Benchmarking Large Language Model Inference

Narain Pattabhiraman
email: npattab1@asu.edu

Satwik Ponnam
email: sponnam@asu.edu

Harish Karthik Kumaran Pillai
email: hkumaran@asu.edu

Abstract

The landscape for training deep learning models (LLMs) has grown markedly intricate and demanding, owing to the rapid expansion in both size and complexity of these models. The substantial computational requirements necessary for processing extensive datasets have emerged as a formidable challenge in the realm of large language models. Our project was driven by a keen interest in addressing this challenge through the integration of parallel computing techniques into our deep learning methodologies. Within this report, we embark on a comprehensive exploration of our approach, beginning with a thorough analysis of the underlying issue, followed by the formulation and implementation of a potential solution. We then proceed to evaluate the efficacy of our solution, while also identifying areas for potential enhancements and future research endeavors.

1. Literature Survey

1.1. Model parallel

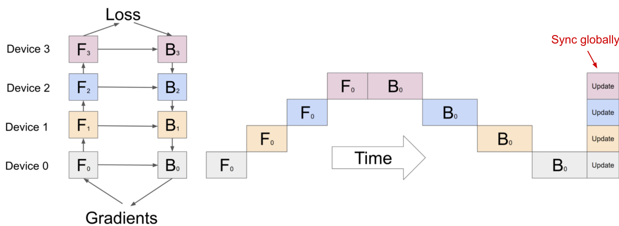

Figure 1. Computation flow of model parallelism across multiple GPU's

Model parallelism addresses scenarios where model weights exceed the capacity of a single node by distributing computation and parameters across multiple machines. Unlike data parallelism, which replicates the full model across workers, model parallelism assigns only a portion of the parameters to each worker, reducing both memory use and computational load.

Deep neural networks, typically composed of vertical layers, appear ideally suited for partitioning by layers, where a small set of consecutive layers is assigned to a single worker. However, a simplistic approach of processing data batches sequentially through these workers can lead to significant idle times and poor utilization of computational resources.

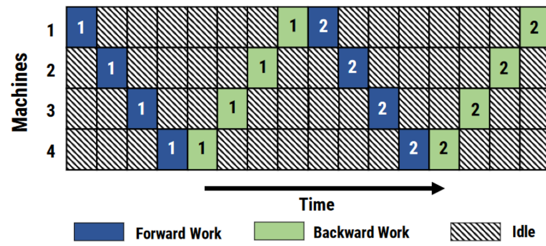

Figure 2. Computation heat map across devices

The model parallel approach, though effective in distributing work, tends to be slower due to the sequential processing of layers, which leads to underutilization of available computational units, as only one is active at any given time.

1.2. Pipeline parallelism

Pipeline parallelism combines data and model parallelism to reduce downtime between model computations. It does this by dividing a single minibatch into several microbatches, enabling each stage worker to process one microbatch at a time, which includes both a forward and backward pass. Communication between workers involves only the exchange of gradients and activations.

Different strategies are employed for scheduling tasks and aggregating gradients in pipeline parallelism, with the setup commonly referred to as pipeline depth. In the GPipe framework, gradients from several microbatches are combined and applied simultaneously, ensuring consistent and efficient learning across various workforce sizes. This method helps to significantly reduce idle time, known as "bubbles," especially when the number of microbatches is much greater than the number of partitions.

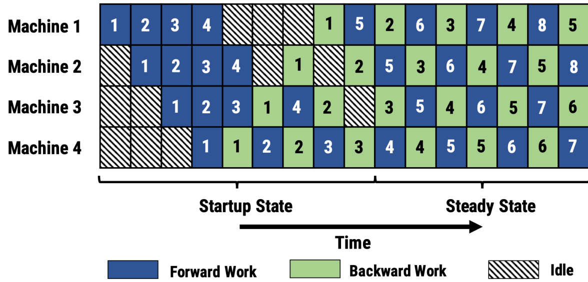

Figure 3. Computation heatmap of pipeline parallelism model

1.3. GPipe

The GPipe's throughput generally scales linearly with the addition of devices, although an uneven distribution of model parameters across workers can prevent this.

In PipeDream's approach, each worker alternates between the forward and backward passes of a model, referred to as "1F1B". Each part of the model is considered a "stage," and to enable data parallelism, each stage may include several replicas.

Workers manage different model versions to ensure that the same version is used for both passes of a minibatch. PipeDream also allows an optional "vertical sync" where the model weights are synchronized across stages along with activations and gradients, maintaining version consistency in an asynchronous manner, unlike GPipe.

1.4. Deep speed Zero

The substantial computational needs of large model inference require accelerators like GPUs to perform efficiently. A critical consideration for managing a limited GPU budget is the effective distribution of GPU memory across model weights, inference inputs, and intermediate results.

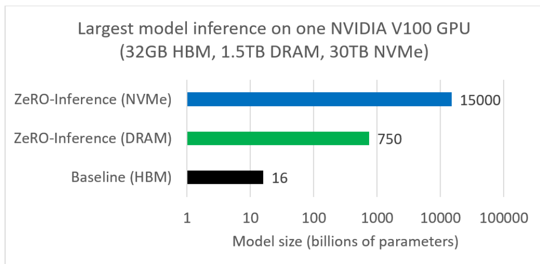

Figure 1:Maximum model sizes for inference on one V100 GPU when hosting the model on GPU memory (HBM), CPU memory (DRAM), or NVMe memory. Note that x-axis is log scale.

ZeRO-Inference adopts a strategy where it keeps all model weights on the CPU or NVMe, depending on which can hold the entire model. It then streams these weights into the GPU one layer at a time for inference processing. After each layer is computed, its outputs are saved in GPU memory for use in the next layer, while the memory used for the layer weights is freed up for the upcoming layer. This approach results in inference times that include both the computation time on the GPU and the time needed to transfer layers over PCIe.

Initially, PipeDream profiles each model layer to determine computation time and memory needs. It then uses dynamic programming to find the optimal way to partition the model into stages.

2. Method

In this study, we implemented model parallelism and tensor parallelism on a plain PyTorch framework, due to challenges experienced with frameworks such as Hugging Face. Specifically, our implementation of tensor parallelism, which involved partitioning weights according to attention head dimensions, yielded higher performance metrics in terms of tokens per second.

Our experiments were conducted on a single node equipped with up to four A100 GPUs within the SOL GPU cluster. The focus was on achieving an asynchronous inferencing mechanism akin to the PipeDream architecture, which required a specialized scheduler for effective weight management on the GPUs.

For inferencing, a batch size of four was utilized, and prompts were sourced from the Alpaca dataset. The findings from this study suggest that strategic weight distribution and parallel execution can significantly enhance the computational efficiency of large-scale language models.

3. Experimental Environments and Setup

Our experiments were conducted within a controlled laboratory environment utilizing the SOL GPU cluster. This cluster is equipped with single-node configurations, each node hosting up to four NVIDIA A100 GPUs, renowned for their high computational capabilities and efficiency in handling large datasets and complex neural network operations.

The experimental setup was configured with PyTorch, a popular deep learning framework known for its flexibility and ease of use. PyTorch facilitated the implementation of both model parallelism and tensor parallelism by allowing custom partitioning of model weights across GPUs. The partitioning strategy was specifically designed to optimize the attention head dimensions for tensor parallelism.

For our inferencing tests, the batch size was set at four to balance throughput with GPU memory constraints. We utilized prompts from the Alpaca dataset, which is well-suited for evaluating performance across different NLP model configurations. The dataset's diversity in text prompts

helped in assessing the robustness of our parallelism implementations under varied linguistic inputs.

4. Evaluation Results

Upon analyzing our implementation, it becomes evident that regardless of whether we employ llama 2 7b or 70b, the tensor parallel approach consistently outperforms its model parallel counterpart. This observation is particularly striking when considering specific metrics. For instance, when utilizing 4 GPUs, the llama 13b model parallel configuration achieves a throughput of 88 tokens per second.

In stark contrast, a comparable implementation employing tensor parallelism achieves a substantially higher throughput of 138 tokens per second. This notable discrepancy underscores the superior efficiency and effectiveness of the tensor parallel approach in our context. Such findings not only validate our decision to explore parallel computing techniques but also highlight the significant performance gains achievable through strategic implementation choices. As we delve deeper into our analysis, these results serve as crucial insights guiding our ongoing optimization efforts and shaping our future research directions.

	1gpu	2gpu	4gpu
Llama 2 7b model parallel	65 tokens/sec	52 tokens/sec	56 tokens/sec
Llama 2 7b tensor parallel	65 tokens/sec	159 tokens/sec	218 tokens/sec
Llama 13b Model parallel	60 tokens/sec	72 tokens/sec	88 tokens/sec
Llama 13b tensor parallel	60 tokens/sec	94 tokens/sec	138 tokens/sec
Llama 70b model parallel	#	6 tokens/sec	9 tokens/sec
Llama 70b tensor parallel	#	22 tokens/sec	38 tokens/sec

Figure 4. Results of Model and Tensor Parallelism Evaluation

4.1. Evaluation Graphs

In our endeavor to comprehensively assess performance, we've graphically represented the results of our bench marking tasks. The line graph in figure 2 provides a visual depiction of the tokens processed per second, with each line representing the number of GPU utilized in our experiments. This visualization offers a clear illustration of how throughput scales with increasing computational resources. Additionally, to underscore the performance disparity between the tensor parallel and model parallel approaches, we've incorporated a bar graph in figure 3. This graph serves as a succinct visual aid, effectively showcasing the notable performance gain achieved through the implementation of tensor parallelism compared to its model parallel counterpart. By combining these graphical representations, we gain deeper insights into the relative efficiencies of different parallel computing strategies, thereby informing our decision-making process and directing our focus towards avenues that promise the greatest performance enhancements.

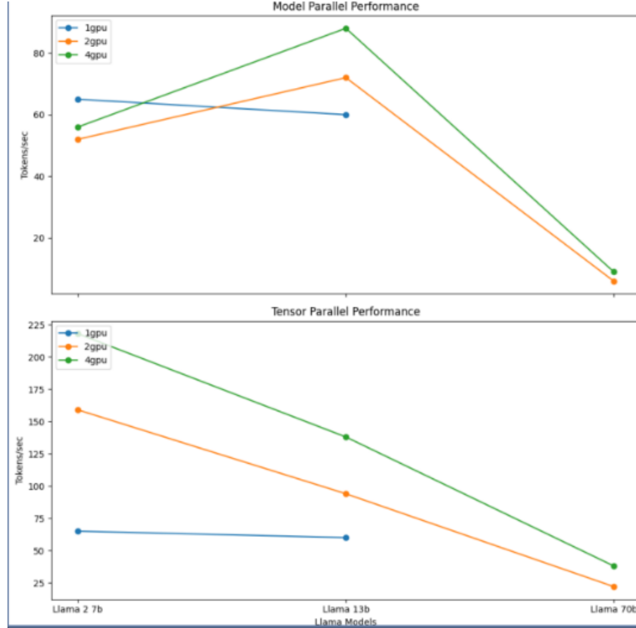

Figure 5. Model Parallel vs. Tensor Parallel

5. Future Works

Looking ahead, our research trajectory aims to delve into several promising avenues for enhancing performance and scalability in deep learning models. One avenue of exploration involves the integration of deepspeed, a cutting-edge framework known for its potential to significantly optimize training efficiency. By delving into deepspeed integration, we aspire to unlock potential performance benefits and further streamline our computational workflows. Additionally, we are eager to broaden our horizons beyond the realms of model and tensor parallelism. Exploring alternative parallel computing techniques holds promise for uncovering novel approaches to tackling the complexities of training large language models. Furthermore, our future endeavors include conducting more expansive benchmarking exercises encompassing a diverse array of models and hardware configurations. By extending our benchmarking efforts beyond the confines of 2 and 4 GPUs, we aim to gain a more comprehensive understanding of performance dynamics across varied scenarios. Through these multifaceted investigations, we aspire to continually push the boundaries of performance optimization in deep learning, ultimately advancing the state-of-the-art in the field.

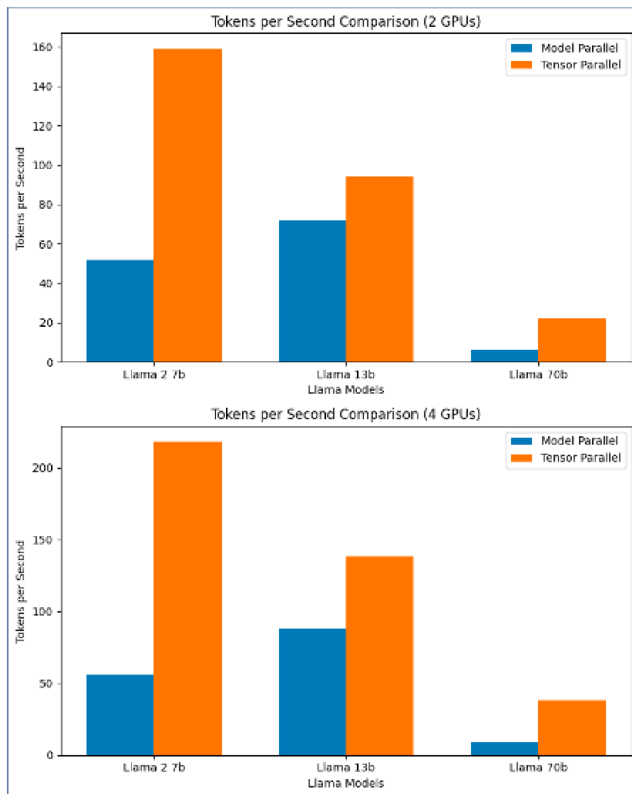

Figure 6. Performance gain via Tensor Parallel

6. References

1. minabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., He, Y. (2022). DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. arXiv. https://doi.org/10.1109/sc41404.2022.00051

2. Bai, G., Chai, Z., Chen, L., Wang, S., L¨u, J., Zhang, N., Shi, T. W., Zhao, Y., Zhu, M., Zhang, Y., Yang, C., Cheng, Y., Zhao, L. (2023).] Be- yond Efficiency: A Systematic survey of Resource- Efficient Large Language models. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2401.00625 Flame Graphs — Wikimedia performance. (n.d.).

3. https://performance.wikimedia.org/php-profiling/

4. Gregg, B. (n.d.). Flame Graphs. https://www.brendangregg.com/flamegraphs.html

5. Liu, Y., He, H., Han, T., Xu, Z., Liu, M., Tian, J., Zhang, Y., Wang, J., Gao, X., Zhong, T., Peng, Y., Xu, S., Wu, Z., Liu, Z., Zhang, X., Zhang, S., Hu, X., Zhang, T., Niu, Q., . . . Ge, B. (2024). Understanding LLMs: A Comprehensive Overview from Training to Inference. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2401.02038

6. Optimize TensorFlow performance using the Profiler. (n.d.). TensorFlow. https://www.tensorflow.org/guide/profile

7. Rajbhandari, S., Rasley, J., Ruwase, O., He, Y. (2019). ZERO: Memory Optimizations Toward Training Trillion Parameter Models. arXiv (Cornell University). https://doi.org/10.48550/arxiv.1910.0205

8. Wang, Q., Chen, Y., Li, Z., Tang, Z., Guo, R., Wang, X., Zhou, A., Chu, X. (2024). Towards Efficient and Reliable LLM Serving: A Real-World Workload Study. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2401.17644

9. hang, H., Ning, A., Prabhakar, R., Wentzlaff, D. (2023). A hardware evaluation framework for large language model inference. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2312.03134