parallelize the model on as few GPUs as possible and then replicate the same model and parallelization strategy across the remaining GPUs. This is a generalization of the case where the model fits in a single GPU — if a model fits in N GPUs, tensor parallelize across N GPUs. Replicate the model and use the same parallelization strategy combined with data parallelism on the remaining GPUs if the total number of GPUs is a factor of N.

- In the generation stage, for small batch sizes, tensor parallelism on as many GPUs as possible is useful for minimizing latency. In large batch settings, if a model fits in at least N GPUs, pipeline parallelize across exactly N GPUs. Replicate the model and use the same parallelization strategy combined with data parallelism on all the remaining GPUs.

1.4 Thesis Outline

Chapter 2 describes the architecture we are focusing on and the implementation of the analytical model, including counting parameters, counting FLOPs, differentiating between the prefilling and generation stage during inference, using a KV cache, estimating the activation memory and maximum supported batch size, and estimating the speed of light latency in two modes — either using only FLOPs or with a roofline model that takes into account memory-boundedness. In addition to describing the analytical model, Chapter 2 also includes takeaways and results obtained with the analytical model. Chapter 3 shows the results of transformer inference empirically with 2 different libraries (Huggingface Transformers, Nvidia FasterTransformer) and getting their hardware utilization, as well as comparing the two implementations. Chapter 4 describes the different parallelism strategies for transformer inference, extends the analytical model to a multi-GPU setting, summarizes the results of running multi-GPU inference empirically, and provides recommendations on efficiently parallelizing transformer inference for different workloads. Chapter 5 describes additional parallelism strategies not explored in our work and surveys other existing frameworks for transformer inference or model parallelism. Chapter 6 summarizes the results and describes the future directions of this work.

Chapter 2

Analytical Model

2.1 Transformer and the GPT Architecture

Transformers are a type of neural network architecture used in a variety of natural language processing (NLP) tasks. They were first introduced in the 2017 paper "Attention is All You Need" [26] and have since become a key component of many state-of-the-art NLP systems. There are many pretrained language models based on transformers. We will be focusing our analyses on the decoder-only GPT [22] family of models; however, since the GPT models are not available to the public, we will be instead using OPT [27], which is a family of open-sourced models that are architecturally similar to GPT and thus have similar performance characteristics.

The following equations are from the GPT paper [22] describing the model architecture. As shown in Figure 2-1, the model consists of text and position embedding operations, followed by repeated transformer blocks, and then a downstream task. Each transformer block has a multi-headed self attention operation, a layer norm operation, a fully connected feed-forward network, and another layer norm operation.

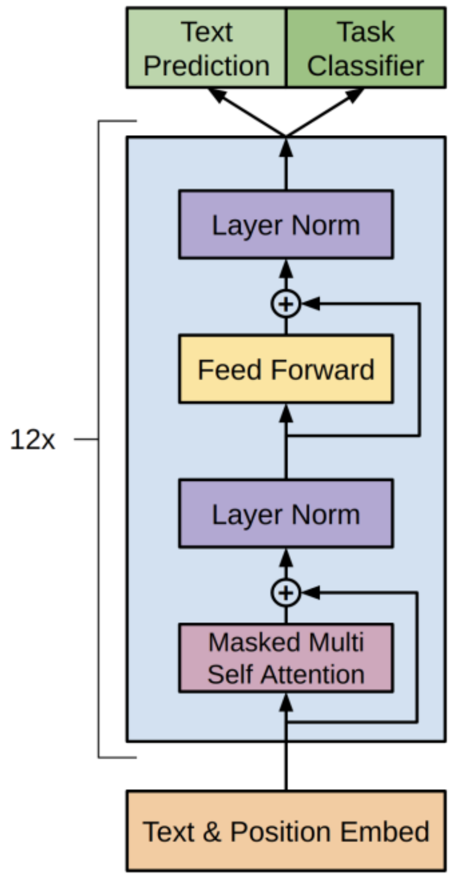

Figure 2-1: Diagram of the GPT architecture from the GPT-2 Paper. The diagram shows word and positional embeddings, repeated transformer blocks, and downstream tasks. There are four components within the residual block — Multi-Headed Attention, Feed Forward, and two LayerNorms. There are residual connections in both the MHA and the FFN parts of the network.

$$\begin{aligned} h_0 &= UW_e + W_p \\ h_l &= \text{transformer\_block}(h_{l-1}) \forall i \in [1, n] \\ P(u) &= \text{softmax}(h_n W_e^T) \end{aligned}$$

$U = (u_{-k}, ..., u_{-1})$ is the context vector of tokens, n is the number of repeated transformer blocks, $W_e \in$ is the token embedding matrix, and W_p is the positional embedding matrix.

Transformer Block

Multi-Headed Attention (MHA): From the equations in the paper "Attention is All You Need" [26] that introduced the transformer architecture, the multi-head attention mechanism is described as

$$\text{multihead}(Q, K, V) = \text{concat}(\text{head\_1}, ..., \text{head\_h})W^O$$

where

$$\text{head\_i} = \text{attention}(XW_i^Q, XW_i^K, XW_i^V)$$

and

$$\text{attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Feed Forward: The feedforward operation is a two-layer multi-layer perceptron (MLP)

$$FFN(x) = max(0, XW_1 + b_1)W_2 + b_2$$

2.1.1 Operation Breakdown and Naming Conventions

This subsection breaks down the transformer operations and defines the naming conventions that will be used in the rest of the analysis.

`word_embedding` and `positional_embedding`

$$h_0 = UW_e + W_p$$

Our analysis of the positional embedding will be mostly agnostic to the specific strategy used and we will model it as an operation proportional to the size of the W_p matrix.

`attention_K`

$$K_i = X * W_i^K$$

`attention_V`

$$V_i = X * W_i^V$$

`attention_Q`

$$Q_i = X * W_i^Q$$

`attention_QK`

$$Q * K^T$$

`attention_softmax`

$$\mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})$$

`attention_multV`

$$\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

`attention_out`

$$\text{concat}(\text{head\_1}, ..., \text{head\_h})W^O$$

`layernorm`

$$\text{LayerNorm}(X)$$

We do not make a distinction between the layernorm after the attention block and the layernorm after the MLP block in our analysis, since they will be doing the same operations on inputs of the same size.

`mlp1` This is the expansion that expands the hidden dimension from d_{model} to d_{ff}

$$XW_1 + b_1$$

`mlp2` This is the contraction that brings the hidden dimension back to d_{model}

$$max(0, XW_1 + b_1)W_2 + b_2$$

2.1.2 Model Family and Hyperparameters

The GPT and OPT families of models consist of models with varying numbers of heads, layers, embedding sizes. These hyperparameters are summarized in Table 2.1, and these notations will be the convention for the rest of the analysis.

In the GPT (GPT-2) and OPT architectures

- $n_{vocab} = 50257$
- $n_{ctx} = 1024$

Table 2.1: Hyperparameters and their notations for the GPT / OPT model architecture. These notations will be used in the rest of the analyses in this work.

Notation	**Description**
n_{vocab}	Vocab size
n_{ctx}	Maximum sequence length
n_{layer}	Number of times the transformer block is repeated
n_{head}	Number of heads
d_{model}	Attention embedding size
d_k, d_v, d_{head}	Attention weights sizes
d_{ff}	Feedforward embedding size

Table 2.2: OPT family of models and some of their hyperparameters: number of repeated transformer blocks (layers), number of attention heads, and the hidden dimension.

Model	n_{layer}	n_{head}	d_{model}
opt-125m	12	12	768
opt-350m	24	16	1024
opt-1.3b	24	32	2048
opt-2.7b	32	32	2560
opt-6.7b	32	32	4096
opt-13b	40	40	5120
opt-30b	48	56	7168
opt-66b	64	72	9216
opt-175b	96	96	12288

- $d_k = d_v = d_{head} = d_{model}/n_{head}$
- $d_{ff} = 4 * d_{model}$

2.2 Parameter Counting

2.2.1 Parameters by Operation

Following our conventions of breaking down the operations, and omitting those without parameters associated with them, we can t `word_embedding` $W_e : n_{vocab} * d_{model}$

`positional_embedding`: $W_p : n_{ctx} * d_{model}$

Each repeated transformer block has its own set of weights, and each of the attention heads (i) has the weights matrices W_i^K, W_i^Q, W_i^V associated with constructing its own K, Q, V matrices. In each transformer block, the heads are concatenated and then multiplied by W_o to get the attention output projection.

`attention_K`: $W_i^K : d_{model} * d_k$ per head per transformer block (layer)

`attention_V`: $W_i^V : d_{model} * d_v$ per head per layer

`attention_Q`: $W_i^Q : d_{model} * d_k$ per head per layer

`attention_out`: $W_i^Q : n_{head} * d_v * d_{model}$ per layer

`layernorm`: $2 * d_{model}$ per layernorm operation per layer
There are 2 layernorms per transformer block. The parameters consist of the scale factor and offset, each of size d_{model}.

We will ignore the bias for the MLPs, since they are vectors of negligible size compared to the weight matrices.

`mlp1`: $W_1 : d_{model} * d_{ff}$

`mlp2`: $W_2 : d_{ff} * d_{model}$

The embedding weights (positional and word) have a total size of

$$\begin{aligned}
&\text{word\_embedding} + \text{positional\_embedding} \\
&= n_{vocab} * d_{model} + n_{ctx} * d_{model} \\
&= (n_{vocab} + n_{ctx}) * d_{model}
\end{aligned}$$

Summing across transformer blocks, the LayerNorm weights have a total size of

$$\begin{aligned}
&2 * 2 * d_{model} * n_{layer} \\
&= 4 * d_{model} * n_{layer}
\end{aligned}$$

Summing across attention heads and transformer blocks, the attention weights have a total size of

$$\begin{aligned}
&((\text{attention\_K} + \text{attention\_V} + \text{attention\_Q}) * n_{head} + \text{attention\_out}) * n_{layer} \\
&= ((d_{model} * d_k + d_{model} * d_v + d_{model} * d_k) * n_{head} + n_{head} * d_v * d_{model}) * n_{layer} \\
&= (3 * d_{model} * d_{head}) * n_{head} + n_{head} * d_{head} * d_{model}) * n_{layer} \\
&= 4 * d_{model}^2 * n_{layer}
\end{aligned}$$

Summing across transformer blocks, the MLP weights have a total size of

$$\begin{aligned}
&(\text{mlp1} + \text{mlp2}) * n_{layer} \\
&= (d_{model} * d_{ff} + d_{ff} * d_{model}) * n_{layer} \\
&= (2 * d_{model} * 4 * d_{model}) * n_{layer} \\
&= 8 * d_{model}^2 * n_{layer}
\end{aligned}$$

Table 2.3: Parameter distribution for the OPT family of models by operation. The operations are broken down as embeddings, attention KQV, attention out, MLP, and layernorm

Model	Word Emb.	Positional Emd.	Attn K,Q,V	Attn Out	MLP	LayerNorm
opt-125m	3.86E+07	7.86E+05	2.12E+07	7.08E+06	5.66E+07	3.69E+04
opt-350m	5.15E+07	1.05E+06	7.55E+07	2.52E+07	2.01E+08	9.83E+04
opt-1p3b	1.03E+08	2.10E+06	3.02E+08	1.01E+08	8.05E+08	1.97E+05
opt-2p7b	1.29E+08	2.62E+06	6.29E+08	2.10E+08	1.68E+09	3.28E+05
opt-6p7b	2.06E+08	4.19E+06	1.61E+09	5.37E+08	4.29E+09	5.24E+05
opt-13b	2.57E+08	5.24E+06	3.15E+09	1.05E+09	8.39E+09	8.19E+05
opt-30b	3.60E+08	7.34E+06	7.40E+09	2.47E+09	1.97E+10	1.38E+06
opt-66b	4.63E+08	9.44E+06	1.63E+10	5.44E+09	4.35E+10	2.36E+06
opt-175b	6.18E+08	1.26E+07	4.35E+10	1.45E+10	1.16E+11	4.72E+06

Adding up the components, the total number of parameters is

$$\begin{aligned}
&\text{embeddings} + \text{attention} + \text{mlp} + \text{layernorm}\\
&= (n_{vocab} + n_{ctx}) * d_{model} + 4 * d_{model}^2 * n_{layer} + 8 * d_{model}^2 * n_{layer} + 4 * d_{model} * n_{layer}\\
&= 12 * d_{model}^2 * n_{layer} + (n_{vocab} + n_{ctx}) * d_{model} + 4 * d_{model} * n_{layer}
\end{aligned}$$

2.2.2 Parameter Distribution

Table 2.3 summarizes the distribution of parameters among the different components for OPT models of different sizes, and Figure 2-2 shows the breakdown by percentage.

We can make several observations here:

1. LayerNorm contributes a negligible number of parameters

2. As model sizes get larger, the percentage of embedding parameters becomes smaller, since the parameters from the repeated transformer blocks will dominate

3. In a transformer block, MLP parameters account for around 66% of the total, and attention parameters account for around 33%.

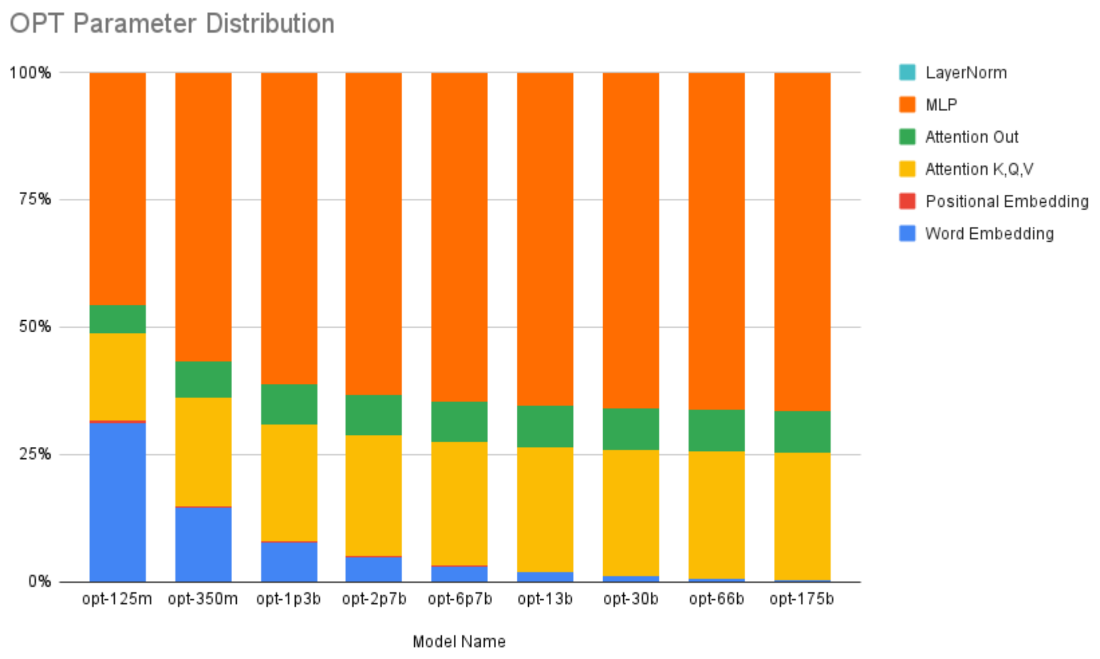

Figure 2-2: Parameter distribution by operation for the OPT family of models of different sizes from 125M to 175B. The distributions are shown as percentages in the bar graph.

2.3 FLOPs Counting

Similar to analyzing the parameter count and breakdown, we will now proceed to analyze the number of floating point operations (FLOPs) in transformer inference. We will base our calculations on the assumption that for a matrix multiplication operation $C[M, K] = A[M, N] * B[N, K]$, the number of flops is $2MNK$. In addition to our previously defined notations, we will use B to denote the batch size (the number of sequences processed in parallel), and s to denote the sequence length.

2.3.1 FLOPs by Operation

This section analyzes the FLOPs count by operation.

`word_embedding` $X * W_e : 2 * B * s * n_{vocab} * d_{model}$

`positional_embedding`: $2 * B * s * d_{model}$

We assume this is a fixed-representation of position such as the sinusoidal representation. As a result, we model the number of FLOPs as a constant factor times the shape of the positional embedding weights: $2 * B * s * d_{model}$

`attention_K`: $2 * B * s * d_{model} * d_k$ per head per transformer block (layer)

`attention_V`: $2 * B * s * d_{model} * d_v$ per head per layer

`attention_Q`: $2 * B * s * d_{model} * d_k$ per head per layer

`attention_QK`: $2 * B * s * d_{head} * s$ per head per layer

`attention_softmax`: $3 * B * s * s$ per head per layer We model this as a constant factor (3) times the size of the matrix to which softmax is being applied.

`attention_multV`: $2 * B * s * s * d_{head}$ per head per layer

`attention_out`: $2 * B * s * d_{model} * d_{model}$ per layer

`layernorm`: $5 * B * s * d_{model}$ per layernorm per layer
We model layernorm as performing 5 FLOPs on each element in the input

`mlp1`: $2 * B * s * d_{model} * d_{ff}$ per layer

`mlp2`: $2 * B * s * d_{ff} * d_{model}$ per layer

The total number of FLOPs of the embeddings (positional and word) is

$$\begin{aligned}
&\text{word\_embedding} + \text{positional\_embedding} \\
&= 2 * B * s * n_{vocab} * d_{model} + 2 * B * s * d_{model} \\
&= 2 * B * s * d_{model} * (n_{vocab} + 1)
\end{aligned}$$

The total number of FLOPs for the layernorm operations

$$\begin{aligned}
&\text{layernorm} * 2 * n_{layer} \\
&= 5 * B * s * d_{model} * 2 * n_{layer} \\
&= 10 * B * s * d_{model} * n_{layer}
\end{aligned}$$

The total number of FLOPs for attention:

$$\begin{aligned}
&((\text{attention\_K} + \text{attention\_V} + \text{attention\_Q} + \text{attention\_QK}+ \\
&\text{attention\_softmax} + \text{attention\_multV}) * n_{head} + \text{attention\_out}) * n_{layer} \\
&= ((2 * B * s * d_{model} * d_k + 2 * B * s * d_{model} * d_v + 2 * B * s * d_{model} * d_k + 2 * B * s * d_{head} * s+ \\
&2 * B * s * s + 2 * B * s * s * d_{head}) * n_{head} + 2 * B * s * d_{model} * d_{model}) * n_{layer} \\
&= (6 * B * s * d_{model} * d_{head} * n_{head} + 4 * B * s^2 * d_{model} + 2 * B * s^2 + 2 * B * s * d_{model}^2) * n_{layer} \\
&= (8 * B * s * d_{model}^2 + 4 * B * s^2 * d_{model} + 2 * B * s^2) * n_{layer}
\end{aligned}$$

The total number of FLOPs for the MLPs:

$$\begin{aligned}
&(\text{mlp1} + \text{mlp2}) * n_{layer} \\
&= (2 * B * s * d_{model} * d_{ff} + 2 * B * s * d_{ff} * d_{model}) * n_{layer} \\
&= (4 * B * s * d_{model} * 4 * d_{model}) * n_{layer} \\
&= 16 * B * s * d_{model}^2 * n_{layer}
\end{aligned}$$

Table 2.4: Asymptotic analysis of the scaling of the number of FLOPs by operation with respect to batch size, hidden dimension, and sequence length.

Operation	Batch Size B	Hidden Dimension d_{model}	Sequence Length s
word_embedding	$O(n)$	$O(n)$	$O(n)$
positional_embedding	$O(n)$	$O(n)$	$O(n)$
attention_K	$O(n)$	$O(n^2)$	$O(n)$
attention_V	$O(n)$	$O(n^2)$	$O(n)$
attention_Q	$O(n)$	$O(n^2)$	$O(n)$
attention_QK	$O(n)$	$O(n)$	$O(n^2)$
attention_softmax	$O(n)$	$O(1)$	$O(n^2)$
attention_multV	$O(n)$	$O(n)$	$O(n^2)$
attention_out	$O(n)$	$O(n^2)$	$O(n)$
layernorm	$O(n)$	$O(n)$	$O(n)$
mlp1	$O(n)$	$O(n^2)$	$O(n)$
mlp2	$O(n)$	$O(n^2)$	$O(n)$

Adding up the components, the total number of FLOPs is:

$$\text{embeddings} + \text{attention} + \text{mlp} + \text{layernorm}$$

$$= 2 * B * s * d_{model} * (n_{vocab} + 1) + (8 * B * s * d_{model}^2 + 4 * B * s^2 * d_{model} + 2 * B * s^2) * n_{layer}$$

$$+ 16 * B * s * d_{model}^2 * n_{layer} + 10 * B * s * d_{model} * n_{layer}$$

$$= 2 * B * s * d_{model} * (n_{vocab} + 1) + 28 * B * s * d_{model}^2 * n_{layer}$$

$$+ (4 * B * s^2 * d_{model} + 2 * B * s^2) * n_{layer} + 10 * B * s * d_{model} * n_{layer}$$

2.3.2 FLOPs Scaling

We analyzed how the number of FLOPs scale in each operation in response to scaling the batch size, the hidden dimension, and the sequence length. The analytical summary is shown in Table 2.4.

For the OPT models of different hidden dimension sizes d_{model} and different number of layers n_{layer}, with $B = 1$, and $s = 1$, the total number of FLOPs scales roughly linearly with the product of n_{layer} and d_{model}.

For the OPT class of models, the maximum sequence length is 1024. For models

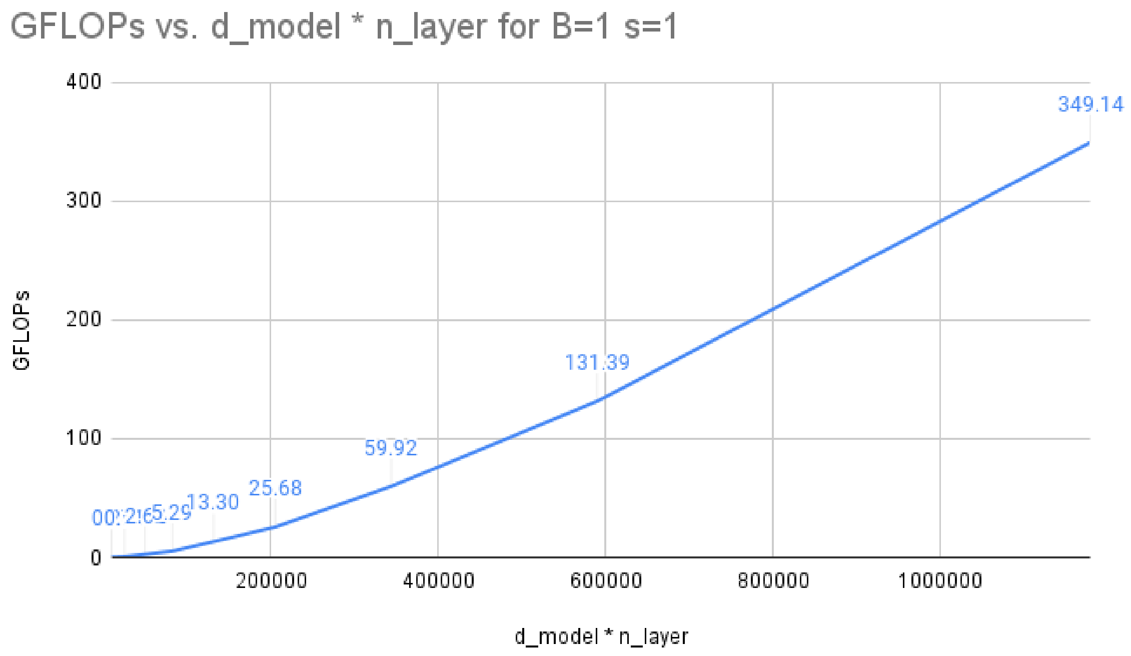

Figure 2-3: Number of FLOPs per token for the OPT family of models of different sizes.

below 350m parameters, the hidden dimension d_{model} is smaller than the maximum sequence length. For bigger models, d_{model} is greater than the maximum sequence length. Figures 2-4 and 2-5 examine the total number of FLOPs for both types of models as we vary the sequence length.

For both models, the total number of FLOPs scale roughly linearly with the sequence length, which suggests that the operations that scale quadratically with the sequence length do not account for a significant part of the total FLOPs, and the operations that scale linearly with the sequence length dominate.

2.3.3 FLOPs Distribution

Since the number of flops in all the individual operations scale linearly with the batch size, varying the batch size would not affect the distribution of FLOPs in a transformer inference workflow. Changing the hidden dimension and the sequence length, on the other hand, would change the distribution of FLOPs among the operations. Here we

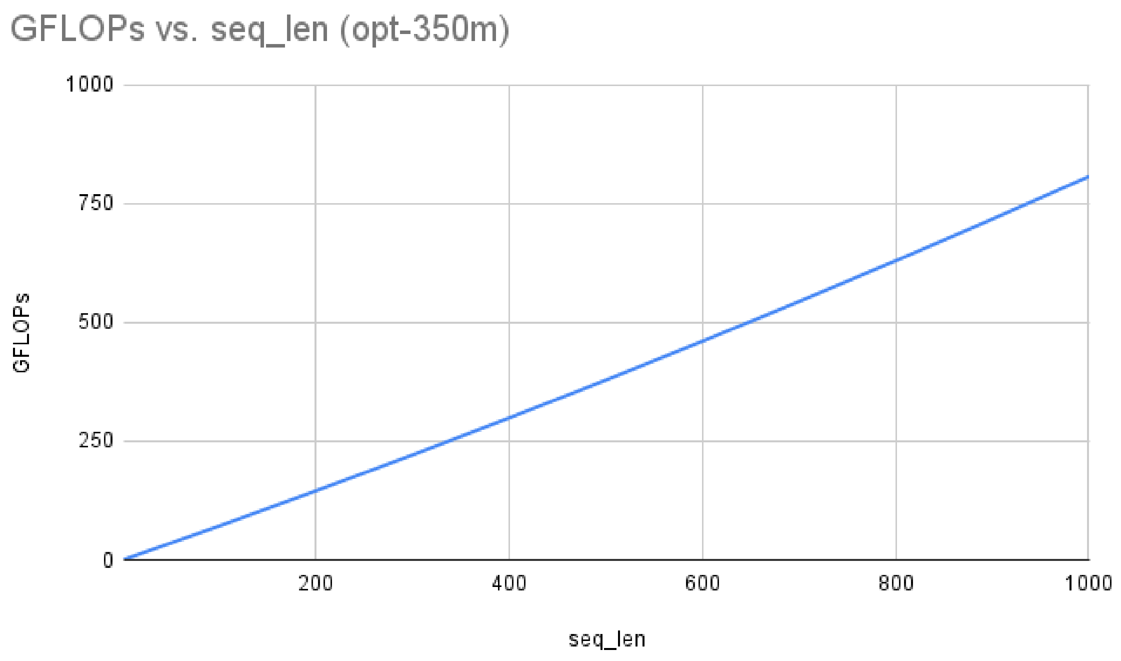

Figure 2-4: Number of FLOPs vs. sequence length for OPT-350M, up to the maximum supported sequence length (1024)

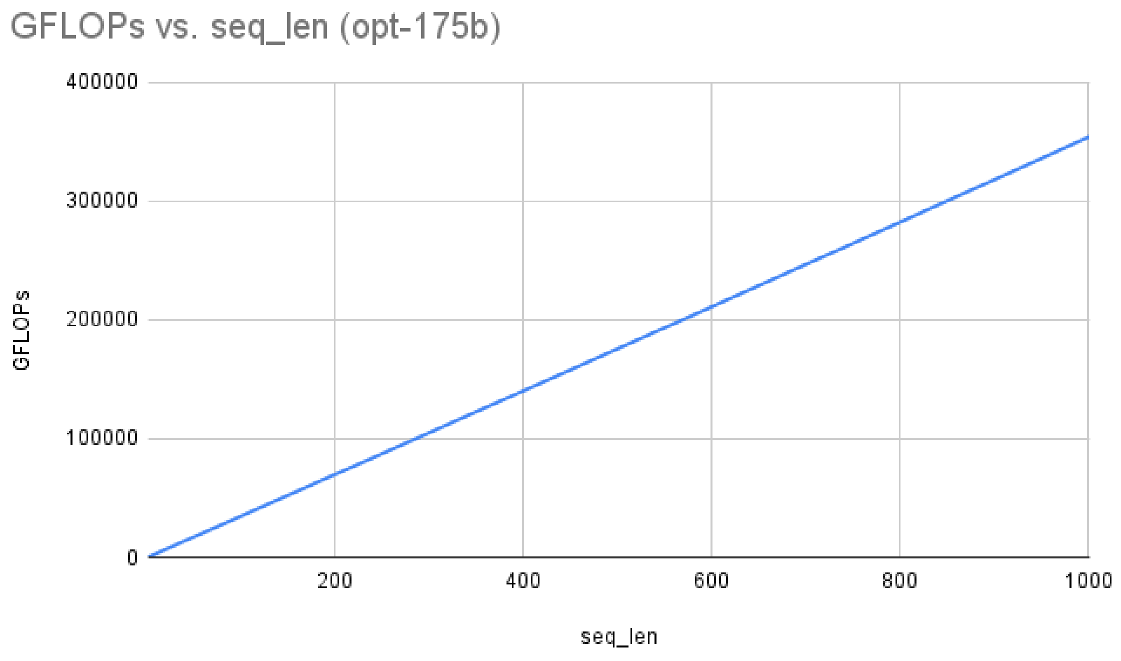

Figure 2-5: Number of FLOPs vs. sequence length for OPT-175B, up to the maximum supported sequence length (1024)

show the distribution of FLOPs for two models (opt-350m and opt-175b) with a short sequence length (10) and a long sequence length (1000) in Figure 2-6.

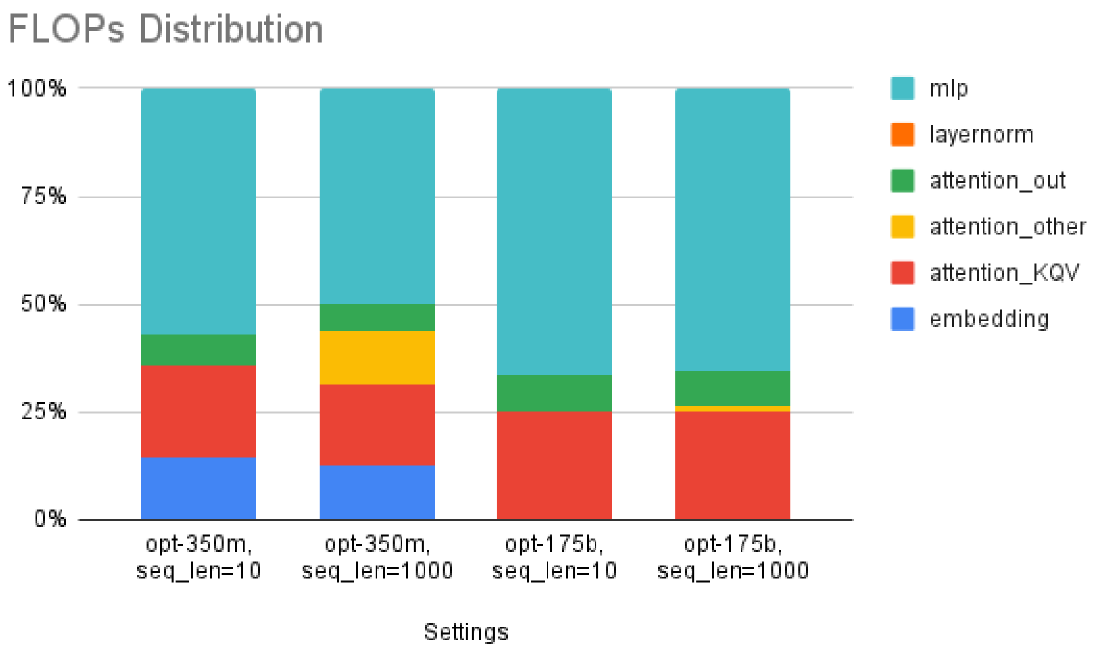

Figure 2-6: FLOPs distribution among operations with OPT-350M (small model) and OPT-175B (big model) for short (10) and long (1000) sequence lengths

We can make several observations here:

1. The MLP operation contributes to most of the FLOPs.
2. The KQV computations in the attention layers dominate most of the time, followed by `attention_out`.
3. The `attention_other` operations, which scale quadratically with sequence length, do not account for a significant number of FLOPs. In smaller models, they account for around 10% at most and are negligible in larger models.
4. The number of FLOPs spent on embeddings becomes a less significant fraction as the model gets larger.

The FLOPs analyses here only apply to the prefilling stage. The FLOPs analyses

Table 2.5: Comparing the number of TFLOPs for prefilling and generation for the same sequence length without using a KV-Cache. Using OPT 1.3b as an example.

seq len	prefilling TFLOPs	generation TFLOPs	ratio
201	0.4936853814	49.09933064	99.45469825
401	1.000867359	198.0358043	197.864185
601	1.523962297	449.992013	295.2776549
801	2.062970194	808.1505488	391.7412627

for the generation stage will be discussed in the next section which also discusses the usage of the KV cache.

2.4 Prefilling, Generation, and the KV Cache

2.4.1 Prefilling vs. Generation

A generative LLM inference task has two stages — the prefilling and the generation stage.

In the prefilling stage, the input prompt with a sequence length of s_{in} tokens is processed in one forward pass and the KV cache is computed. In the generation stage, the tokens are generated one at a time, since each of the generated tokens depends on the previous tokens. As a result, for an output sequence of sequence length s_{out}, s_{out} forward passes are required, and in each iteration, the sequence length increases by 1 from $s_{in} + 1$ to $s_{in} + s_{out}$. The KV cache is also updated in each iteration of the generation phase, with the stored K and V matrices for each head growing larger by $1 * d_k$, $1 * d_v$ respectively.

As a result of this autoregressive generation, for a given sequence length s, generating s output tokens takes a lot more FLOPs than processing s input tokens. Using the opt-1.3b model as an example, Table 2.5 shows the number of FLOPs required for generating or prefilling s tokens for varying s.

The number of FLOPs for generation grows quadratically as the sequence length increases, and it can be more than 2 orders of magnitude greater than prefilling for the same sequence length.

2.4.2 KV Cache and FLOPs

In the generation stage, we can save some of the recomputations on the entire sequence and only process the new token. The "KV cache" saves the past computations of K_i and V_i for each layer i. As a result, in each generation iteration, we only need to compute the new addition to K and V based on the newly added token t with shape $[B, 1, d_{model}]$ instead of computing the entire sequence with shape $[B, s+1, d_{model}]$.

Let s be the sequence length (including the new token) of the current iteration. The number of FLOPs based on only computing this one additional token is as follows:

`attention_K`

$$K_i = \text{concat}(K_i, t \cdot w_i^K)$$

Shapes: t: $[B, 1, d_{model}]$; W_i^K : $[d_{model}, d_k]$; K_i: $[B, s, d_k]$

Number of FLOPs: $2 * B * 1 * d_{model} * d_k$ per head per layer

`attention_V`

$$V_i = \text{concat}(V_i, t \cdot w_i^V)$$

Shapes: t: $[B, 1, d_{model}]$; W_i^V : $[d_{model}, d_v]$; V_i: $[B, s, d_v]$

Number of FLOPs: $2 * B * 1 * d_{model} * d_v$ per head per layer

`attention_Q`

$$t_i^Q = t * W_i^Q$$

Shapes: t: $[B, 1, d_{model}]$; W_i^Q : $[d_{model}, d_k]$; t_i^Q: $[B, 1, d_k]$

Number of FLOPs: $2 * B * 1 * d_{model} * d_k$ per head per layer `attention_QK`

$$t_i^Q * K^T$$

Shapes: t_i^Q: $[B, 1, d_{head}]$; K^T: $[B, d_{head}, s]$

Number of FLOPs: $2 * B * 1 * d_{head} * s$ per head per layer

`attention_softmax`

$$\text{softmax}(\frac{t^Q K^T}{\sqrt{d_k}})$$

Shapes: $t^Q K^T$: $[B, 1, s]$

Number of FLOPs: $3 * B * 1 * s$ FLOPs per head per layer

`attention_multV`

$$\text{softmax}(\frac{t^Q K^T}{\sqrt{d_k}})V$$

Shapes: $(t^Q * K^T)$: $[B, 1, s]$; V: $[B, s, d_{head}]$

Number of FLOPs: $2 * B * 1 * s * d_{head}$ per head per layer

`attention_out`

$$\text{concat}(\text{head\_1}, ..., \text{head\_h})W^O$$

Shapes: $U = \text{concat}(\text{head\_1}, ..., \text{head\_h})$: $[B, 1, d_{model}]$; W^O: $[d_{model}, d_{model}]$

Number of FLOPs: $2 * B * 1 * d_{model} * d_{model}$ per layer

Table 2.6: Comparing the number of TFLOPs during generation with a KV cache vs. without a KV cache, using OPT-1.3b as an example.

Seq Len	Gen TFLOPs with KV Cache	Gen TFLOPs without KV Cache	Percentage
201	0.492121728	49.09933064	1.002298242
401	1.002030336	198.0358043	0.5059844302
601	1.529725824	449.992013	0.3399451056
801	2.075208192	808.1505488	0.2567848522

`mlp1`

$$tW_1 + b_1$$

Shapes: t: $[B, 1, d_{model}]$; W_1: $[d_{model}, d_{ff}]$

Number of FLOPS: $2 * B * 1 * d_{model} * d_{ff}$ per layer

`mlp2`

$$tW_2 + b_2$$

Shapes: t: $[B, 1, d_{model}]$; W_2: $[d_{ff}, d_{model}]$

Number of FLOPS: $2 * B * 1 * d_{ff} * d_{model}$ per layer

The layernorm operations will have the same number of FLOPs as without using KV cache. And the final output will be concatenated to X for the next iteration.

When using the KV cache, the number of FLOPs in generation is less than 1% of the number of FLOPs when not using the KV cache, as shown in Table 2.6

Furthermore, with the KV cache, the number of FLOPs for generation becomes roughly the same as the number of FLOPs for prefilling for a given sequence length s. However, in practice, we will see that generation is still a lot slower than prefilling for the same sequence length even if we are using a KV cache, as shown in Table 2.7 (the latency numbers in this table are obtained from running the opt-1.3b model with

Table 2.7: Prefilling vs. Generation Latency with OPT-1.3b using Huggingface Transformer

Batch Size	Sequence Length	Prefilling Latency (ms)	Generation Latency (ms)
1	128	29.40297127	2754.386663
1	256	30.16424179	5513.639212
1	512	33.46157074	11039.16216
1	1000	74.98526573	21547.35923
16	16	4.95542757	385.1943016
16	128	40.01406281	3037.932634
16	256	80.87530793	6082.014799
16	512	165.139345	12195.14251
16	1000	335.1545119	24587.50057

the HuggingFace Transformer library). This is due to the differences in arithmetic intensity and will be investigated in a later section.

2.4.3 KV Cache Tradeoffs

Using the KV cache is essentially a tradeoff between memory and compute. Using the KV cache avoids recomputing the K and V matrices at every token, but it requires caching these matrices in memory. The size of the KV cache is the total size of all the K and V matrices at maximum sequence length $s_{in} + s_{out}$ in all layers. Using the KV cache is a tradeoff between memory and compute in which the amount of extra memory used can be computed as

$$\begin{aligned}
&(\text{size of K} + \text{size of V}) * n_{head} * n_{layer} \\
&= (B * (s_{in} + s_{out}) * d_k + B * (s_{in} + s_{out}) * d_v) * n_{head} * n_{layer} \\
&= B * (s_{in} + s_{out}) * d_{model} * n_{layer}
\end{aligned}$$

The extra memory consumption is linear in d_{model} and s, while compute savings are quadratic in d_{model} and s. This makes using a KV cache almost always worthwhile in generation.

2.5 Estimating Memory Usage

2.5.1 Memory Components

The memory footprint of transformer inference comes from mainly three components: parameters, the kv cache, and intermediate activations.

For each of the components, the memory usage is the number of elements multiplied by bytes per number, which is 2 for FP16 and 4 for FP 32.

In previous sections, we calculated the number of parameters as well the size of the KV cache. Recall that the number of parameters is

$$12 * d_{model}^2 * n_{layer} + (n_{vocab} + n_{ctx}) * d_{model} + 4 * d_{model} * n_{layer}$$

and the number of elements in the KV cache is

$$2 * B * d_{model}^2 * n_{layer} * (2 * s_{in} + s_{out} + 1) * s_{out}$$

To estimate the memory usage of intermediate activations, we can assume that the GPU efficiently allocates and deallocates intermediate tensors in each of the operations. To estimate the memory usage of each of the individual operations, we will sum up the sizes of the inputs and outputs for that operation (excluding parameters), and then pick the maximum (peak) across all operations. In addition, to account for the residual stream in the transformer blocks, we need to add the size of the residual stream to the estimate of the intermediate activation, which is $B * s * d_{model}$.

2.5.2 Activation by Operation

The sizes of the intermediate activations by operations are described in this section.

`word_embedding`: $B * s * (n_{vocab} + d_{model})$

Inputs: X: $[B, s, n_{vocab}]$

Outputs: X: $[B, s, d_{model}]$

`positional_embedding`: $2 * B * s * d_{model}$

Inputs: X: $[B, s, d_{model}]$

Outputs: X: $[B, s, d_{model}]$

`attention_K`: $B * s * (d_{model} + d_k)$ per head per layer

Inputs: X: $[B, s, d_{model}]$

Outputs: K_i: $[B, s, d_k]$

`attention_V`: $B * s * (d_{model} + d_v)$ per head per layer

Inputs: X: $[B, s, d_{model}]$

Outputs: V_i: $[B, s, d_v]$

`attention_Q`: $B * s * (d_{model} + d_k)$ per head per layer

Inputs: X: $[B, s, d_{model}]$

Outputs: Q_i: $[B, s, d_k]$

`attention_QK`: $2 * B * s * d_{head} + B * s * s$ per head per layer

Inputs: Q: $[B, s, d_{head}]$, K^T: $[B, d_{head}, s]$

Outputs: $Q * K^T$: $[B, s, s]$

`attention_softmax`: $2 * B * s * s$ per head per layer

Inputs: QK^T: $[B, s, s]$

Outputs: $[B, s, s]$

`attention_multV`: $B * s * s + 2 * B * s * d_{head}$ per head per layer

Inputs: $\text{softmax}(\frac{QK^T}{\sqrt{d_k}})$: $[B, s, s]$; V: $[B, s, d_{head}]$

Outputs: $[B, s, d_{head}]$

`attention_out` $2 * B * s * d_{model}$ per layer

Inputs: $U = \text{concat(head\_1, ..., head\_h)}$: $[B, s, d_{model}]$

Outputs: $[B, s, d_{model}]$

`layernorm`: $2 * B * s * d_{model}$ per layernorm per layer

Inputs: X: $[B, s, d_{model}]$

Outputs: X: $[B, s, d_{model}]$

`mlp1`: $B * s * (d_{model} + d_{ff})$ per layer

Inputs: X: $[B, s, d_{model}]$

Outputs: X: $[B, s, d_{ff}]$

`mlp2`: $B * s * (d_{ff} + d_{model})$ per layer

Inputs: X: $[B, s, d_{ff}]$

Outputs: X: $[B, s, d_{model}]$

2.5.3 Maximum Batch Size

The maximum batch size is limited by the GPU memory, which has to store the weights, the kv cache, and intermediate activations. Let's consider the single GPU case for now where all the weights reside on the same GPU.

To calculate the maximum batch size that fits on the GPU, we will first need to account for the memory used for storing parameters, and that leaves us with roughly the available memory for the KV cache and intermediate activations. (We will assume a very efficient GPU memory utilization here since we are doing speed of light estimations.) Then we will get the memory footprint of an individual input, divide the remaining GPU memory by that, and get the maximum number of batches it can fit.

Let $A(\text{model\_config}, \text{seq\_len})$ be the maximum activation from one input sequence, then the maximum batch number (number of input sequences) is

$$\text{max batches} = \frac{\text{GPU memory} - \text{parameter size}}{A(\text{model\_config}, \text{seq\_len})}$$

For the model opt-1.3b with FP16 precision, the weights take up around 2.6G of storage. Assume a maximum context length of 1024, the maximum activation of each input is around 0.10 GB, and the kv cache takes up 0.11 GB. In total, each input will result in 0.21G of memory footprint. On a V100 GPU with 32GB of memory, the maximum number of batches is $(32 - 2.6)/0.21 \approx 140$. On an A100 GPU with 80GB of memory, the maximum number of batches is approximately 360. If we restrict the maximum context length to 100, the maximum number of batches on a V100 is 1418, and on an A100 is 3737.

2.6 Speed of Light Latency

The "Speed of Light" Latency refers to the lowest possible latency assuming maximum hardware utilization. In practice, it will be unlikely for actual workloads to reach the speed of light (achieve full utilization of the hardware), but it will be a lower bound for analyzing latency.

2.6.1 FLOPs Latency

One simple way of estimating latency is to get the total number of FLOPs and divide by the maximum FLOPs per second supported by the GPU, which is a number that is specified in the GPU datasheets.

$$\text{FLOPs latency} = \frac{\text{total FLOPs}}{\text{maximum FLOPs / s}}$$

This approximation will be reasonable for compute-bound operations, but it will lead to non-negligible underestimations of the latency for memory-bound operations, which we will discuss in the next section.

2.6.2 Arithmetic Intensity

The arithmetic intensity is defined as the ratio of the number of floating point operations (FLOPs) to memory accesses (bytes).

$$\text{arithmetic intensity} = \frac{\text{FLOPs}}{\text{memory accesses}}$$

For a specific GPU, the "critical ratio" is the ratio of the maximum number of FLOPs per second over the memory bandwidth. For a V100 GPU, using FP16 tensor cores, this ratio is

$$\begin{aligned}\text{critical ratio} &= \frac{\text{maximum FP16 tensor core FLOPs per second}}{\text{Memory bandwidth}} \\ &= \frac{125 * 10^{12}}{300 * 10^{9}} \\ &= \frac{5}{12} * 10^{3}\end{aligned}$$

Operations with an arithmetic intensity below the critical ratio are memory-bound, and operations with an arithmetic intensity above the critical ratio are compute-bound.

2.6.3 Roofline Latency

For compute-bound operations, we will estimate the latency by using the FLOPs latency as defined above. For memory-bound operations, we will estimate the latency by using the maximum of the FLOPs latency and the memory latency. The memory latency is defined as the time it takes to access the data from memory or write the data to memory, which is the activation memory of the operation divided by the memory bandwidth of the hardware.

$$\text{memory latency} = \frac{\text{total activation memory}}{\text{memory bandwidth}}$$

$$\text{estimated latency} = \max(\text{FLOPs latency}, \text{memory latency})$$

Chapter 3

Inference with a Single GPU

3.1 Empirical Hardware Utilization

In the previous chapter, we developed an analytical model for estimating transformer inference performance. We use latency as our main metric here. In this chapter, we validate our analytical model and compare that to empirical measurements. We measure the latency of transformer inference on a single GPU with the implementations in two different libraries — HuggingFace Transformers and Nvidia FasterTransformer.

We also use the measurements to estimate the hardware utilization of the GPU. We use the term "FLOPs utilization" to refer to the ratio of actual FLOPs achieved to the maximum FLOPs supported by the hardware, which can be estimated by the ratio of the actual latency to the FLOPs latency.

$$\text{FLOPs utilization} = \frac{\text{actual FLOPs}}{\text{maximum FLOPs}} \approx \frac{\text{FLOPs latency}}{\text{actual latency}}$$

In addition to FLOPs utilization, we also define the "analytical utilization" as the ratio of the latency given by the analytical model to the actual latency.

$$\text{Analytical utilization} \approx \frac{\text{Analytical latency}}{\text{Actual latency}}$$

3.1.1 Nvidia FasterTransformer Hardware Utilization

For analyzing the hardware utilization of an existing implementation, we will use Nvidia FasterTransformer, since it is one of the most well-optimized and publicly available implementations of transformers. We will look at both the FLOPs utilization and the analytical utilization. The analytical utilization will tell us how efficient the implementation is at maximizing hardware utilization, and the FLOPs utilization, in combination with the analytical utilization, will tell us how efficient the network architecture is at maximizing hardware utilization.

Prefilling Stage

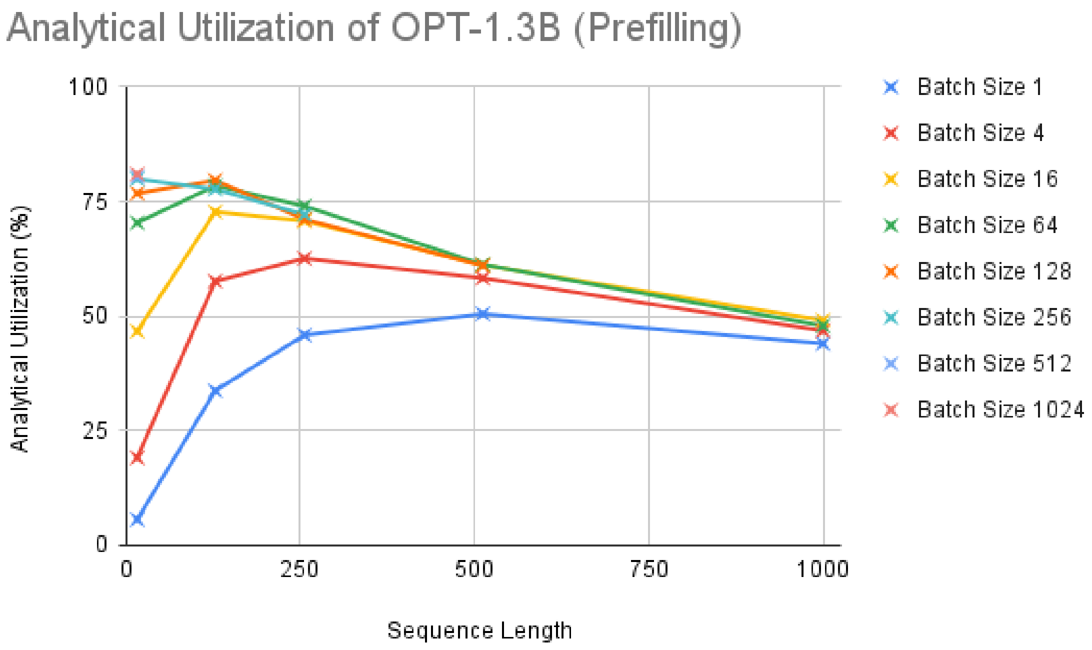

Figure 3-1: Analytical utilization (%) of OPT-1.3B during the prefilling stage for various batch sizes and input sequence lengths

Figure 3-1 shows the analytical utilization (%) of OPT-1.3B during the prefilling stage for various batch sizes and input sequence lengths, and Figure 3-2 shows the FLOPs utilization. There are several observations we can make here

1. The shapes of the analytical utilization and FLOPS utilization curves look very

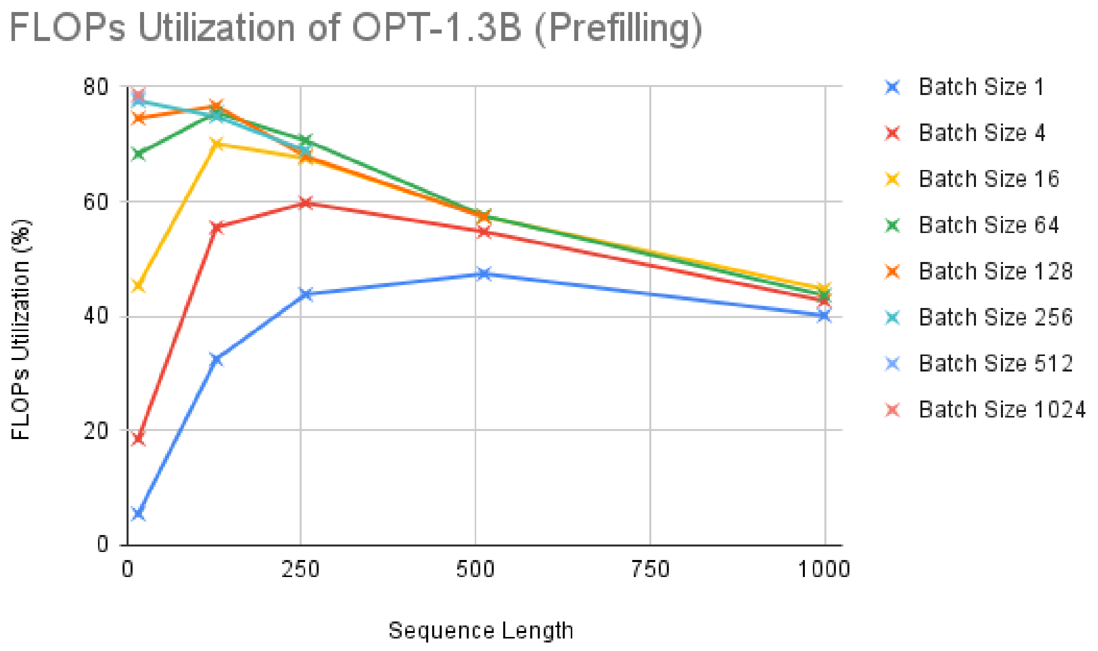

Figure 3-2: FLOPs utilization (%) of OPT-1.3B during the prefilling stage for various batch sizes and input sequence lengths

similar, and for each (batch size, sequence length) data point, the FLOPs utilization is only lower than the analytical utilization by a few percent. This means that the prefilling stage workloads generally have high arithmetic intensity.

2. As the sequence length increases, the analytical utilization for all batch sizes converges to around 46%, and the FLOPs utilization for all batch sizes converges to around 42%.

3. For almost all batch sizes (except batch size 1), both the FLOPs and analytical utilization peak at a sequence length of around 128 - 256, which is about 10% - 20% of the maximum context length. The utilizations decrease as the sequence lengths grow longer than that range.

4. The highest FLOPs utilization we observed for prefilling is 78% for batch size 1024 and input sequence length 16. The highest analytical utilization we observed for prefilling is 81% also for batch size 1024 and input sequence length

16. This indicates that the implementation of Nvidia FasterTransformer is reasonably efficient at utilizing the hardware.

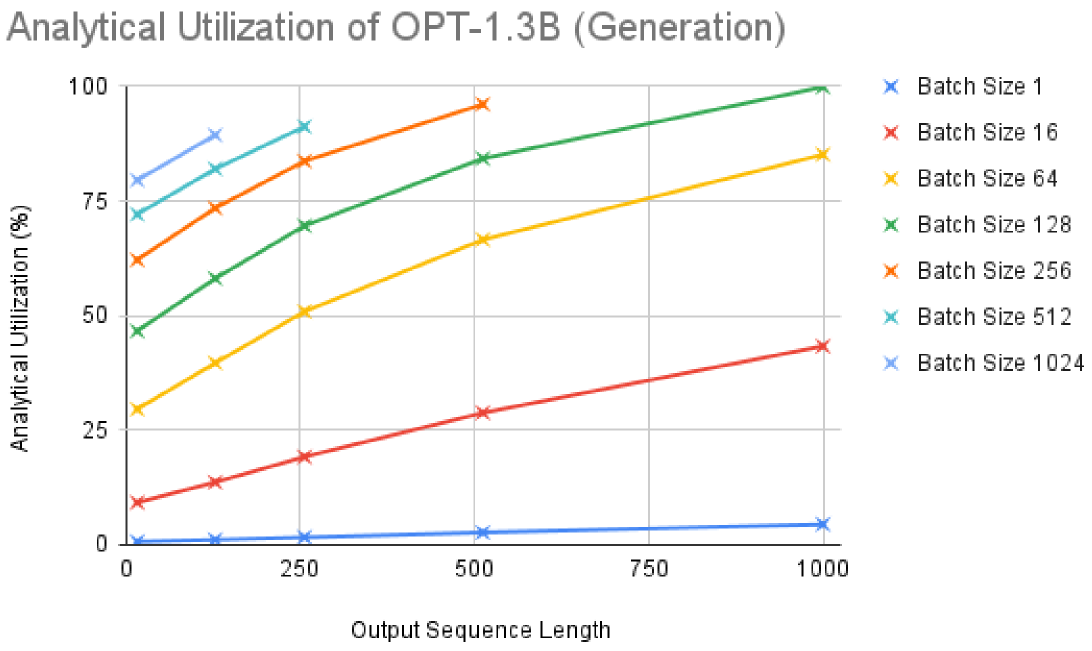

Figure 3-3: Analytical utilization (%) of OPT-1.3B during the generation stage for various batch sizes and input sequence lengths

Generation Stage

Figure 3-3 shows the analytical utilization (%) of OPT-1.3B during the generation stage for various batch sizes and input sequence lengths, and Figure 3-4 shows the FLOPs utilization. Some observations we can make here are

1. Unlike the prefilling stage, the shapes of the FLOPs and analytical utilization curves are very different. The FLOPs utilization is quite low while the analytical utilization is reasonably high. This is consistent with our analysis that the operations in the generation stage are mostly memory bound (compute bound operations would show similar FLOPs and analytical utilization). From this disparity in FLOPs vs analytical utilization, we can also conclude that the implementation of Nvidia FasterTransformer is also reasonably efficient at utilizing

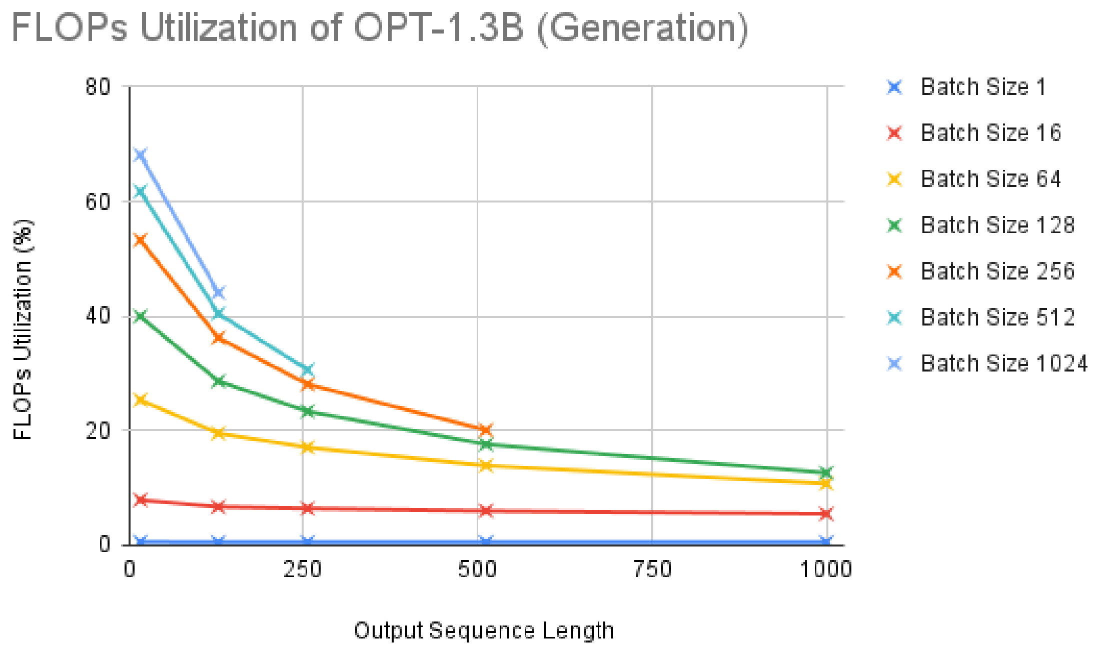

Figure 3-4: FLOPs utilization (%) of OPT-1.3B during the generation stage for various batch sizes and input sequence lengths

the hardware, but we are limited by the memory-bound operations to achieve a high FLOPs utilization.

2. The FLOPs utilization decreases for all batch sizes as the sequence length increases. This is consistent with our analysis that the autoregressive generation steps are memory bound. With more generation steps, the FLOPs utilization decreases as there are more memory bound operations. Overall, the generation FLOPs utilization is quite low.

3. The analytical utilization becomes higher as the sequence length grows longer for all batch sizes.

3.1.2 Comparing the Performance of Huggingface Transformers and Nvidia FasterTransformer

Table 3.1 compares the latency between Huggingface Transformers (HFT) and Nvidia FasterTransformer (FT) for the prefilling stage for different batch sizes and sequence lengths. There are several observations we can make

1. For most of the different (batch size, input sequence length) configurations, the ratio of the HFT latency to the FT latency is about 1.5, which indicates that Huggingface Transformer is 1.5x slower than Nvidia FasterTransformer for most of the prefilling workloads.

2. Workloads with small batch sizes and short sequence lengths have a higher ratio, since they have fewer compute-bound matrix multiplication operations. The latency of the matrix multiplication operations should not be too different for Huggingface Transformers and Nvidia Transformer, since they should both be using optimized CUDA kernels for matrix multiplication, which is a well-studied problem.

3. Huggingface Transformers runs out of memory earlier than Nvidia FasterTransformer, as indicated by "N/A"s in Table 3.1. This indicates that FT is more efficient at memory usage than HFT.

Table 3.2 compares the latency between Huggingface Transformers (HFT) and Nvidia FasterTransformer (FT) for the generation stage for different batch sizes and sequence lengths. There are several observations we can make

1. The ratio HFT / FT Latency ratio decreases as we increase the batch size. This is consistent with our expectation that operations with higher arithmetic intensity will show a smaller gap between the latency of the different implementations.

2. With the same batch size, the HFT / FT Latency ratio is similar.

Table 3.1: Huggingface Transformers vs Nvidia FasterTransformer prefilling latency for various batch size and input sequence length configurations

Batch Size	Input Seq Len	HFT Latency	FT Latency	HFT / FT Latency
1	16	28.05018425	6.12	4.58336344
1	128	29.40297127	8.34	3.525536123
1	256	30.16424179	12.5	2.413139343
1	512	33.46157074	23.54	1.421477092
1	1000	74.98526573	56.26	1.332834442
4	16	28.87535095	7.24	3.988308143
4	128	28.73539925	19.54	1.470593615
4	256	51.5396595	36.65	1.406266289
4	512	112.9214764	81.54	1.384859901
4	1000	N/A	211.55	#VALUE!
16	16	29.0248394	11.86	2.447288314
16	128	91.42208099	61.91	1.47669328
16	256	190.1779175	129.65	1.466856286
16	512	N/A	310.97	#VALUE!
16	1000	N/A	806.99	#VALUE!
64	16	45.50933838	31.48	1.44565878
64	128	338.6039734	229.74	1.473857288
64	256	N/A	495.84	#VALUE!
64	512	N/A	1241.87	#VALUE!
64	1000	N/A	3308.78	#VALUE!
128	16	86.56692505	57.71	1.500033357
128	128	N/A	452.64	#VALUE!
128	256	N/A	1032.06	#VALUE!
128	512	N/A	2493.15	#VALUE!
128	1000	N/A	N/A	#VALUE!
256	16	167.2084332	110.89	1.507876573
256	128	N/A	927.71	#VALUE!
256	256	N/A	2033.72	#VALUE!
256	512	N/A	N/A	#VALUE!
256	1000	N/A	N/A	#VALUE!
512	16	320.7089901	220.32	1.455650827
512	128	N/A	N/A	#VALUE!
512	256	N/A	N/A	#VALUE!
512	512	N/A	N/A	#VALUE!
512	1000	N/A	N/A	#VALUE!
1024	16	635.9052658	437.48	1.453564199
1024	128	N/A	N/A	#VALUE!
1024	256	N/A	N/A	#VALUE!
1024	512	N/A	N/A	#VALUE!
1024	1000	N/A	N/A	#VALUE!

3. For the different (batch size, input sequence length) configurations, the ratio of the HFT latency to the FT latency does not stay fairly constant as in the prefilling case. It is around 3 for most of the workloads, indicating that HFT is less efficient than FT at the generation stage.

4. Compared to prefilling, both HFT and FT are able to support larger batch size and sequence length combinations. We think this is because in the attention operation, when using a KV cache during generation, we save on some of the matrices with sequence length as dimensions, since some of them will become 1 with the KV cache.

Table 3.2: Huggingface Transformers vs Nvidia FasterTransformer generation latency for various batch size and output sequence length configurations

Batch Size	Input Seq Len	HFT Latency	FT Latency	HFT / FT Latency
1	16	348.7701416	60.97	5.720356595
1	128	2754.386663	465.65	5.915143699
1	256	5513.639212	937.79	5.879396466
1	512	11039.16216	1888.29	5.846115882
1	1000	21547.35923	3902.62	5.521254754
16	16	385.1943016	75.38	5.110033187
16	128	3037.932634	613.74	4.949869056
16	256	6082.014799	1288.86	4.718910354
16	512	12195.14251	2838.09	4.296954116
16	1000	24587.50057	6455.6	3.808708807
64	16	400.21348	94.14	4.251258551
64	128	3163.727045	845.59	3.741443306
64	256	6729.804516	1945.33	3.459466782
64	512	17832.17216	4911.49	3.630705174
64	1000	54524.64676	13149.42	4.14654386
128	16	401.484251	119.53	3.358857618
128	128	3571.768045	1154.57	3.093591593
128	256	9511.126995	2844.58	3.343596241
128	512	29716.56609	7758.76	3.830066413
128	1000	N/A	22415.87	#VALUE!
256	16	408.7283611	179.35	2.278942632
256	128	5267.625093	1826.36	2.884220577
256	256	15967.69786	4730.66	3.375363661
256	512	N/A	13606.83	#VALUE!
256	1000	N/A	N/A	#VALUE!
512	16	629.3461323	309.31	2.034677612
512	128	9450.854301	3272.29	2.888146925
512	256	N/A	8681.98	#VALUE!
512	512	N/A	N/A	#VALUE!
512	1000	N/A	N/A	#VALUE!
1024	16	1126.393557	560.8	2.008547711
1024	128	N/A	6003.39	#VALUE!
1024	256	N/A	N/A	#VALUE!
1024	512	N/A	N/A	#VALUE!
1024	1000	N/A	N/A	#VALUE!

Chapter 4

Inference with multiple GPUs

4.1 Parallelism Strategies

Most machine learning models in the past fit in the memory of a single GPU without issues; however, as the models become more complex and larger, a single GPU is no longer sufficient, and computation must move into the distributed space. The opportunity to parallelize work across multiple machines enables performance boosts, and, parallelization, combined with intelligent partitioning, allows us to use large models that would otherwise be impossible to fit on a single GPU. Some parallelization strategies are described below.

4.1.1 Data Parallelism

Data parallelism is the simplest parallelism technique with minimal modification to the serial code. It can be used when the entire model fits in a single GPU. To adapt the model to a multi-GPU setting, the same model is replicated across all the GPUs, and each replica receives and processes different slices of the dataset. For inference, the model parameters do not need to be synchronized and updated, and different GPUs can be used to generate results on different batches of data which are then aggregated to produce the final results. For transformer inference with input sizes $[B, s, d_{model}]$, this is splitting along the batch dimension B.

4.1.2 Model Parallelism

Model parallelism is a strategy where the model is split across different GPUs, and it is usually applicable in the case where the model cannot fit in the memory of a single GPU, since each device only needs to store and compute a fraction of the model. The term "model parallelism" can be an all-encompassing term describing different strategies of partitioning a large model, and ambiguity exists regarding what specific partitioning strategy is described by the term. In the sub-section below, we will describe the different parallelism strategies that the term "model parallelism" can refer to, focusing specifically on transformer-based models.

4.1.3 Tensor Parallelism

In tensor parallelism, individual tensors are sharded across different GPUs, and each GPU only processes a slice of a tensor. The results from the individual GPUs are only aggregated into a full tensor for operations that need the entire tensor. In transformers, tensor parallelism is used in the multi-headed attention and the MLP to split larger matrices into smaller ones across different accelerators. We will follow the tensor parallelism scheme described by Nvidia Megatron [20] and implemented in Nvidia FasterTransformer [14].

MLP

We ignore the bias of the MLP and use the simplified definition of the operation

$$f(X) = (X * A) * B$$

where X has shape $[B, s, d_{model}]$, A has shape $[d_{model}, d_{ff}]$, and B has shape $[d_{ff}, d_{model}]$. A is partitioned along columns such that on each device i, A has shape $[d_{model}, \frac{d_{ff}}{d_{model}}]$, and $X * A$ has shape $[B, s, \frac{d_{ff}}{d_{model}}]$. B is partitioned along rows, such that on each device i, B has shape $[\frac{d_{ff}}{d_{model}}, d_{model}]$. The output will have shape $[B, s, d_{model}]$.

Multi-Headed Attention (MHA)

The MHA has multiple attention heads, and each attention head has the key, value, and query matrices. The K, Q, V matrices have size $[B, s, d_{head}]$, where $d_{head} = \frac{d_{model}}{n_{head}}$. The weight matrices W_K, W_Q, W_V are for each head, with shape $[d_{model}, d_{head}]$. The heads are evenly split among the devices, such that each device has $\frac{n_{head}}{TP}$ heads.

The W_o matrix has shape $[d_{model}, d_{model}]$, and it is split along the row, such that on each device the shape of the shard is $[\frac{d_{model}}{TP}, d_{model}]$.

4.1.4 Pipeline Parallelism

The model is split up by layers so that one or several layers of the model reside on a single GPU. Due to the sequential nature of DNNs, this strategy, if implemented naively with only partitioning the model layers, would result in only one device actively computing at a time, resulting in underutilization of the hardware. To more efficiently use multiple GPUs, the incoming data batches (mini-batches) are chunked into micro-batches so that different GPUs can work on separate micro-batches at the same time. Parallelism is achieved by pipelining the execution of micro-batches.

4.1.5 More on Parallelism

The motivation for model parallelism currently is to fit large models into the GPU memory; however, we believe that there is also the potential to explore model parallelism in the case of a small model that fits in a single GPU in order to optimize for latency, especially for workflows that are heavily-compute bound. More generally, we want to also answer the question that if a model can fit in N GPUs, do we ever want to use $k * N$ GPUs for inference?

Weight Stationary

We focus on weight-stationary parallelization strategies here where the weights stay on the GPUs and the activations get sent around. **Activation Stationary** refers to the situation where activations stay on the same GPU and the weights get sent around.

4.2 Extending the Analytical Model to Multiple GPUs

We built an analytical model for the latency of multi-GPU inference using the strategies we described in the previous section.

4.2.1 Data Parallelism

Data parallelism can be easily modeled as replicating the same model across different devices with no additional communication overhead. If we originally have N GPUs, and running inference with batch size B has latency T, then with $k * N$ GPUs, we will be able to run inference with a total batch size of $k * B$ with the same latency T.

4.2.2 Tensor Parallelism

As we recall from earlier, tensor parallelism can be applied to the MLP and Multi-head attention layers.

MLP

For an inference job with weights of sizes $[d_{model}, d_{ff}]$ and $[d_{ff}, d_{model}]$, the weights can be parallelized across TP GPUs, such that each device will work weights of sizes $[d_{model}, \frac{d_{ff}}{TP}]$ and $[\frac{d_{ff}}{TP}, d_{model}]$.

As a result, the activation between the two layers of the MLP becomes $[B, s, \frac{d_{ff}}{TP}]$, and across the two layers, the number of FLOPs reduces by a factor of TP.

The communication will happen at the end of MLP block with an all-reduce operation. In a ring all-reduce operation with N devices, the communication volume is tensor size $* 2 * (TP - 1) = 2 * B * s * d_{model} * (TP - 1)$ for each device in layer. Since it is possible to have all GPUs communicate with each other at a given time as long as the NVLink memory bandwidth is not exceeded, we can assume that all the

GPUs are communicating at the same time. As a result, the communication latency resulting from MLP tensor parallelism in each layer is

$$\begin{aligned}&\text{MLP TP Comm Latency Per Layer}\\&= (2 * B * s * d_{model} * (TP - 1)) * \frac{1}{\text{NVLink Memory Bandwidth}} + \text{NVLink Latency}\\&= \frac{2 * B * s * d_{model} * (TP - 1)}{\text{NVLink Memory Bandwidth}} + \text{NVLink Latency}\end{aligned}$$

where NVLink Latency is the initial latency to communicate a message.

To get the communication cost across all layers, multiply that by the number of layers in the transformer.

MHA

Tensor parallelism for MHA is similar to MLP. Each device will get $\frac{n_{head}}{TP}$ heads, which means on each device, the sizes of the K, Q, V matrices will be reduced by a factor of TP. The communication volume is the same as the MLP, since the sizes of the communicated tensors are the same, and in each transformer block there is one MHA and one MLP component.

In total, the tensor parallelism contributes the following communication cost

$$\begin{aligned}\text{TP Comm Latency} &= (\text{MLP TP Comm Latency} + \text{MHA TP Comm Latency}) * n_{layer}\\&= (\frac{4 * B * s * d_{model} * (TP - 1)}{TP * \text{NVLink Memory Bandwidth}} + 2 * \text{NVLink Latency}) * n_{layer}\end{aligned}$$

4.2.3 Pipeline Parallelism

In pipeline parallelism communication only happens between the pipeline stages, and for each stage the communication volume is $B * s * d_{model}$. If we just process one batch, only one GPU will be active at a time, and there will be no benefits to parallelism — only extra communication costs.

In practice, to solve the problem of pipeline bubbles and idle GPUs, a single batch is sliced into mini-baches of size $[b, s, d_{model}]$ where $b *$ number of batches $= B$.

We will define a timestep as everything moving forward one stage in the pipeline. The total number of time steps needed to complete the entire batch B is $\frac{B}{b} + PP - 1$. In each timestep, the communication volume for all GPUs can happen at the same time, so the communication latency at each time step is

$$\text{PP Comm Latency} = \frac{b * s * d_{model}}{\text{NVLink Memory Bandwidth}} + \text{NVLink Latency}$$

If we define timesteps this way, the compute latency at each timestep is the overall inference latency divided by the number of pipeline stages.

$$\text{Compute latency per pipeline stage} = \frac{\text{Full Forward Pass Latency}}{PP}$$

4.2.4 Combining Pipeline and Tensor Parallelism

When we have both pipeline and tensor parallelism, the inference workflow can be described by the following pseudocode:

```
total steps = global batch size / local batch size + PP - 1
for step in total steps:
    for layer in total layers in PP stage:
        Compute
        TP communication for MHA
        Compute
        TP communication for MLP
    PP communication
```

We will now proceed to calculate the total latency when using both pipeline and tensor parallelism in combination.

Compute Latency

In each step, the total compute latency is the latency to perform a tensor-paralleled computation ($\frac{1}{TP}$ of the weights on a single GPU) with $\frac{n_{layer}}{PP}$ transformer blocks and on an input size of $[b, s, d_{model}]$. We can obtain this number by first invoking the latency calculations we defined in the single-GPU case: let $T_{forward}$ be the latency for a full forward pass on an input size $[b, s, d_{model}]$ with the weights divided appropriately, and then the compute latency for each step is $T_{step} = \frac{T_{forward}}{PP}$. The total compute latency is then $T_{step} *$ total steps.

Communications Latency

In each layer in each step, the tensor parallelism communication latency is $(\frac{4*b*s*d_{model}*(TP-1)}{\text{NVLink Memory Bandwidth}} + 2*\text{NVLink Latency})$. Multiply that by the number of layers in each stage ($\frac{n_{layer}}{PP}$) and then by the total number of steps to get the total communication latency contributed by tensor parallelism.

The pipeline communication latency at each step is $\frac{b*s*d_{model}}{\text{NVLink Memory Bandwidth}}+\text{NVLink Latency}$, multiply that by the total number of steps to get the total communication latency contributed by pipeline parallelism.

4.3 Multi-GPU Experiments

Typically model parallelism is used when the model doesn't fit in a single GPU. For our experiments, we want to expand the search space to cases where the model fits in a single GPU as well. More generally, we want to answer the question, "if a model fits in N GPUs, does it make sense to use $k * N$ GPUs to optimize for latency, and how to optimally use the available GPUs for various workloads."

Model parallelism will always be beneficial for increasing throughput, since with the model shared across different GPUs as opposed to being replicated, each GPU can perform inference on a larger batch size. As a result, we will use minimizing latency as our main objective.

We limit our experiments to a single-node setting, since inter-node communication is more expensive and involves extra overhead.

4.3.1 Finding the Best Parallelism Strategy

Since the prefilling and generation stages have different performance characteristics, we analyzed them separately. For the prefilling experiments, we do not perform any generation. For the generation experiments, we use a small prefill sequence length of 3 before the generation.

We study two cases where the model either fits or doesn't fit in a single GPU. For these two cases, we use OPT-1.3B and OPT-13B, respectively, as the models.

The best strategies with the lowest latencies are in bold font.

Model Fits in a Single GPU - Prefilling

The model we use here is OPT-1.3B. We will try different combinations of batch sizes and input sequence lengths, and output sequence length 0.

- **Small Batch Size, Short Sequence Length**: Table 4.1 shows that in this case, model parallelism only increases the latency. To get the lowest latency, the best strategy here is to use data parallelism to replicate the same model across all GPUs, split the batch evenly across GPUs, and perform inference independently.

- **Big Batch Size, Short Sequence Length**: Table 4.2 shows that in this case, the best strategy to minimize latency is also to replicate the model and only do data parallelism across the batch dimension.

- **Small Batch Size, Long Sequence Length**: Table 4.3 shows that the best strategy to minimize latency here is also data parallelism.

- **Medium Batch Size, Medium Sequence Length**: Table 4.4 shows that the best strategy here is also data parallelism.

Table 4.1: MultiGPU latency experiments for OPT-1.3B with a small batch (4) size and short input sequence length (20) during the prefilling stage.

Batch Size	Input Sequence Length	TP	PP	Total GPUs	Latency (ms)
4	20	1	1	1	7.74
4	20	2	1	2	8.88
4	20	4	1	4	9.12
4	20	1	2	2	9.48
4	20	1	4	4	11.45
4	20	2	2	4	9.39
1	20	1	1	1	**6.17**

Table 4.2: MultiGPU latency experiments for OPT-1.3B with a big batch size (1000) and short input sequence length (20) during the prefilling stage.

Batch Size	Input Sequence Length	TP	PP	Total GPUs	Latency (ms)
1000	20	1	1	1	530.8
1000	20	2	1	2	348.5
1000	20	4	1	4	336.66
1000	20	1	2	2	341.72
1000	20	1	4	4	209.15
1000	20	2	2	4	234.86
250	20	1	1	1	**129.16**

Takeaways: In summary, there is no reason to use model parallelism during the prefilling stage to minimize latency if the model fits in a single GPU. Data parallelism is useful for decreasing latency.

Model Fits in a Single GPU - Generation

The model we use here is OPT-1.3B. We will try different combinations of batch sizes and output sequence lengths, and input sequence length 3.

- **Small Batch Size, Short Sequence Length**: Table 4.5 shows that tensor parallelism with TP=4 is the best strategy to minimize latency here.
- **Big Batch Size, Short Sequence Length**: Table 4.6 shows that the best strategy for minimizing latency here is to replicate the model across all GPUs

Table 4.3: MultiGPU latency experiments for OPT-1.3B with a small batch size (4) and long input sequence length (1000) during the prefilling stage.

Batch Size	Input Sequence Length	TP	PP	Total GPUs	Latency (ms)
4	1000	1	1	1	211.21
4	1000	2	1	2	128.58
4	1000	4	1	4	107.13
4	1000	1	2	2	148.23
4	1000	1	4	4	105.06
4	1000	2	2	4	92.92
1	1000	1	1	1	**57.3**

Table 4.4: MultiGPU latency experiments for OPT-1.3B with a medium batch size (128) and medium input sequence length (128) during the prefilling stage.

Batch Size	Input Sequence Length	TP	PP	Total GPUs	Latency (ms)
128	128	1	1	1	446.59
128	128	2	1	2	296.19
128	128	4	1	4	285.14
128	128	1	2	2	339.62
128	128	1	4	4	191.03
128	128	2	2	4	234.58
32	128	1	1	1	**116.81**

Table 4.5: MultiGPU latency experiments for OPT-1.3B with a small batch size (4) and short output sequence length (20) during the generation stage.

Batch Size	Output Sequence Length	TP	PP	Total GPUs	Latency (ms)
4	20	1	1	1	85.63
4	20	2	1	2	68.69
4	20	4	1	4	**60.07**
4	20	1	2	2	97.21
4	20	1	4	4	104.66
4	20	2	2	4	87.76
1	20	1	1	1	74.16

and do data parallelism on the batch dimension.

- **Small Batch Size, Long Sequence Length**: Table 4.7 shows that in this setting, tensor parallelism has lower latency than pipeline parallelism, and the best strategy to minimize latency is to use tensor parallelism on 4 GPUs. We also tried a smaller batch size (1) such that the batch cannot be split into local batches. The results in Table 4.8 show that for a batch size of 1, we have the same conclusion that TP=4 is the best parallelization strategy.

- **Medium Batch Size, Medium Sequence Length**: Table 4.9 shows that in this case, if we are using model parallelism, pipeline parallelism has lower latency than tensor parallelism, which means that there is probably benefit from splitting the batch into smaller mini-batches. This indicates that data parallelism is likely the best strategy, and our experiments confirmed this.

Takeaways: In the generation stage, using tensor parallelism in small batch settings is beneficial for decreasing latency; however, this doesn't scale infinitely. As the communication costs of tensor parallelism grow quickly, we will hit diminishing returns for using GPUs for tensor parallelism until eventually, the communication overhead exceeds the computation savings. In large batch settings, data parallelism by replicating the model is best for minimizing latency.

Table 4.6: MultiGPU latency experiments for OPT-1.3B with a big batch size (1000) and short output sequence length (20) during the generation stage.

Batch Size	Output Sequence Length	TP	PP	Total GPUs	Latency (ms)
1000	20	1	1	1	688.64
1000	20	2	1	2	537.67
1000	20	4	1	4	611.5
1000	20	1	2	2	423.92
1000	20	1	4	4	310.48
1000	20	2	2	4	387.51
250	20	1	1	1	**215.68**

Table 4.7: MultiGPU latency experiments for OPT-1.3B with a small batch size (4) and long output sequence length (1000) during the generation stage.

Batch Size	Output Sequence Length	TP	PP	Total GPUs	Latency (ms)
4	1000	1	1	1	4778.18
4	1000	2	1	2	3732.89
4	1000	4	1	4	**3151.72**
4	1000	1	2	2	4949.62
4	1000	1	4	4	5079.32
4	1000	2	2	4	4109.59
1	1000	1	1	1	3907.22

Table 4.8: MultiGPU latency experiments for OPT-1.3B with a small batch size (1) and long output sequence length (1000) during the generation stage.

Batch Size	Output Sequence Length	TP	PP	Total GPUs	Latency (ms)
1	1000	1	1	1	3907.22
1	1000	2	1	2	3072.43
1	1000	4	1	4	**2791.28**
1	1000	1	2	2	3900.44
1	1000	1	4	4	3937.71

Table 4.9: MultiGPU latency experiments for OPT-1.3B with a medium batch size (512) and medium output sequence length (256) during the generation stage.

Batch Size	Output Sequence Length	TP	PP	Total GPUs	Latency (ms)
512	256	1	1	1	8620.12
512	256	2	1	2	6558.72
512	256	4	1	4	5748.1
512	256	1	2	2	5681.33
512	256	1	4	4	3635.71
128	256	1	1	1	**2844.58**

Model Does Not Fit in a Single GPU - Prefilling

The model we use here is OPT-13B. We will try different combinations of batch sizes and input sequence lengths, and output sequence length 0.

- **Small Batch Size, Short Sequence Length**: Table 4.10 shows that the best strategy here is to use tensor parallelism to parallelize the model on as few GPUs as possible first. Then replicate the same parallelization scheme on the remaining available GPUs. Specifically, we get the lowest latency with 4 GPUs having 2 copies of the model, and each model having 2-way tensor parallelism.
- **Big Batch Size, Short Sequence Length**: Table 4.11 shows that the best strategy here is the same as the small batch size, short sequence length case — 4 GPUs with 2 copies of the model, each model with 2-way tensor parallelism.
- **Small Batch Size, Long Sequence Length**: Table 4.12 shows that the best strategy here is also the same — 4 GPUs with 2 copies of the model, each model with 2-way tensor parallelism.
- **Medium Batch Size, Medium Sequence Length** Table 4.13 shows that the best strategy here is also 4 GPUs with 2 copies of the model, each model with 2-way tensor parallelism.

Takeaways: For prefilling, to minimize latency, tensor parallelize the model on as few GPUs as possible and then replicate the same model and parallelization strategy

Table 4.10: MultiGPU latency experiments for OPT-13B with a small batch size (4) and short input sequence length (20) during the prefilling stage.

Batch Size	Input Sequence Length	TP	PP	Total GPUs	Latency (ms)
4	20	2	1	2	15.9
4	20	4	1	4	15.53
2	20	2	1	2	**12.2**
4	20	1	2	2	18.51
4	20	1	4	4	22.2
2	20	1	2	2	17.1
4	20	2	2	4	17.19

Table 4.11: MultiGPU latency experiments for OPT-13B with a big batch size (1000) and short input sequence length (20) during the prefilling stage.

Batch Size	Input Sequence Length	TP	PP	Total GPUs	Latency (ms)
1000	20	2	1	2	752.06
1000	20	4	1	4	714.11
500	20	2	1	2	**377.69**
1000	20	1	2	2	767.42
1000	20	1	4	4	466.87
500	20	2	1	2	399.27
1000	20	2	2	4	509.5

Table 4.12: MultiGPU latency experiments for OPT-13B with a small batch size (4) and long input sequence length (1000) during the prefilling stage.

Batch Size	Input Sequence Length	TP	PP	Total GPUs	Latency (ms)
4	1000	2	1	2	263.74
4	1000	4	1	4	224.27
2	1000	2	1	2	**138.14**
4	1000	1	2	2	303.81
4	1000	1	4	4	212.62
2	1000	2	1	2	179.73
4	1000	2	2	4	184.57

Table 4.13: MultiGPU latency experiments for OPT-13B with a medium batch size (128) and medium input sequence length (128) during the prefilling stage.

Batch Size	Input Sequence Length	TP	PP	Total GPUs	Latency (ms)
128	128	2	1	2	643.34
128	128	4	1	4	597.78
64	128	2	1	2	**325.51**
128	128	1	2	2	709.15
128	128	1	4	4	394.86
64	128	1	2	2	341.99
128	128	2	2	4	465.64

across the remaining GPUs. This is a generalization of the case where the model fits in a single GPU — if a model fits in N GPUs, tensor parallelize across N GPUs. Replicate the model and use the same parallelization strategy combined with data parallelism on the remaining GPUs if the total number of GPUs is a factor of N. With bigger models, the communication cost of tensor parallelism grows fairly quickly since there are more layers and the communicated tensor at each layer is also bigger.

Model Does Not Fit in a Single GPU - Generation

The model we use here is OPT-13B. We will try different combinations of batch sizes and output sequence lengths, and input sequence length 3.

- **Small Batch Size, Short Sequence Length**: Table 4.14 shows that the best strategy here is to use tensor parallelism to parallelize the model on 4 GPUs.

- **Big Batch Size, Short Sequence Length**: Table 4.15 shows that the best strategy here is to use pipeline parallelism to parallelize the model on as few GPUs as possible first. Then replicate the same parallelization scheme on the remaining available GPUs.

- **Small Batch Size, Long Sequence Length**: Table 4.16 shows that the best strategy here is to use tensor parallelism to parallelize the model on 4 GPUs.

Table 4.14: MultiGPU latency experiments for OPT-13B with a small batch size (4) and short output sequence length (20) during the generation stage.

Batch Size	Output Sequence Length	TP	PP	Total GPUs	Latency (ms)
4	20	2	1	2	132.95
4	20	4	1	4	**106.36**
2	20	2	1	2	129
4	20	1	2	2	182.53
4	20	1	4	4	187.53
2	20	1	2	2	165.32
4	20	2	2	4	145.06

Table 4.15: MultiGPU latency experiments for OPT-13B with a big batch size (1000) and short output sequence length (20) during the generation stage.

Batch Size	Output Sequence Length	TP	PP	Total GPUs	Latency (ms)
1000	20	2	1	2	997.22
1000	20	4	1	4	1069.39
500	20	2	1	2	559.76
1000	20	1	2	2	939.69
1000	20	1	4	4	591.3
500	20	1	2	2	**518.45**
1000	20	2	2	4	675.5

- **Medium Batch Size, Medium Sequence Length**: Table 4.17 shows that the best strategy here is to use pipeline parallelism to parallelize the model on as few GPUs as possible first. Then replicate the same parallelization scheme on the remaining available GPUs.

Takeaways: For small batch sizes, tensor parallelism on as many GPUs as possible is useful for minimizing latency. In large batch settings, if a model fits in at least N GPUs, pipeline parallelize across exactly N GPUs. Replicate the model and use the same parallelization strategy combined with data parallelism on all the remaining GPUs.

Table 4.16: MultiGPU latency experiments for OPT-13B with a small batch size (4) and long output sequence length (1000) during the generation stage.

Batch Size	Output Sequence Length	TP	PP	Total GPUs	Latency (ms)
4	1000	2	1	2	7150.58
4	1000	4	1	4	**5707.08**
2	1000	2	1	2	6808.27
4	1000	1	2	2	9289.93
4	1000	1	4	4	9245.15
2	1000	1	2	2	8250.48
4	1000	2	2	4	7280.16

Table 4.17: MultiGPU latency experiments for OPT-13B with a medium batch size (128) and medium output sequence length (128) during the generation stage.

Batch Size	Output Sequence Length	TP	PP	Total GPUs	Latency (ms)
128	128	2	1	2	1905.61
128	128	4	1	4	1825.45
64	128	2	1	2	1448.68
128	128	1	2	2	1716.89
128	128	1	4	4	1544.12
64	128	1	2	2	**1401.62**
128	128	2	2	4	1570.53

Chapter 5

Survey of Existing Transformer Frameworks

5.1 Other Parallelism and Model Sharding Strategies

In addition to data parallelism, tensor parallelism, and pipeline parallelism we mentioned in the earlier sections, there are also parallelism strategies that we did not use in our experiments but will describe here.

5.1.1 Zero Redundancy Optimizer (ZeRO)

ZeRO (Zero Redundancy Optimizer) [24] is a parallelism solution developed by Microsoft aimed at overcoming the limitations of data and model parallelism while achieving the merits of both. ZeRO partitions the model states and optimizer states across the devices such that it can fit a model of the size of the aggregate memory of all available devices. There are three main optimization stages in ZeRO which correspond to various degrees of model and optimizer state partitioning:

1. Optimizer state partitioning, also called "Zero-1"
2. Gradient partitioning, also called "Sharded DDP", "Zero-2"

3. Parameter partitioning, also called "Fully Sharded Data Parallelism", "Zero-3"

The ZeRO paper claims that stages 1 and 2 result in the same data communication volume as naive data parallel, and stage 3 incurs a modest 50% increase in communication cost.

Computation is done on one or some layers of the model at a step, and each device receives a different data batch. When computing a specific layer or layers group, the GPU with the parameters and states associated with this specific layer/layers group sends those states to all the GPUs. After this computation step, all the GPUs except the one originally dedicated to storing the model parameters of the layer/layer group can delete the received parameters. This repeats in the next layer/layer group with a different GPU until the computation has been done for all layers of the partitioned model.

Only Zero-3 is relevant for inference, since there will be no need for optimizer state or gradient sharding during inference.

5.1.2 3D Parallelism

3D Parallelism [5] refers to using a combination of data, model, and tensor parallelism. Since tensor parallelism (Megatron-style) has the largest communication costs, 3D parallelism prioritizes placing model parallel groups within a single compute node to utilize the intra-node bandwidth, which is higher than the inter-node communication bandwidth. Pipeline parallelism has lower communication costs due to a lower communication volume, so it is used across nodes.

5.1.3 CPU Offloading

In the case where the full model doesn't fit in the collective memory of all the GPUs, it is also possible to use CPU-offloading, which refers to storing the model weights, optimizer states, or gradients in the main memory and loading them to the GPU when it is needed to perform computations. CPU offloading is supported by Deepspeed ZeRO-Offload [6]. CPU offloading enables the use of large models on low-cost hardware without requiring as many GPUs and can increase the throughput of tasks; however, throughput is a tradeoff with latency. FlexGen [21], which is a "high-throughput engine for running large language models with limited GPU memory", demonstrated that in limited GPU-resources settings, it is possible to have throughput-oriented workloads through using CPU offloading, but the latency is significantly higher than the case where we have enough GPUs to hold the whole model.

5.2 Other Frameworks and Implementations

In addition to the libraries we used for our experiments, HuggingFace Transformers and Nvidia FasterTransformer, we also include a survey of various existing frameworks that include optimized transformer-specific libraries implementing transformer-baesd architectures, or frameworks or features that help with parallelizing large models.

5.2.1 FairScale

FairScale [9] is a Pytorch extension library by Facebook for large scale training. It supports pipeline parallelism, Megatron-style tensor parallelism, and ZeRO (all three stages), and it has been integrated with frameworks including Fairseq, Pytorch Lightning, HuggingFace. The pipeline parallelism is forked from TorchGPipe, which is a scalable pipeline parallelism library published by Google Brain and based on GPipe. The model (tensor) parallelism is forked from Megatron-LM.

5.2.2 Native PyTorch

Pytorch Distributed Data Parallel

In the case where a model can fit in the memory of a single GPU, simple data parallel can be used to easily scale up computation across multiple GPUs by replicating the same model across all the GPUs, so that each GPU can work on a different batch of the data. In Pytorch, for example, this is implemented in the DistributedDataParallel (DDP) modules. Using DDP requires minimal modification to the serial code, since the user only needs to wrap the original model in a DDP class and specify some device parameters.

Pytorch Pipeline Parallelism

Pipeline parallelism is a natively supported feature in PyTorch and is subject to change. It is based on FairScale's pipe implementation of pipeline parallelism and torchgpipe.

PyTorch Fully Sharded Data Parallelism

Pytorch's FSDP is a native implementation of ZeRO (stages 1-3). Its implementation is based on FairScale's version.

5.2.3 DeepSpeed

DeepSpeed [4] is an open-source library by Microsoft that implements ZeRO parallelism. It has been integrated with several different open-source DL frameworks such as Huggingface Transfomers, Huggingface Accelerate, Lightning, and MosaicML. DeepSpeed is also able to support pipeline and tensor parallelism, but this has not been fully integrated with other DL frameworks such Huggingface Accelerate.

5.2.4 Megatron-LM

Megatron-LM [20] is a large transformer developed by Nvidia, and the Megatron-LM Github repository is used on ongoing research for training large transformer based language models at scale. Many existing implementations of model parallelism use the Megatron-style tensor (model) parallelism. The Megatron repository implements tensor, sequence, and piepeline parallelism.

5.2.5 Megatron-DeepSpeed

Megatron-DeepSpeed [12] is a fork of the Megatron-LM Github repository that also adds in other features such as mixture of expert model training, curriculum learning, 3d parallelism, and others.

5.2.6 HuggingFace Transformers

The HuggingFace Transformers library [10] provides APIs to easily download and use the pretrained models available on HuggingFace. Transformer currently integrates the DeepSpeed and Fairscale implementations of all three stages of zero. Transformers currently does not support full pipeline parallelism or tensor parallelism, although some models such as GPT2 and T5 have naive pipeline parallelism support.

5.2.7 Parallelformers

Parallelformers [15] is based on MegatronLM and is designed to make model parallelization with the HuggingFace Transformers library easier. Currently Parallelformers only supports inference but not training.

5.2.8 Huggingface Accelerate

Huggingface Accelerate [11] is a library that enables Pytorch code to be run in distributed configurations. It provides interfaces for other frameworks including Pytorch's FSDP, DeepSpeed, and Megatron-LM.

5.2.9 Facebook Fairseq

Fairseq [7] by Meta (Facebook) is a language modeling toolkit that allows researchers to train custom language models. Fairseq supports Megatron-style tensor parallelism, fully sharded data parallelism, and CPU offloading.

5.2.10 Facebook Metaseq

Metaseq [8] is a fork of Fairseq that removed most features offered in Fairseq to enable faster iteration, leaving the bare minimum that is needed for working at the 100B parameters model scale. It is a codebase specifically for working with OPT (Open Pretrained Transformers)[27].

5.2.11 ByteDance LightSeq

ByteDance LightSeq [3] is a high performance library supporting both inference and training for sequence related tasks. The library is built on top of CUDA with custom kernel functions that have fusion patterns optimized for transformer-based models. LightSeq supports Megatron-style tensor parallelism for training, data parallelism, and FSDP, but it does not offer pipeline parallelism.

5.2.12 Tencent TurboTransformers

Tencent TurboTransformers [16] is an inference engine for accelerating transformer inference. It includes an efficient parallel algorithm for GPU-based batch reduction operations, a memory allocation system for balancing memory footprint and allocation / free efficiency, and a batch scheduler based on dynamic programming to achieve optimal throughput on variable-length sequences.

5.2.13 Alpa

Alpa [1] is a system for automatically parallelizing large deep learning models by generating execution plans that use a combination of data, tensor, and pipeline parallelism in a performant way. It is currently available in Jax, XLA, and Ray.

Chapter 6

Conclusion

6.1 Summary

In conclusion, this thesis focused on analyzing the performance of transformer inference for GPT-like architectures through a combination of an analytical model and empirical studies using existing frameworks. We distilled takeaways and observations that provide insights into the performance characteristics of transformer inference and offer recommendations for effective parallelization of large language models.

The analysis of parameter and FLOP counts revealed that MLP parameters constitute a significant portion of the total, while attention parameters and play a crucial role as well. As models increase in size, the relative importance of embedding parameters diminishes. Most operations scale linearly with sequence length and quadratically with the hidden dimension. The MLP operation contributes the most FLOPs, while KQV computations dominate in attention layers.

A comparison between prefilling and generation stages highlighted the difference in performance characteristics. FLOPs for generation grow quadratically with sequence length and can be much higher than prefilling. Although using a KV cache reduces the gap, generation remains slower due to low arithmetic intensity operations.

The empirical studies using Huggingface Transformers and Nvidia FasterTransformer on single GPUs demonstrated the hardware utilization and latency differences between the frameworks. The analytical utilization and FLOPs utilization curves exhibited similarities for prefilling, indicating efficient hardware utilization of both the implementation and the architecture. The generation curves indicate an efficient implementation but the architecture itself is cannot have high FLOPs utilization of the GPU. The results showed that Nvidia FasterTransformer outperformed Huggingface Transformers in terms of both FLOPs utilization and memory efficiency, although at the expense of usability.

The analysis further extended to multi-GPU inference, providing parallelization strategies for cases when the model fits in a single GPU and when it does not.

Overall, this thesis provided insights and recommendations for the performance analysis and parallelization of transformer inference in GPT-like architectures. The combination of the analytical model and empirical studies provides a comprehensive understanding of the performance characteristics and trade-offs involved, which can guide researchers and practitioners in optimizing the utilization of hardware resources and improving the efficiency of large language models.

6.2 Future Work

Analytical Model

- A more sophisticated activation memory model for the analytical model; for example, the analysis for the exact minimum lifetime of a tensor, and more accurate modeling of memory access times.
- Take into account more types of overheads, such as the kernel launching time.
- The analytical model currently is not able to model CPU offloading and ZeRO parallelism. Extending the model to capture these strategies can expand the search space of efficient parallelism strategies.

Experiments

- Setting up frameworks, especially in a distributed setting, can be difficult and time-consuming, and running experiments require a lot of GPU hours. We chose to perform our experiments with two representative frameworks — Huggingface Transformers was selected for its popularity and usability, and Nvidia FasterTransformer was selected for its performance. Running experiments with more frameworks would allow us to have a better understanding of the existing implementations.

- We did not explore multi-node inference because we think that the communication cost would be too high to justify using model parallelism beyond a single node when the model fits in a single node, but experiments confirming this would be assuring.

Parallelism Solutions

- Develop a framework that automatically parallelizes a GPT-like model in the most efficient way given the runtime configurations (batch size, input and output sequence lengths.)
- For tensor parallelism in transformers, the pattern is predefined by the megatron paper. A more general system to analyze the communication costs and effectiveness of all possible tensor parallelism patterns would be valuable in exploring a larger design space.

Bibliography

[1] Alpa. https://alpa.ai/index.html. Accessed: 2023-05-10.

[2] Bloom. https://huggingface.co/bigscience/bloom. Accessed: 2023-05-10.

[3] Bytedance lightseq. https://github.com/bytedance/lightseq. Accessed: 2023-05-10.

[4] Deep speed github. https://github.com/microsoft/DeepSpeed. Accessed: 2023-05-10.

[5] Deepspeed: Extreme-scale model training for everyone. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/. Accessed: 2023-05-10.

[6] Deepspeed zero offload. https://www.deepspeed.ai/tutorials/zero-offload/. Accessed: 2023-05-10.

[7] Facebook fairseq. https://github.com/facebookresearch/fairseq. Accessed: 2023-05-10.

[8] Facebook metaseq. https://github.com/facebookresearch/metaseq. Accessed: 2023-05-10.

[9] Fairscale. https://fairscale.readthedocs.io/en/latest/what$_is_fairscale.html.Accessed:$ $2023-05-10$.

[10] Hugging face model parallelism. https://huggingface.co/transformers/v4.9.2/parallelism.html. Accessed: 2023-05-10.

[11] Huggingface accelerate. https://huggingface.co/docs/accelerate/index. Accessed: 2023-05-10.

[12] Megatron deep speed github. https://github.com/microsoft/Megatron-DeepSpeed. Accessed: 2023-05-10.

[13] Mit satori cluster. https://mit-satori.github.io/satori-basics.htmlwhat-is-satori. Accessed: 2023-05-10.

[14] Nvidia fastertransformer. https://github.com/NVIDIA/FasterTransformer. Accessed: 2023-05-10.

[15] Parallelformers. https://github.com/tunib-ai/parallelformers. Accessed: 2023-05-10.

[16] Tencent turbotransformers. https://github.com/Tencent/TurboTransformers. Accessed: 2023-05-10.

[17] Turing-nlg: A 17-billionparameter language model by microsoft. microsoft research blog, 2:13. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/. Accessed: 2023-05-10.

[18] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[20] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters. *CoRR*, abs/2104.04473, 2021.

[21] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022.

[22] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2018.

[23] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[24] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training A trillion parameter models. *CoRR*, abs/1910.02054, 2019.

[25] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie

Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model, 2022.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[27] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.