Reimplementing FlexGen - High-Throughput Generative Inference of Large Language Models with a Single GPU

Atif Abedeen aabedeen@umass.edu
Shreyans Babel sbabel@umass.edu
Siddharth Jain siddharthjai@umass.edu

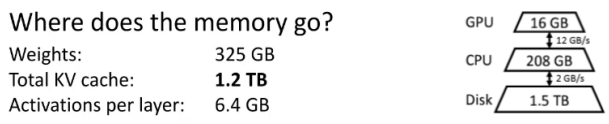
The University of Massachusetts Amherst

1 Introduction

After the introduction of Large Language Models by Vaswani in 2017 [5], the increasing complexity and size of Large Language Models (LLMs) have made their deployment on limited-resource hardware extremely challenging. These models are essential for advancing natural language processing but typically require extensive computational resources, leading to significant deployment challenges on standard hardware. There have been a lot of research that attempts to either improve the architecture, attention mechanisms, prompt tuning, RLHF, and model serving to get optimize this system and reduce carbon footprint. FlexGen, based on the conceptual foundation laid by the authors [4], is a system is one of those attempts designed to enable high-throughput generative inference of LLMs on a single commodity GPU. This approach helps increase throughput for batched jobs by cleverly using and moving resources across the memory hierarchy. Our project, centered around the FlexGen system, aims to engineer a similar process on systems with constrained GPU capabilities by utilizing a sophisticated architecture that minimizes memory overhead and maximizes computational efficiency.

1.1 Problem Statement

The primary problem addressed by the FlexGen system is the computational and memory challenges associated with deploying large language models (LLMs) on limited-resource hardware, specifically single GPU setups. Traditional methods struggle due to the high memory demands of LLMs and the inefficiencies in input/output operations between the GPU, CPU, and disk. Moreover, offloading the KV Caches and movement between CPU/GPU/Disk incurs huge I/O costs.



OPT-175B in FP16: (batch size 512, input seq len 512, output seq len 32)

These challenges result in significant performance bottlenecks, particularly for throughput-centric tasks that require processing large volumes of data efficiently rather than optimizing for latency. The goal of this project is to optimize memory management and computation strategies as done in the FlexGen system, allowing for high-throughput generative inference of LLMs on batch prediction jobs.

1.2 Proposed Solution

To address the problem, the FlexGen system introduces an architecture that strategically divides computational tasks between the CPU and GPU to optimize both memory usage and processing efficiency. This hybrid CPU-GPU architecture enables the CPU to handle sequential, intensive tasks like the initial generation of Key (K), Query (Q), and Value (V) tensors, reducing the computational load on the GPU, which is reserved for parallelizable tasks such as activation generation through Multi-Head Attention (MHA).

1.3 Key Aspects of FlexGen's Enhanced Efficiency

1.3.1 Efficient Tensor Management and Reduced I/O Costs

- Optimized Layer-wise Loading: Traditional computational models typically load all necessary model weights into GPU memory either at once or in large, non-optimized batches. This often leads to frequent swapping of weights in and out of memory, especially when GPU capacity is exceeded, incurring high I/O costs due to significant data movement across the GPU, CPU, and potentially disk storage. When data must be offloaded to disk due to memory constraints, the time required for disk read/write operations substantially increases computational latencies and diminishes throughput.

  FlexGen addresses these issues by employing optimized layer-wise loading, where it only loads layers of weights one by one into the GPU for processing large batches of input tokens and are retained until the giant batch's KV and activation values are calculated and stored. This minimizes the need for frequent access to slower secondary storage.

- Strategic Offloading: FlexGen utilizes a strategic zig-zag computational pattern that aligns the loaded data more precisely with computational demands. This approach allows for multiple operations to be performed while the data resides in the fastest-access memory, significantly reducing the time and energy associated with data loading and unloading, thereby enhancing overall system efficiency and throughput.

1.3.2 Throughput Optimization via Batch and Block Sizing

The high-throughput generation engine improves throughput by:

- Dynamic Batch Sizing: FlexGen's ability to use larger effective batch sizes allows for more efficient processing of data. By adjusting the batch size based on the memory availability and computational strategy (e.g., the zig-zag approach), FlexGen can maximize the GPU utilization, thus increasing the throughput.

- Block Scheduling: The block schedule approach in FlexGen allows for extensive reuse of data loaded into the memory, which minimizes the need for data transfer and maximizes the throughput by effectively utilizing computational and memory resources.

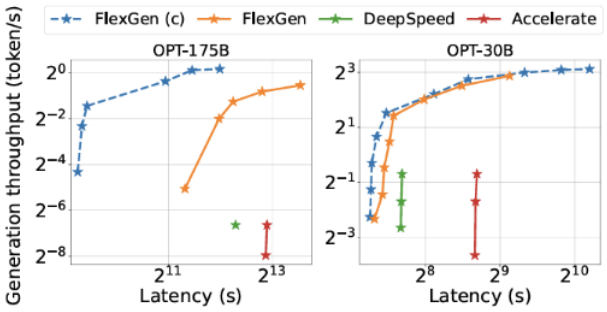


Figure 1: The total latency for a block and throughput trade-offs of three offloading-based systems. "(c)" denotes compression

1.3.3 Compression Techniques to Minimize Memory Footprint

Standard approaches may not implement advanced compression techniques, leading to a larger memory footprint for storing weights and activations. This larger footprint can limit the model's capacity to run on constrained hardware, as more frequent memory swaps are required. FlexGen employs fine-grained group-wise quantization, reducing the memory requirements of weights and KV caches by compressing them to 4 bits. This approach allows much larger models to fit into the limited memory, reducing the need for swaps and enabling higher throughput.

2 Methodology

Let's delve deep into the architecture of our system and the algorithmic framework behind it.

2.1 Hybrid CPU-GPU Architecture

As discussed earlier, we implement a hybrid CPU-GPU architecture designed to efficiently manage and balance computational loads based on the characteristics of each task:

2.1.1 CPU Responsibilities

- Initial Data Processing: The CPU is responsible for initial computations such as generating Key (K), Query (Q), and Value (V) tensors from input tokens. This is done on the CPU to avoid the huge I/O cost. If the K,Q,V calculations were to be done on the GPU, there will be huge time consumption while transferring huge KV cache from GPU to CPU or disk,

- Attention Calculations: Key matrix transformations and the computation of attention scores are performed on the CPU to utilize its superior ability to handle these sequential tasks without the bandwidth limitations of transferring data back to the GPU. This also allows FlexGen to focus on data movement from GPU to CPU asynchronously while the computations are still being done on the CPU.

2.1.2 GPU Responsibilities

- Parallelizable Tasks: The GPU is utilized for highly parallelizable tasks such as computing activation maps from the results of the attention scores and final output generation. This ensures that the GPU's processing power is maximized for operations that benefit most from parallel processing.

- Activation Handling: Activations generated during the attention process are handled directly in the GPU to take advantage of its faster processing speeds, reducing the time required for data transfers between the CPU and GPU. This helps prevent the huge I/O costs of sending larger KV cache to GPU and instead when needed the activations are brought back to the CPU for attention calculation.

2.2 Optimized Data Flow

The next part of the algorithm employs a zigzag traversal pattern for managing data flow between the CPU, GPU, and disk, optimizing computational efficiency by minimizing unnecessary data movement.

In Figure 2, the grid illustrates the basic computational graph of the LLM inference in FlexGen. The inference process is visualized as a series of batches processed across different layers of the model. Each

square represents a GPU batch while the different colors refer to the different layers.

Normally, systems process data row-by-row across layers (Figure 2a). Even though this method is simple, it incurs high I/O costs because of frequent loading of weights and offloading of KV caches, as layers do not share weights between batches. Another approach is the column-by-column processing where we can traverse the graph column-by-column. So in this case, all squares in a column will share weights, so we can let the weights stay on the GPU and only load/unload the activations and KV cache. But this approach also has its limitations. A column cannot be traversed all the way to the end because the activations and KV cache still need to be stored CPU and disk memory are limited.

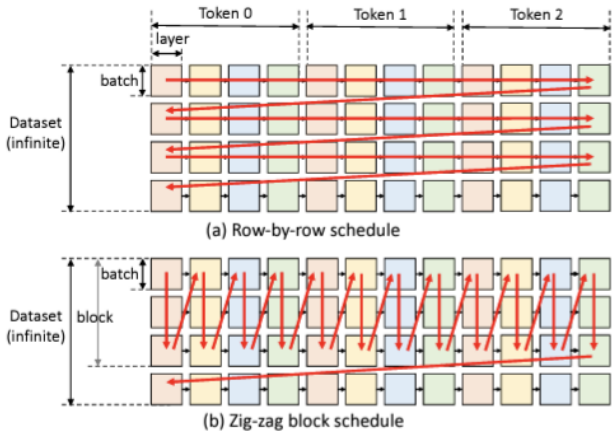


Figure 2: Two different schedules. The red arrows denote the computation order.

So FlexGen uses a combination of both these processes to create a method called the Zig Zag approach, which organizes the data processing in a more diagonal or column-wise manner. The input data is divided into batches. Each batch contains multiple input sequences that can be processed in parallel.

For each batch, the processing might begin with the first layer and then, instead of moving to the next layer immediately, the system could decide to process another part of the batch through the same layer depending on memory availability and computational dependencies. The diagonal processing across layers means that for any given batch, as soon as the necessary data (like KV caches) from the previous layers are available, the next layer can start processing its part without waiting for the entire previous layer to complete across all batches. This creates the zig-zag movement through the grid of batch-layer intersections (Figure 2b).

This approach allows for overlapping of data fetching (especially KV caches) and computation across layers, which optimizes both the CPU and GPU usage by ensuring that both are kept busy with minimal idle time.

2.3 Detailed Algorithmic Framework

2.3.1 Prefill Stage Implementation

Step-by-step dataflow -

- Layer-wise KV Generation: Each transformer layer is processed sequentially. Key (K), Query (Q), and Value (V) tensors are calculated using the current input tokens and stored layer weights.

- In a typical transformer model, there are several matrices of weights (W_Q, W_K, W_V) which are applied to the input embeddings to generate queries (Q), keys (K), and values (V). In FlexGen, these weights are loaded into the CPU from where they can be used for computation without the need for constantly loading them into the GPU memory. This minimizes data movement and saves time.

- Multi-Head Attention (MHA) is calculated from K, Q, and V tensors. This computation is split across the CPU for tensor calculations and the GPU for generating activation maps, optimizing the use of both types of processors and minimizing memory transfer costs.

- To manage the memory footprint, K and V tensors are offloaded to the disk immediately after use, while activations are kept in GPU memory for quick access during subsequent computations.

In lines 3-8 of Algorithm 1, the model loads weights into the CPU for each layer (j) and for each batch (k). This reflects the initial handling of different layers and preparation for batch processing, aligning with the Zig Zag approach's strategy of handling data non-linearly to optimize memory and processing power.

Lines 11-17 detail the calculation of Key (K), Query (Q), and Value (V) tensors, followed by their usage in the attention mechanism (Softmax(K $\times$ Q) $\times$ V) as illustrated in Figure 3, and the subsequent sending of attention scores to the GPU. This shows the efficient use of the CPU for computational heavy lifting and the GPU for executing tasks parallely.

Algorithm 1 Prefill Stage of FlexGen Model

```
1: procedure PREFILL
2:     InputEMB
3:     for j = 1 to N do
4:         LoadWeightsCPU(layer j)
5:         for k = 1 to K do
6:             if j = 1 then
7:                 x ← output from InputEMB
8:             else
9:                 x ← GetActivationsGPU(layer j − 1, batch k)
10:            end if
11:            K ← W_k × x
12:            Q ← W_q × x
13:            V ← W_v × x
14:            AttentionScore ← Softmax(K × Q) × V
15:            SendToGPU(AttentionScore)
16:            AppendKVStorage(kv_home[j][k], K, V)
17:            UpdateActivationGPU(AttentionScore, k)
18:        end for
19:    end for
20:    OutputEMB
21: end procedure
```

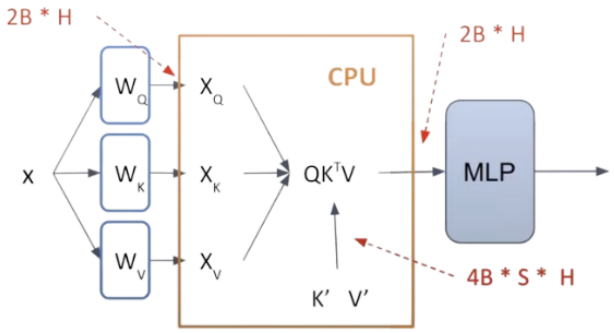


Figure 3: Illustrating the CPU's Role in Handling K, Q, and V Computations and Subsequent Processing by the MLP (Multi-Layer Perceptron)

2.3.2 Decode Stage Implementation

- The decode stage uses the KV cache to generate tokens one at a time, applying the zigzag method to optimize memory use and computational efficiency. Lines 12-14 describe the retrieval and update of KV caches where new K and V tensors are concatenated with existing ones. This operation illustrates the Zig Zag approach's effectiveness in reducing I/O cycles and improving computational flow by dynamic memory management and processor usage.

- Newly computed K and V tensors are merged with the pre-existing cache, minimizing redundant computations and enhancing response times.

We performed multiple experiments using the strategies explained above and we discuss about the results in the next section.

Algorithm 2 Decode Stage of FlexGen Model

```
1: procedure DECODE
2:     InputEMB
3:     for j = 1 to N do
4:         LoadWeightsCPU(layer j)
5:         for k = 1 to K do
6:             if j = 1 then
7:                 x ← output from InputEMB
8:             else
9:                 x ← GetActivationsGPU(layer j − 1, batch k)
10:            end if
11:            (old_k, old_v) ← RetrieveKV(kv_home[j][k])
12:            new_k ← W_k × x
13:            new_v ← W_v × x
14:            K ← Concatenate(old_k, new_k)
15:            V ← Concatenate(old_v, new_v)
16:            AttentionScore ← Softmax(K × Q) × V
17:            SendToGPU(AttentionScore)
18:            ReplaceKV(kv_home[j][k], K, V)
19:            UpdateActivationGPU(AttentionScore, k)
20:        end for
21:    end for
22:    OutputEMB
23: end procedure
```

3 Results

Our implementation of FlexGen focused on replication of FlexGen's core functionality of zig-zag approach for LLM inference on a single GPU. We ran experiments and benchmarks them using the OPT-1.3B language model on Google Colab. Our default hyperparameters that we chose for all the experiments are provided in Fig 4 Results to our experiments with comparisons of No FlexGen vs Original FlexGen vs Our FlexGen are compiled below.

Hyperparameter	Value
Name	opt-1.3
Maximum Sequence Length	24
Number of Hidden Layers	24
Number of Attention Heads	32
Hidden Size	2048
Input Dimension	2048
Vocabulary Size	50272
Data Type	float16
Number of GPU Batches	4
GPU Batch Size	4
Prompt Length	512

Figure 4: Default Hyperparameters for the experiments

3.1 Throughput Analysis

In this section, we report, compare and analyze the throughput results achieved by three approaches. We evaluate the performance of each approach based on the number of tokens generated per second, which serves as a measure of throughput shown in Table (1) .

- **Without FlexGen / Default Way**

  This approach refers to the traditional approach of using LLMs without any optimization techniques. Here, the whole model is loaded into GPU memory and the whole inference is carried out sequentially on a small batch of inputs. The throughput achieved by this approach is higher because in this case OPT 1.3b fits entirely on the GPU with the data and activations. We believe that as we move towards bigger models, the effect of FlexGen is much higher, since there will be huge I/O costs associated with moving weight, activations, and KV caches from one device to another.

  We then ran this script on the CPU only (all computations on the CPU) and after running for an hour the code crashed and we could not collect any results for this experiment.

- **With original FlexGen**

  As talked about earlier, the original FlexGen approach combines multiple optimization techniques to get extremely high throughput values. This approach uses an optimal policy to calculate data distributions, parallel execution of tasks, uses compression, and use a efficient data offloading strategies through the memory hierarchy.

- **With our FlexGen**

  Our implementation aimed to replicate the core project ideas. However due to limited computational resources, and knowledge and experience, we were not able to achieve the original FlexGen level numbers. While we followed the overall zigzag approach and offloading strategies, our implementation currently performs loading, computation, and storage of tensors sequentially, which is reasons for a lower throughput values. Even though we couldn't replicate the scores, we still tried our best to analyze where our approach felt behind. We report an in-depth profiling analysis in the next section.

Table 1: Throughput comparison (tokens/second)

Approach	Throughput
Default (CPU)	OOM[1]
Default (GPU)	15.6
Original FlexGen	22.7
Our FlexGen	6.7

This table shows the throughput comparison for different approaches, with the original FlexGen implementation achieving the highest throughput.
OOM : Out of Memory

3.2 Profiler Analysis

To profile the three programs (FlexGen, our implementation of FlexGen, and inference without FlexGen), we used two profilers: Pytorch's built-in profiler and Scalene [2]. Scalene is a powerful profiler that helped us narrow down exact lines of code that were bottleneck for both memory and latency. We not only used it for comparison between the three systems but also used it to understand what was causing the bottleneck and see how we could optimize our code further. This profiler also breaks down the time into three categories: python, native, and system. Python is basically the time spent in executing the main python script, native is the time taken to execute the low-level impelementation of each of the functions and libraries used, and finally, system is the time taken in I/O operations.

3.2.1 Without FlexGen:

We first ran the scalene profiler on the GPU without FlexGen to demonstrate the utilization of the CPU and GPU resources.

Time %			
Python	Native	System	
< 1	77	< 1	torch.cat((input, torch.tensor([output_ids[-1]])
5	< 1	< 1	h = self.mha(q, k, v)
< 1	< 1	< 1	print(prof.key_averages().table(sort_by="self_cpu_time_total"))
< 1	< 1	< 1	print(prof.key_averages().table(sort_by="self_cuda_time_total"))

Figure 5: Showing top 5 results of Scalene Profiler for inference without FlexGen on the GPU

Fig. 5 shows the top 5 lines in the program that takes the most amount of time. It is evident that it is very efficient and the total time it took to finish executing was around 4 minutes. Other than the native code, there were no major bottlenecks.

Name	Self-CUDA	Self-CPU
aten::addmm	121.923s	3.292s
cudaLaunchKernel	8.403s	1.411s
cudaMemsetAsync	4.084s	498.758ms
cudaMemcpyAsync	51.041ms	123.377s

Figure 6: Showing top 4 results of Pytorch's Profiler for inference without FlexGen on the GPU

In Fig. 6, we show the results of the pytorch's built-in profiler. We can see that the matrix multiplication operator takes the most amount of time followed by the operator that launches the CUDA kernel and the operator that handles I/O between CPU and GPU. Again as before, We then ran this script on the CPU only (all computations on the CPU) and after running for an hour the code crashed and we could not collect any results for this experiment.

3.2.2 With FlexGen:

We then profiled the FlexGen code given by the authors of the paper. Since their implementation had a lot of different options and settings (batch size, prompt length, overlapping I/O, etc.), we made sure to tune it with the right settings so that it would be close to our implementation for comparison purposes. We ran FlexGen on OPT-1.3b model and with the same settings as mentioned previously, and the code finished execution in 1 minute 40 seconds.

Time %			
Python	Native	System	Code
< 1	59	11	item = queue.get()
< 1	< 1	< 1	F.embedding()
<1	<1	<1	F.linear()
< 1	< 1	<1	torch.bmm()

Figure 7: Showing top 4 results of Scalene Profiler for FlexGen

From Fig.7, we observe that the code executes quite efficiently. queue.get() is part of the multiprocessor class and is used to communicate safey between threads. Since most of the time is in native code, this code is optimized and very little further optimizations can be done.

3.2.3 Our Implementation of FlexGen:

Here we show our profiler results for our implementation of FlexGen. The code finishes execution in 7 minutes 20 seconds. There is a sizeable difference in total execution time when compared to the original FlexGen code. In Fig. 8, we observe that aten::mm (responsible for matrix multiplication) and aten::copy_(responsible for tensor copy operations) take the most amount of time. These operations are computationally expensive and are not fully optimized to be run on the CPU compared to the GPU. In Fig. 9, we notice that 55% of our execution time is spent in native code for nn.multiheadattention() function. Whereas about 23% of the execution time is spent in system time i.e, I/O movement. We observe such a high cost in I/O movement since we store the KV cache on the disk due to CPU and GPU memory constraints. This movement of KV cache from disk to CPU incurs I/O cost. On the other hand, it is very difficult for us to explain why there is a huge cost in time when computing MHA. Since most of the time for that function is spent in native code and that is the code provided for MHA so it must be fully optimized. We were not able to find any reasonable explanation as to why MHA computations were taking such a large amount of time even after spending quite a bit of time going through our code. One possible explanation is that due to us not using any threads for parallelization, the scheduler of the CPU has to execute each of the various tasks sequentially. This includes movement of the tensors from the disk to the CPU. And since MHA function cannot proceed without the tensors on the same device, it has to wait a significant amount of time for the tensors to be on the same device to begin computation.

Name	Self-CPU
aten::mm	287.556s
aten::copy_	45.882s
cudaMemcpyAsync	21.793s
aten::cat	11.793s

Figure 8: Showing top 4 results of Pytorch Profiler for our version of FlexGen

Time %			
Python	Native	System	Code
< 1	55	16	mha() During Decoding
3	< 1	7	v.to(device)
< 1	7	3	mha() During Prefill
< 1	< 1	< 1	torch.cat()

Figure 9: Showing top 4 results of Scalene Profiler for our version of FlexGen

4 Limitations/Future Work

4.1 Limitations

4.1.1 Basic Offloading and Scheduling

FlexGen's use of a linear programming-based search algorithm for optimizing offloading strategies allows it to efficiently manage memory and computation resources across the memory hierarchy. Our current version uses a more straightforward manual offloading strategy without the advanced linear programming optimization, leading to less optimal tensor management and higher I/O costs.

4.1.2 No compression implementation

FlexGen possesses the ability to compress both weights and the KV cache to 4 bits which effectively reduces memory demands and I/O bandwidth. Our initial development focused on achieving stable operation and correctness, with optimizations planned for later phases. We tried experimenting compression with the python library bitsandbytes[3], however we saw that out throughput decreased, and due to time constraints could not refine the experiment.

4.1.3 Low Throughput with Large Batch Sizes

FlexGen's capability to handle large effective batch sizes due to its sophisticated memory management and scheduling strategies is critical for high throughput. Limited access to advanced optimization tools and hardware resources restricted us from experimenting with large batch sizes and models.

4.2 Future Directions

- Parallel Processing / Overlapping: Introducing overlapping of loading, unloading, and computing operations using custom CUDA Streams and

CPU Threads. Due to time constraints and limited knowledge

- Compression Improvements: Implementing compression techniques for weights and KV caches.
- Scalability: Extending FlexGen's capabilities to support multi-GPU environments.
- Optimization of Multi-Head Attention (MHA): Refining the cost and efficiency of the MHA component.

5 Conclusions

In conclusion, Flexgen provides us with on optimized way to run highly batched high throughput inference jobs. Through this project of re-implementing the FlexGen paper, we have gained valuable skills and a higher level of understanding of systems and large language models, especially for production model serving and inference pipelines. Working with FlexGen helped us understand how a transformer block is executed/-computed at a lower level, and by cleverly changing the scheduling pattern, we could get high increase in throughput by storing the KV caches instead of recomputing them when required without encountering that huge I/O cost. As discussed in our challenges and limitations, we certainly could not implement every single feature and optimization that came with the original FlexGen (Quantization, overlapping, linear programming, etc), we did have the opportunity to read their code-base [1] closely and fully understand how these optimizations are implemented and tied in together. The knowledge gained while working on this project will be beneficial for us in our future industrial and research work pertaining to generative AI or data related jobs. This project also gave us an idea on how teams operate in a project oriented environment, which is a very valuable skill for the future. All in all, we believe the FLexGen has high potential, and we are excited in seeing how the industry of GenAI continue to progress and projects like FlexGen to keep improving on traditional approaches to give us better results and most importantly help reduce the carbon footprint.

References

[1] GitHub - FMInference/FlexGen: Running large language models on a single GPU for throughput-oriented scenarios. — github.com. `https://github.com/FMInference/FlexGen`. [Accessed 17-05-2024].

[2] GitHub - plasma-umass/scalene: Scalene: a high-performance, high-precision CPU, GPU, and memory profiler for Python with AI-powered optimization proposals — github.com. `https://github.com/plasma-umass/scalene`. [Accessed 17-05-2024].

[3] GitHub - TimDettmers/bitsandbytes: Accessible large language models via k-bit quantization for PyTorch. — github.com. `https://github.com/TimDettmers/bitsandbytes`. [Accessed 17-05-2024].

[4] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.