Hugo Latendresse
Matthew Katz

Parallelizing Mixture of Experts Transformers in FlexFlow for Low-Latency Inference

15-618 Final Project

Final Report

Project Website
https://hugolatendresse.github.io/15618-final-project/

Code Repository
https://github.com/hugolatendresse/FlexFlow/tree/15618_project

Summary

We added support for serving Mixture of Experts-based Large Language Models in the FlexFlow framework on one GPU. We benchmarked the per-token latency of serving an MoE model using our implementation in FlexFlow against Hugging Face Transformers, achieving similar performance. We also profiled our implementation and compared the runtime of the operators we implemented to that of operators already implemented in FlexFlow. All development and testing was done on AWS G-type EC2 instances with L4 Nvidia GPUs.

Background

Mixture of Experts

Transformer-based Large Language Models (LLMs) have dominated the Artificial Intelligence (AI) landscape in recent years. Since the introduction of ChatGPT in 2022, LLMs have been widely adopted by the tech industry for a variety of purposes, such as streamlining workflows, generating code, performing data analysis, and implementing customer-facing chatbots. [1]

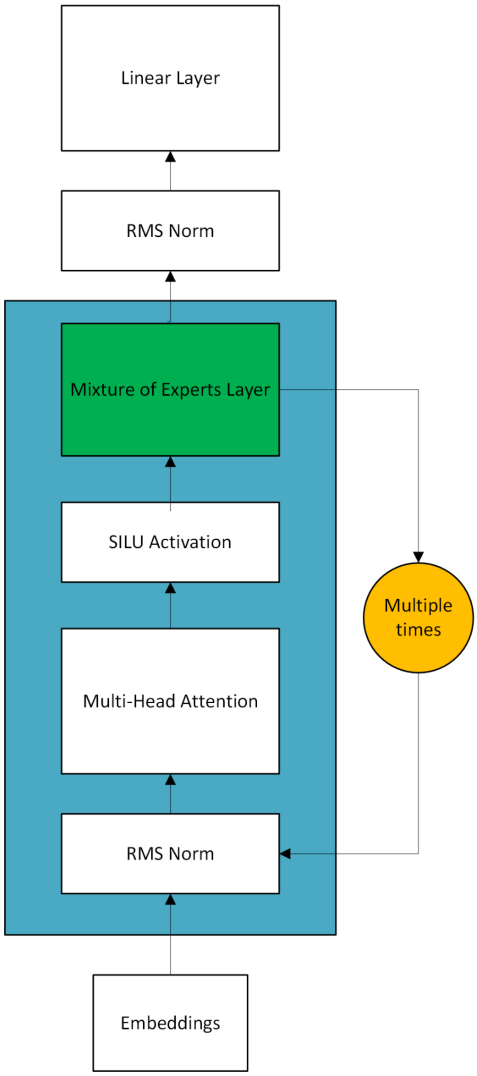
Perhaps the most salient feature of LLMs is their scale. The largest version of LLaMA 3, a popular open source model introduced by Meta, contains more than 400 billion parameters, representing more than a terabyte of data at full precision. It was trained on over 15 trillion tokens – several petabytes of data. Clearly, training and inferring with state-of-the-art models requires immense computational capabilities. [2]

Mixture of Experts (MoE) has become a popular method to increase model capacity of LLMs. The key innovation is to replace the dense multi-layer perceptron (MLP) within each block with the MoE mechanism. The MoE mechanism contains a set of experts, each of which is a MLP. Each token in a given sequence is assigned to a subset of those experts by a linear layer called “gate”. This means that performing a forward pass for a particular token input entails using only a subset of model parameters. See diagrams on the next pages for a description of the model architecture and MoE layer.

Experts have no data dependencies between themselves and so they can process their inputs in parallel. The increased parallelism of MoE can lead to less costly parameter scaling and lower per-token latency for training and inference.

Mixtral

In our project, we focus on the MoE implementation introduced by Mistral.AI in their Mixtral 8x7B model, a decoder-only Transformer model that popularized the use of MoE in LLMs [3]. We followed the corresponding “MixtralForCausalLM” Hugging Face architecture.as specified in the modeling_mixtral.py script of the Hugging Face Transformers library.

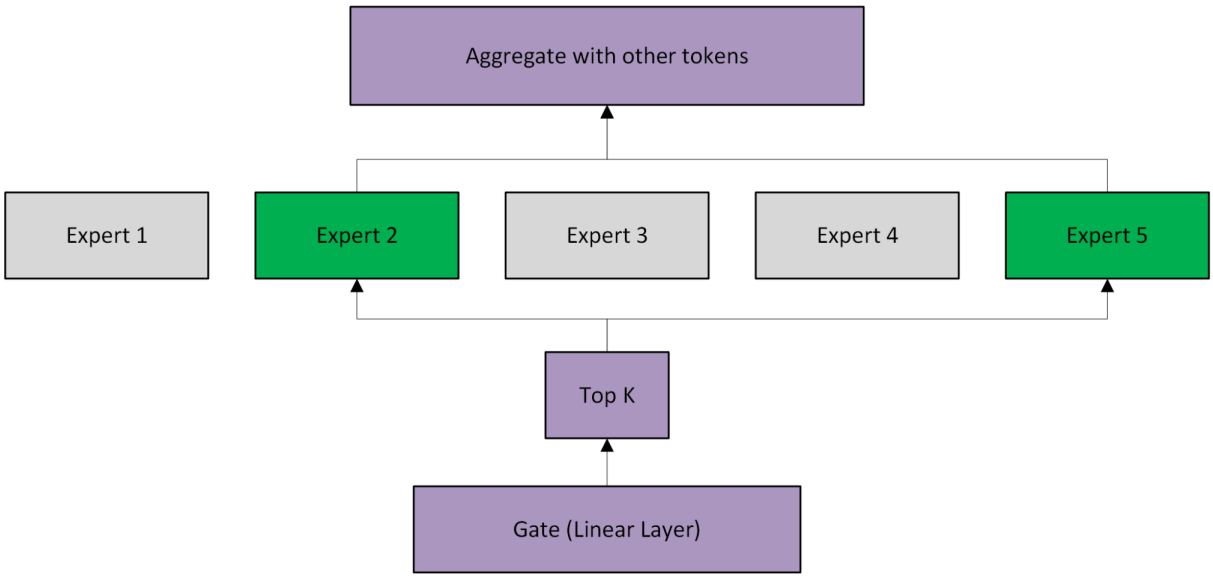


Mixtral replaces the Feed Forward Network (FFN) that typically comes after the Self-Attention module in Transformer architectures with a MoE block composed of *n* FFN experts. The MoE block begins with a Gate, a simple linear function that can be interpreted as outputting similarity scores between each input token and expert. A TopK function extracts the indices of the top *k* scores for each token as well as the top *k* scores themselves. These indices identify the experts to which the token will be sent. Copies of the tokens are then grouped by their corresponding expert indices and sent to be processed by each expert.

In Mixtral, each expert consists of a SwiGLU block. In simple terms, the block consists of four operations. First, two linear functions are applied to the input in parallel. Second, a SiLU activation function is applied to the output of the first linear function. Third, the activations are multiplied (element-wise) with the output of the second linear function. Last, another linear function is applied to the output.

The outputs of each expert are weighted by the score calculated earlier (using only the top *k* for each token), and aggregated back to create a tensor the same shape as the original input to the MoE block.

Below is a diagram of the MoE mechanism for a given token in a sequence.



Mixtral is known as a Sparse MoE model because it has $n = 8$ experts, but sends each token to only the top $k = 2$ of them. Processing each token therefore only utilizes a small fraction of the MoE block's parameters.

Parallelism

There are several popular methods to parallelize serving LLMs for inference that can be applied to Mixtral.

Practically all large deep learning models require the use of hardware specialized for massively parallel processing for inference performance to be reasonably efficient. In general, these constraints can be satisfied by a GPU(s) with sufficient VRAM capacity. In LLMs, all computation typically occurs on the GPU(s) and operations are defined by specialized low-level code like CUDA kernels. When using only one GPU, each model operation (linear layer, SiLU, etc.) of fused operation will be launched one at a time on the GPU and will be parallelized across the GPU cores.

Mixtral 8x7B has approximately 45 billion parameters, which equates to about 180 GB at full precision and 90 GB in half precision. To serve a model of that size, we need access to multiple GPUs and to split the model parameters between them. There are multiple possible forms of parallelism across GPUs. In tensor parallelism, embeddings and/or weight tensors are split and allocated on separate devices. Each device performs the same computations, and an all-gather or all-reduce is performed at the end of every operation to assemble the output of the computation. In pipeline parallelism, entire layers of the model are split across devices. Outputs are sent between the devices for the next stage of computation. Since we are working on

inference and benchmarking with respect to one short prompt, we do not focus on data parallelism. [4]

The MoE layer also presents a unique opportunity to exploit parallelism. Since there are no data dependencies between each expert, each expert can be entirely hosted by a single device, such that multiple experts run concurrently on different devices. This is known as expert parallelism.

FlexFlow Serve

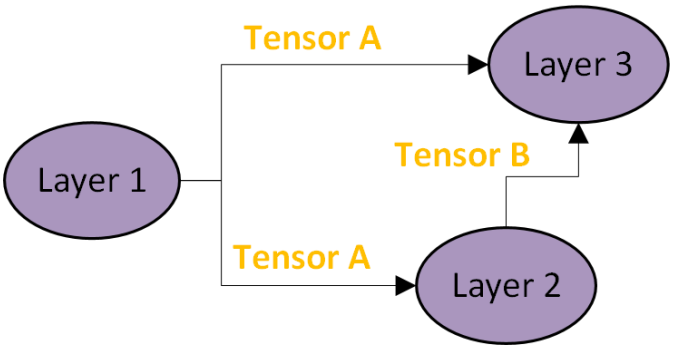
FlexFlow Serve is an open-source distributed framework for low-latency LLM inference, developed and maintained by Zhihao Jia and his research team at Carnegie Mellon University. It employs two key innovations to increase performance. First, it performs graph-level optimizations on the computational graphs to dynamically fuse operations together and to search for the best parallelization strategy given the user's hardware topology [5]. Second, it employs speculative inference to decrease per-token generation latency [6]. However, our project focused on incremental decoding, which is also supported by FlexFlow.

The backend of the portion of FlexFlow Serve that we worked with is written in C++ and CUDA. It also includes a Python interface to serve LLMs.

It is worth noting that FlexFlow also relies on the Legion framework for memory management and data movement patterns. Legion is a programming system designed to standardize parallel processing on heterogeneous architectures. It comes with its own set of abstractions. [7]

Key Abstractions

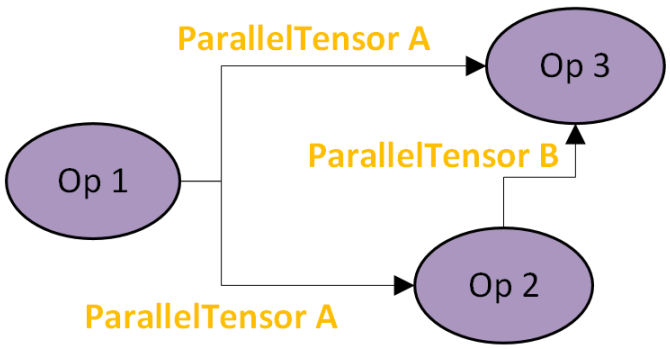
To add inference support for Mixtral in FlexFlow Serve, there were several key abstractions that we had to implement and debug. Please note that this is a vast simplification of the FlexFlow API and framework, which is highly complex.



At the highest level of abstraction, a model consists of an `FFModel` defined by a sequence of `Layer` objects. These `Layer` objects each represent an operation take in one or more `Tensor` objects as well as parameters relevant to the underlying

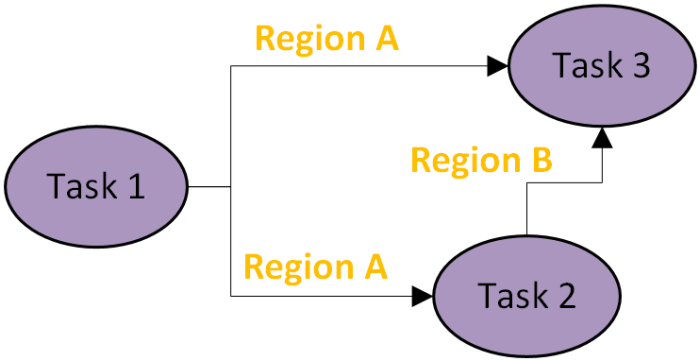
operation performed – such as the data type of the `Tensor`, whether there is any activation function to be applied after the computation, the shape of the output `Tensor`, and so on. While `Layers` coincide with the layers of a model architecture, they do not define computation in themselves and do not directly handle data or memory. Their primary use is to define dependencies between the different stages of the model, which is implicit within the movement patterns of their input and output `Tensors`. The `Layers` are connected to one another through a computational graph in which the edges are `Tensors`.

A `Tensor` is a logical abstraction of an n-dimensional array that also does not contain any underlying data itself. It contains information about dimensionality, initialization, its layer of origin. It can correspond to inputs, weights, or outputs.



Each `Layer` is associated with an `Op`. These are logical abstractions of the underlying computation as well, and are similar to `Layers` except that they handle `ParallelTensors` instead of `Tensors`. The two are similar in that they contain high-level information about the underlying data but do not point to the data itself. `ParallelTensors` also contain information about how the data can be parallelized (e.g. along which dimensions the tensor can be split). The `Ops` are connected to one another through a computational graph in which the edges are `ParallelTensors`. Each `Op` has several methods corresponding to different modes of computation. Typically, there is one defined for a forward pass, a backward pass, and an inference pass.

Each input and output is assigned a Legion `Region`, a logical abstraction for a contiguous chunk of data.



Each mode-specific method in an `Op` corresponds to a `Task` to be scheduled by the runtime. Within each `Task`, we obtain accessors to the memory allocated for the data

to be used in computation and to the memory allocated for the outputs. The `Task` calls a wrapper to a CUDA kernel which actually defines the computation to be performed. The dependencies between them are conveyed through the different regions of memory they access.

Model Compilation

Once a model is defined, it is compiled and optimized before actual execution begins. The core of model compilation consists of code validation and computational graph optimization. The code at all levels of abstraction is first checked for inconsistencies and errors. Then, each layer of the model is converted into operators and a parallel computation graph is obtained from the data dependencies between them.

The graph is then optimized for performance. Operators may be fused together and reordered from the sequential order implicit in the sequence of the `Layers`. Since `ParallelTensor` includes information about how the underlying data is to be parallelized, replicated, and mapped to the underlying hardware, the optimization step can also map data and computation to different hardware devices and determine the optimal parallelization strategy.

The inputs and outputs of the operators in the optimized graph are then checked for correctness (whether the shapes are valid and consistent, for example), and the relevant n-dimensional arrays are allocated in memory. It is worth noting that correctness is checked at multiple different stages in the compilation process.

Parallelism

Flexflow supports several different types and combinations of parallelism. First, since virtually all computation is performed on GPU, it always utilizes SIMD parallelism at a massive scale. It also supports multi-GPU parallelism such as data parallelism, tensor parallelism, and pipeline parallelism. The graph optimization step in model compilation also may leverage multiple types of parallelism, creating a kind of hybrid parallelism. The best parallelization strategy is determined through this optimization procedure, as explained above.

This parallelization information encoded within the `ParallelTensors` used by each operator must be properly implemented for FlexFlow to be able to perform parallelization across GPUs.

Approach

Technologies Used

The main language we used is C++, as the bulk of FlexFlow is built in C++. FlexFlow supports Nvidia and AMD GPUs, but we focused our project on Nvidia L4 GPUs. Hence, all of our kernels were written in CUDA. We tried multiple environments and ended up using AWS g4dn.4xlarge and g6.4xlarge EC2 instances. An important dependency of FlexFlow is Legion, a runtime developed at Stanford that abstracts memory management of both CPU and GPU in distributed environments. Other environments that we unsuccessfully tried using include PSC (with GPU), our own laptop (Ubuntu 24.04 with Nvidia Quadro T2000), CMU's GHC machines, Google Colab, and P instances on AWS.

For development purposes, we used the TinyMistral-6x248 model by M4-ai. It has the same architecture as Mixtral, but is smaller and fits on one GPU. Below is a comparison.

	TinyMistral-6x248	Mixtral 8x-7B
Number of Blocks	12	32
Number of Experts	6	8
Embedding Dimension	1024	4096
Expert Hidden Dimension	4096	14336

Starting Point

Our starting point was the "inference" branch of FlexFlow as well as a draft pull request developed by Gabriele in March on a branch called "mixtral". In practice, the main goal of our project was to complete this pull request by solving the bugs present in such a work-in-progress.

The first two weeks of the project were spent trying to install and run FlexFlow (more details in the Results section). We were finally able to have a meeting with Gabriele on November 26, where he helped us complete the installation. The next two weeks were spent debugging the "mixtral" branch.

Parallelizing MoE on GPU

As described in the Background section, we had to make several different layers of abstraction and data structures work together to make our MoE model work. The MoE model uses the following operations:

Specific to MoE	Common to FlexFlow
Top K	Embedding
Group-By	RMS Norm
Aggregate	Residual RMS Norm
	Linear
	Multiquery Self-Attention
	Softmax
	SiLU

Model

We completed the mixtral.cc script that defines the model based on Tensors and Layers corresponding to the operations above. The operations in the right-hand-side column were all stable in our starter code, while we had to complete and debug the implementation of the left column. For each operation, we had to define a Layer based on Tensors, an Op based on ParallelTensors, and an initialization and inference Tasks based on Legion regions. At the model level, the architecture is implemented with each of those operations being sequential. Parallelization occurs because each inference task is ultimately centered around a CUDA kernel that runs on GPU. Inference tasks achieve that by passing pointers corresponding to Legion regions when launching kernels. (More specifically, the FlexFlow compiler creates a graph of all tasks and performs some optimizations, but this did not affect your work directly).

Below is a description of the three operations used in the MoE model that are specific to MoE (the experts are based on operations that were already stable in FlexFlow, Linear and SiLU). Under Inputs and Outputs, the parentheses indicate the dimensions of the tensors.

TopK

Description

TopK selects which *k* experts have the highest value for each token in the output of the “gate” linear layer.

Inputs

“Gate” (number of experts x numbers of batches x sequence length), which represents the score of each token with respect to each expert

Outputs

“Topk_values” (experts per token x number of batches x sequence length), which represents the weight of each selected expert for each token.
“Topk_indices” (experts per token x number of batches x sequence length), which represents the index of each selected expert on each token.

Parallelization

Each CUDA block processes one sequence (one element in the batch). Each CUDA thread processes a shard of the sequence. Tokens are assigned to each shard in an interleaved way (thread k processes tokens k, thread_count+k, 2*thread_count+k, etc.).

Use of Memory

Inputs and outputs are stored in global memory. A heap stored in shared memory is used to determine the top experts for each token.

Group-By

Description

Group-By creates one array for each expert, where each array contains only the tokens relevant to that expert.

Inputs

The tensor (hidden dimension x batch x sequence) coming out of multi-head attention.
“Topk_indices” (experts per token x number of batches x sequence length) from the Gate, which represents the index of each selected expert for each token.

Outputs

An array of tensors, where each tensor is (hidden_dimension x batch x max token count per expert).

Parallelization

The multiple outputs are seen as one large 1D array. This one array is divided across the CUDA blocks. In each block, one thread first creates a lookup table called chosen_exp_preds to map each token to each output. Next, the tokens are divided among and processed by each thread.

Use of Memory

The chosen_exp_preds mapping described above lives in shared memory. Everything else is in global memory.

Aggregate

Description

After each expert computes a new representation of its assigned tokens, the Aggregate operation calculates the final representation of each token as the weighted average of the output of each of its experts (using topk_values as weights). The result for all tokens is concatenated into one tensor.

Inputs

“Topk_values” (experts per token x number of batches x sequence length), which represents the weight on each selected expert for each token.
“Topk_indices” (experts per token x number of batches x sequence length), which represents the index of each selected expert on each token.
The output of each expert, each of which is (hidden_dimension x batch x max token count per expert)

Outputs

A (hidden dimension x numbers of batches x sequence length) tensor containing the final representation of each token, to be passed to the next model layer.

Parallelization

Like in Group-By, each thread is assigned a subset of the tokens in the output, and a chosen_exp_preds array holds pointers to the locations in global memory where the chosen experts' outputs reside.

Use of Memory

The chosen_exp_preds is in shared memory, and everything else lives in global memory.

Other Approaches

We successfully outputted tokens with our selected MoE model (TinyMistral) on one GPU for the first time on December 10. Professor Jia agreed that outputting on only one GPU was a reasonable project scope considering the difficulty of working with a large system. Since we had a few more days to try additional approaches, we tried the following:

1. **Tensor Parallelism.** That requires specifying in each operation how to split ParallelTensors across GPUs, and ensuring that the underlying code (tasks and kernels) still work. As we were short on time, we were not able to successfully make everything work. As of our last commit, we are getting tensor dimension validation errors.
2. **Fused Experts**. In our implementation, each expert is a series of independent tasks, and the group of experts is a sequential for-loop where each iteration corresponds to one expert. We tried to optimize this by first creating a fused operation corresponding to each expert, with the goal of being able to launch each expert concurrently in one MoE-layer operation instead of sequentially. This required a lot of additional implementation, and the fused expert is still a work-in-progress.

Expert parallelism was the first item in the list of “Hope to Achieve” goals for this project, but we did not get to it.

Results

Our primary result is that we were able to finally run, benchmark, and profile inference using incremental decoding on an MoE model with FlexFlow. This was the result of dozens of hours of debugging obscure (at least to us!) error messages. Our process took us on a journey through many key components of FlexFlow – the runtime, all the layers of abstraction discussed previously, parallel computation graph optimization, CUDA kernels, and memory management through Legion.

Our project became an exercise in debugging a high performance computing system with minimal guidance. Moreover, as mentioned previously, we only had two weeks between successful installation and the poster session to familiarize ourselves with the inner workings of FlexFlow and get our implementation working.

We were able to achieve all the “Plan to Achieve” goals on our list, except for the fact that our final implementation lacks correctness. A bug somewhere causes the generated tokens to be wrong. We began the debugging process through running alignment tests - validating the input and output tensors of each operator against those

of a correct implementation such as HuggingFace. We were able to pinpoint that the issue lies in the faulty implementation of the GroupBy kernel. However, due to time constraints, we did not get around to fixing it.

Installation Journey

We tried to install FlexFlow using different methods, on different platforms, and tested builds using different models. We followed the installation documentation as best as we could but ran into difficulties each time. We based this section off of our project milestone report.

We tried three installation methods: pip install, build from source, and build while running the FlexFlow docker container. We tried first locally on one of our ThinkPad machines running Ubuntu with an Nvidia GPU. We ran into difficulties installing and initializing the CUDA drivers and getting the FlexFlow installation process to recognize them.

We then tried the same with the PSC machines, which already have CUDA support. We tried all three methods again, but each time we ran into different dependency issues for each installation method. Upon trying to manually install missing dependencies, new issues would arise. Moreover, it seemed that FlexFlow did not recognize the CUDA installation on the PSC machines even after manually loading the CUDA module and setting the corresponding environment variables pointing to the CUDA driver and library locations. We tried several different ways to configure the dependencies, with and without the use of Singularity. We tried several approaches to fix the issues we were facing by looking at the Bridges-2 documentation and the Singularity documentation.

We then decided to spin up our own (GPU-enabled) AWS instances. We initially ran into the same type of dependency issues we experienced locally and with the PSC. After trying repeatedly on P-type instances, we finally were able to complete the installation by building a docker image on a G-type instance and run a Python training example. However, C++ is preferable for development (for debugging), so we needed to ensure that the C++ inference examples worked. We ran into obscure error messages when we tried to run the inference examples with a LLaMA model, which is currently supported by FlexFlow.

The installation issues cost us more than two weeks of progress. In the end, we were able to schedule a meeting with Gabriele on November 26th. We spent more than an hour troubleshooting the installation process with him. With his help, we were able to get the LLaMA-3-1B C++ inference example working on a g6.4xlarge EC2 instance on AWS.

Debugging Journey

We started with a draft implementation of Hugging Face's MixtralForCausalLM in FlexFlow written by Gabriele. We fixed bugs until we got it working on one GPU. This was difficult for the following reasons.

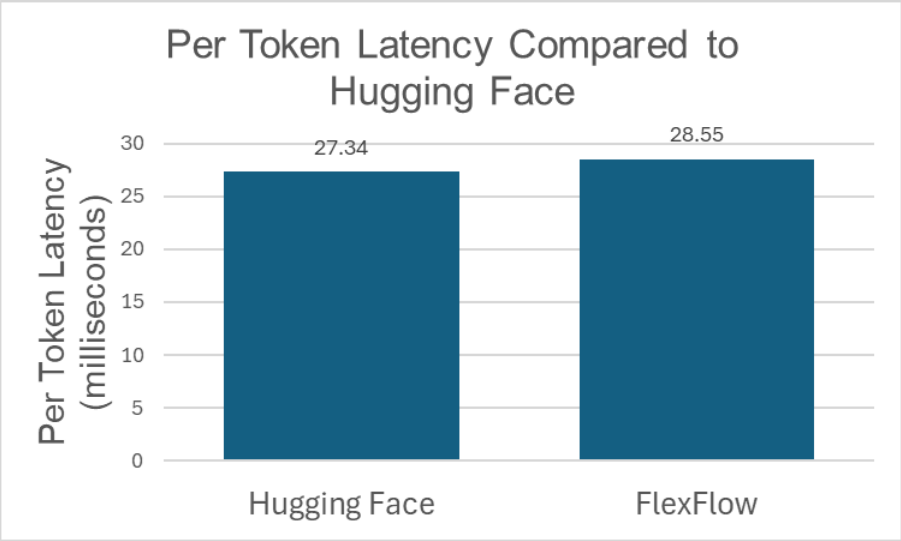
- The sheer complexity of FlexFlow. There were issues at all levels of abstraction, and to solve them we had to take the time to understand many key components of FlexFlow.
- There is essentially no documentation regarding how the code works, inside or outside the repository, so we had to understand what the code is doing by reading and understanding a lot of lines of code.
- The errors arose at all stages of compilation and levels of abstraction, including the Legion runtime and graph optimization step.
- The error messages, especially those concerning memory management from Legion, were obscure to us.
- In the two weeks we had to debug the "mixtral" branch, we were only able to meet with Gabriele twice.

We rarely arrived at a point where we were literally blocked. Every bug was investigable, and we rarely ran out of ideas on how to debug. It's just that the many moving parts made the process of understanding each issue relatively slow, and we had very limited time. When we arrived at a point where we were blocked, we emailed Gabriele or Professor Jia. On the other hand, we avoided sending emails for questions for which we knew a (albeit slow) way to obtain the answer.

Benchmarking

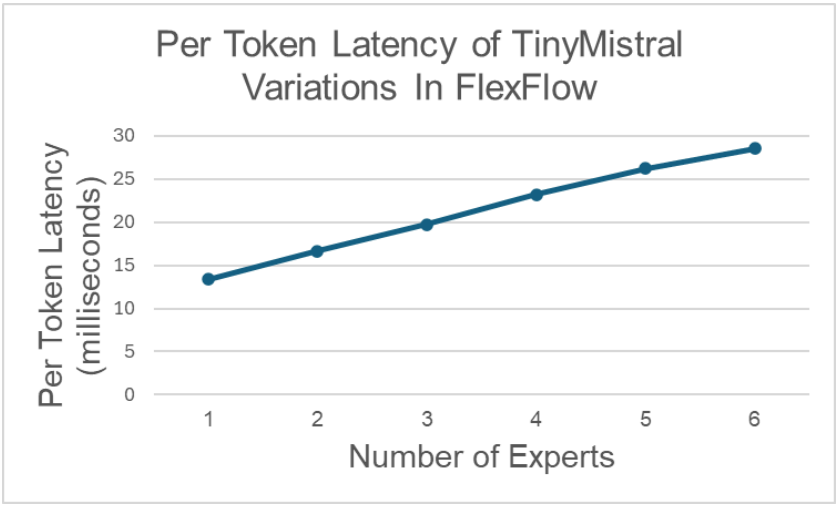
We benchmarked the per-token generation latency of the M4-ai/TinyMistral-6x248M model in our FlexFlow implementation against the Hugging Face Transformers library [8]. The test was performed using the single GPU (Nvidia L4) AWS g6.4xlarge EC2 instance.

We used an identical generation configuration for both settings. We generate 128 tokens at maximum based on the prompt "Three tips for staying healthy are: ", using simple argmax sampling.



Our implementation achieved similar performance to that of Hugging Face. The lack of performance gains is unsurprising considering the sequential nature of our experts. Our implementation could be optimized with the different approaches previously discussed.

It is also important to note that we used the Python interface for Hugging Face and the C++ incremental decoding interface for FlexFlow. The overhead incurred by Python may have increased the latency of Hugging Face, so it is possible that our implementation also performs even worse in comparison. We did not have the time to develop a more equitable benchmarking setting.



We also measured the per-token generation latency of including different numbers of experts, which is charted above. Because we were only able to implement a sequential approach to expert computation, the latency increases essentially linearly (with a

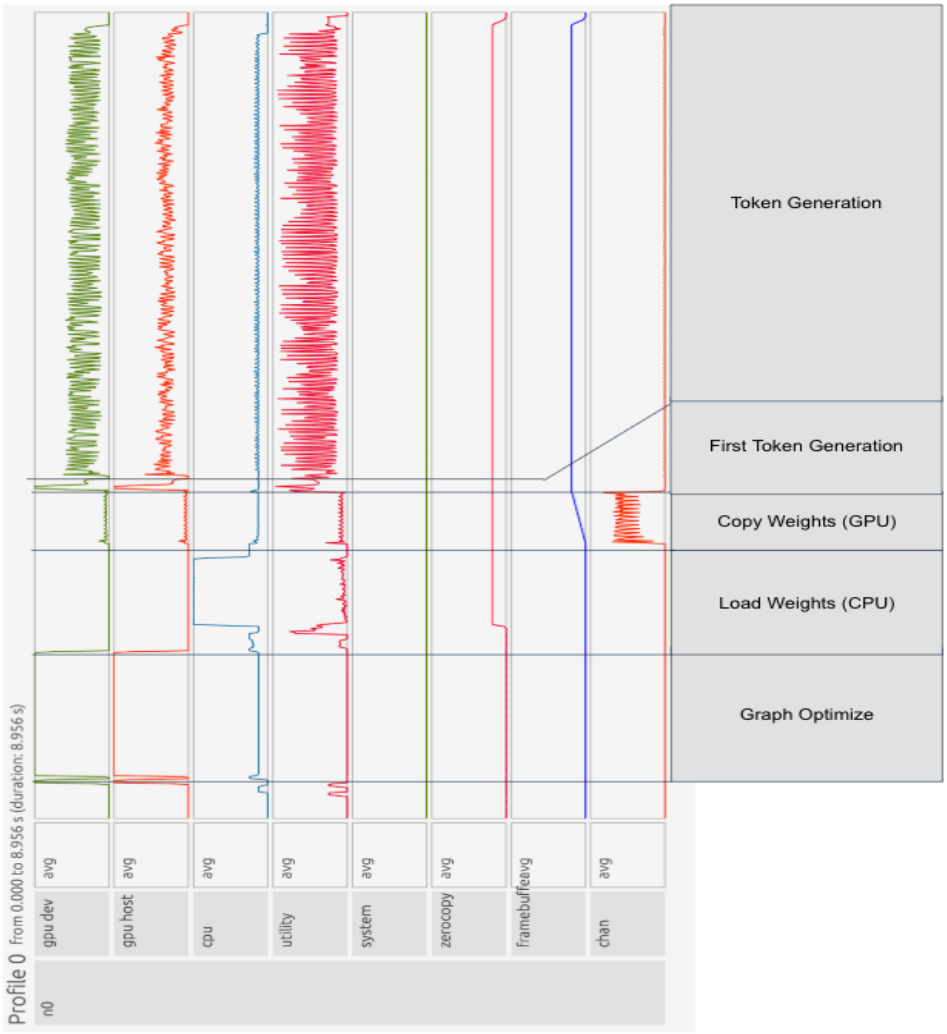
non-zero intercept) with each additional expert. Ideally, with expert fusion or parallelism, the latency would scale better with the number of experts.

Profiling

We profiled the inference performance of the TinyMistral-6x248M model using the same setup as the benchmarking tests discussed previously, using Legion's profiling software.

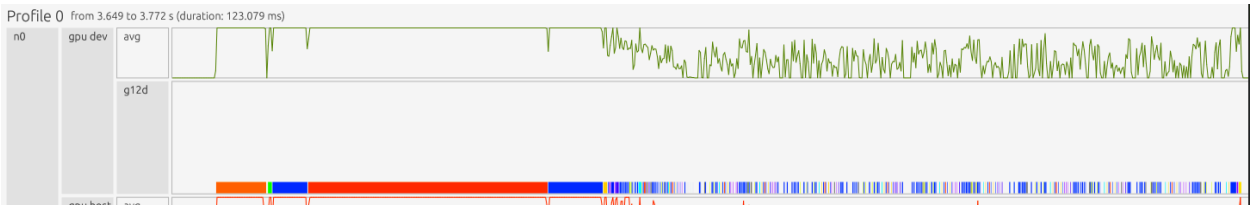
Full Profile

The full profile, annotated with high-level distinct stages of model serving in FlexFlow, is on the next page. The height of each line represents percent utilization on each physical (or logical) device.



First Token Generation Profile

GPU utilization (gpu dev) is at 100% for the generation of the first token, but it oscillates in a mid-level utilization range for subsequent token generation steps. We think that this is the result of GPU warm-up. In addition we found that the multi-head attention operator (the red bar in the following profile) has extraordinarily large runtime in comparison to the others. This can be explained by the implementation of a KV cache that amortizes the quadratic complexity of attention by caching attention scores at each token-step. There is substantially more computation performed in the generation of the first token than the subsequent ones.



MoE Block Runtime Analysis and Performance Limitations

We also analyzed the runtime of each operator used in the MoE mechanism, as you can see below. We obtained this information from the profile of the MoE block on the following page. The table is sorted by the last column.

MoE Operators	**Average runtime per call (ms)**	**Cummulative runtime per MoE block (ms)**	**Cummulative runtime per entire inference pass (ms)**
Linear	0.027	0.513	6.156
SiLU	0.024	0.149	1.788
Aggregate	0.043	0.043	0.516
GroupBy	0.036	0.036	0.432
Softmax	0.021	0.021	0.252
TopK	0.018	0.018	0.216

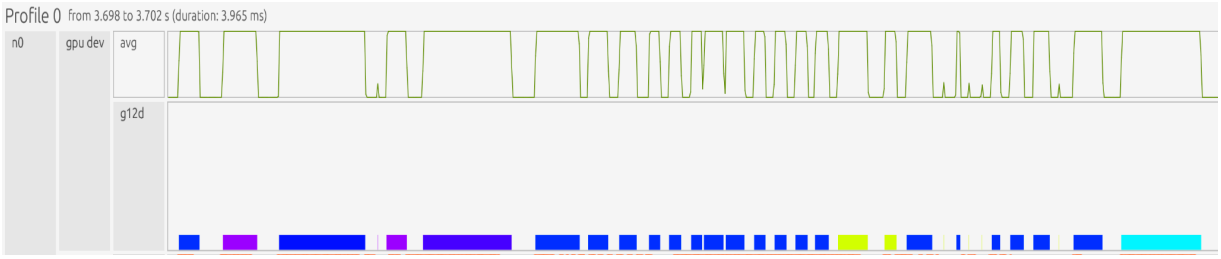
We averaged the runtime per operator call for the operators present in the MoE block, and extrapolated the average runtime to cumulative runtimes per MoE block and entire forward passes (excluding non-MoE related invocations).

The last column above shows that the bulk of the run time is spent on Linear and SiLU, which are already stable in FlexFlow. Linear and SiLU are used by each expert. This confirms the value of the possible improvements we previously discussed: fusing each expert, and parallelizing the experts across GPUs (expert parallelism). Our main performance limitation was to not make use of multiple GPUs.

The “Average runtime per call” column shows that Aggregate and GroupBy have the highest runtime per operation. We think this is due to the fact that they have multiple inputs and outputs (respectively), leading to additional memory access to global memory, which is expensive.

Below is the profile obtained from a single run of the MoE block. The green line represents GPU utilization, and the blocks below represent the execution of different operators. The following is a map from color to operator (apologies that the colors are similar for many operators, this is the direct output of the profiling tool):

- Dark blue (first): TopK
- Purple (second): Softmax
- Violet (fifth): GroupBy
- Dark blue (sixth): Linear
- Light Green: SiLU
- Light Blue (last): Aggregate



Work Distribution

Hugo	**Matt**
Installation: successful installation of Python interface on AWS + experimentation with different EC2 instances + unsuccessful attempt on personal laptop, Colab, and PSC with and without Singularity	Unsuccessful attempts to install FlexFlow on PSC and AWS EC2 instances
Found way to have persistent docker containers on EC2	Research on MoE and expert parallelism
Found an MoE model we could successfully run on one GPU (TinyMistral)	
Debugging: ● Weight file loading ● Config parsing ● Tokenization ● Full precision vs half precision ● MHA with small head size ● Register Divide ● GroupBy (Layer/Ops/Tasks) ● Aggregate (all methods) ● Extra dimension in ParallelTensors ● Add layer guid to ops	Debugging: ● GroupBy (Op/Layer/Tasks/kernel) ● Reduce_sum (Op) ● Divide (Op) ● Add layer guid to ops ● num_tokens_per_expert threshold ● Enabled full number of experts ● Softmax dims ● Fixed nullptrs for aggregate input
	Profiling and benchmarking
Meetings with Prof Jia and Gabriele	Meetings with Prof Jia and Gabriele
Debugging multi-GPU parallelism	
Started expert fusion	

Distribution of Total Credit: 55% Hugo, 45% Matt

Codebase

Codebase for the project: https://github.com/hugolatendresse/FlexFlow

Final branch: 15618_project

Branch for work in-progress for fused experts: fused_expert

Resources

1. Cai, W., Jiang, J., Wang, F., Tang, J., Kim, S., & Huang, J. (2024). A Survey on Mixture of Experts. *arXiv*. https://doi.org/arXiv:2407.06204
2. Meta. (2024, April 18). *Introducing Meta Llama 3: The most capable openly available LLM to date*. AI at Meta. https://ai.meta.com/blog/meta-llama-3/
3. Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. de las, Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., … Sayed, W. E. (2024). *Mixtral of Experts.* arXiv. https://doi.org/10.48550/arXiv.2401.04088
4. HuggingFace. (n.d.). Mixtral. https://huggingface.co/docs/transformers/en/model_doc/mixtral
5. Colin Unger and Zhihao Jia and Wei Wu and Sina Lin and Mandeep Baines and Carlos Efrain Quintero Narvaez and Vinay Ramakrishnaiah and Nirmal Prajapati and Pat McCormick and Jamaludin Mohd-Yusof and Xi Luo and Dheevatsa Mudigere and Jongsoo Park and Misha Smelyanskiy and Alex Aiken. Unity: Accelerating {DNN} Training Through Joint Optimization of Algebraic Transformations and Parallelization. 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 2022, Carlsbad, CA, 267-284. https://www.usenix.org/conference/osdi22/presentation/unger
6. Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), Vol. 3. Association for Computing Machinery, New York, NY, USA, 932–949. https://doi.org/10.1145/3620666.3651335
7. Stanford. (n.d.). *Legion overview*. Legion Programming System. https://legion.stanford.edu/overview/index.html
8. Huggingface, M4-ai. (n.d.). https://huggingface.co/M4-ai/TinyMistral-6x248M