Scaling Large Language Models

Koushik Sivarama Krishnan
V01031395
koushik0901@uvic.ca
University of Victoria

ABSTRACT

In this report, it is our intention to discover the pragmatic issues associated with scaling out the training of Large Language Models on multi-GPU configurations to GPT-3 [1] quality models. We do this by first taking the matrix multiplication operation using two NVIDIA T4 GPUs. We then compare the performance of this approach with that of the case when a single GPU is used. Next, we proceed to comprehend and outline the correlation between variety of designs of Machine Learning models, and their computational complexity. About all the different layers that are involved in building a transformer model and the computational complexity are described in the report. Last but not the least, we present the conclusion of the PTD-P technique introduced by Narayanan et al. [3] for the training of large-scale Megatron-LM on the GPU clusters.

ACM Reference Format:
Koushik Sivarama Krishnan. 2024. Scaling Large Language Models. In *Proceedings of University of Victoria (Koushik Sivarama Krishnan).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The release of GPT-3 [1] and other large language models represents a new era in the technology space of artificial intelligence and machine learning, opening up new avenues for what is possible. These models have even shown superior results in many NLP tasks such as translation, summarization, and even producing their own creative work. Still, the training of such models poses a very significant computational problem especially in the issues of scalability of the computational power. This report aims at investigating the complexities and real-world implications of increasing the training of LLMs to the size of GPT-3 [1] with particular emphasis on use of multiple GPUs.

GPT-3 [1] and similar models are notable for their performance but also for their size – often billions of parameters. The training of such giants is computationally intensive, memory intense, and time-consuming, making their training to be done in distributed computing environments. This demand is one of the most effectively managed by Multi-GPU configurations which use the parallel processing capabilities of multiple GPUs to handle the calculation-intensive workloads necessary for training. However, effectively utilizing such configurations involves overcoming several non-trivial engineering challenges, including issues related to data and model parallelism, synchronization of model updates, and optimization of communication overhead among GPUs.

This report starts with an experiment that involved two NVIDIA T4 GPUs to perform matrix multiplication, a basic task in training of neural networks. This experiment forms part of the basic groundwork to analyze the gains in efficiency and performance in using a system with two GPUs than when using a system with one GPU. The results offer a working reference of what can and cannot be expected when moving from one to several GPUs.

After the experimental analysis, the report goes into the details of the complexity of the designs of machine learning models and computational complexity. Every layer and part of a transformer model which serves as the basis of models such as GPT-3 [1] has its own computation requirements. It is therefore important to understand these requirements for training in a way that can be scaled up and made affordable.

Finally, the discussion broadens to a thorough review of the novel PTD-P technique reported by Narayanan et al. [3] in their paper in 2021. This technique handles the problem of training large scale language models on GPU clusters by combining all types of parallelism: tensor, pipeline and data-parallelism in a way that results in enhanced computational efficiency and reduced training time. Several technical approaches and methodological frameworks used in the PTD-P technique are valuable in advancing the learning to unprecedented magnitudes and within the most efficient resource utilization.

In the process of this report, we propose to outline both the practical and methodological difficulties of LLMs' scaling and to provide the reader with the clear understanding of how it is possible to apply the modern techniques of parallel computing to obtain high performance in machine learning. It is valuable for researchers, practitioners, and technologists who want to further advance the state-of-art in machine learning, so that they need to stay current on these strategies and technologies which are crucial for the creation of the future generations of AI systems.

2 MULTI-GPU MATRIX MULTIPLICATION

The C++ implementation of the matrix multiplication code using CUDA also clearly explains the utilization of multi-GPU parallelisation for high scale computation. The code is made specifically to divide the computational load of matrix multiplication into two GPUs, thereby demonstrating the benefits of parallel computing. The subsequent sections describe the theoretical approach of how the matrix multiplication is supported on the multiple GPUs.

Koushik Sivarama Krishnan, July 28 2024, Victoria, BC

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

2.1 GPU Parallelization and Workload Load Balance

The main focus of the implementation is to use the computing power of several GPUs for the multiplication of matrices in the least amount of time. Multiplication of matrices is a time-consuming process most especially if the matrices involved are large matrices. In this implementation, the workload is divided to half of the rows of matrix A to each GPU thus making it faster than single GPU setup.

The reason that we only send a portion of matrix A and the whole matrix B to each GPU is that matrix B is utilized in every computation of element of matrix C, therefore, if we partition matrix B, the overhead of communication between the GPUs would more than outweigh the benefits of parallelism. This is not the case through the duplication of matrix B across both GPUs, the implementation will not be hindered through this and each GPU will compute the element of matrix C on its own.

2.2 Memory Management Across Multiple GPUs

An important part of this implementation is memory management because the code includes matrix allocation on both GPUs for matrices A, B, and C. The implementation guarantees that every GPU works on a unique portion of the matrix A and the corresponding portion of matrix C The memory for these matrices is allocated on the GPUs to reduce data transfer time between the CPU and the GPU during the computation phase.

First of all, the matrices are defined on the host (CPU), while only the part of the matrix A needed for computation, and matrix B are transferred to the GPU memory. The result matrix C is set to zero in both GPUs to ensure that there are no interferences that may be occasioned by the different results that are generated by the different warps and threads. This computation is followed by transfer of partial results by each GPU back to the host and then the coalescing of these results into the result matrix.

2.3 Kernel Launch and Warp-Level Parallelism

A CUDA kernel is called for the core part of the matrix multiplication and this is performed by the GPUs in parallel. In the kernel, the threads are organized into warps, with a warp containing 32 threads, and all the threads of a warp perform the same operation simultaneously. It uses warp-level parallelism which means that 16 threads in a warp compute one element of the final matrix, C, and since it can make use of the built-in CUDA warp-level synchronization and communication instructions such as __shfl_down_sync for partial sums within the warp.

Multiplication of two given matrices is done by blocks of matrix A columns and blocks of matrix B rows, and each thread gets the partial product. To be efficient, each thread computes through the elements of matrix B and move through the matrix B in chunk of sixteen to ensure all passes through the matrix. The partial sums are then summed within the warp using warp-level primitives which are much cheaper in terms of synchronisation compared to thread block.

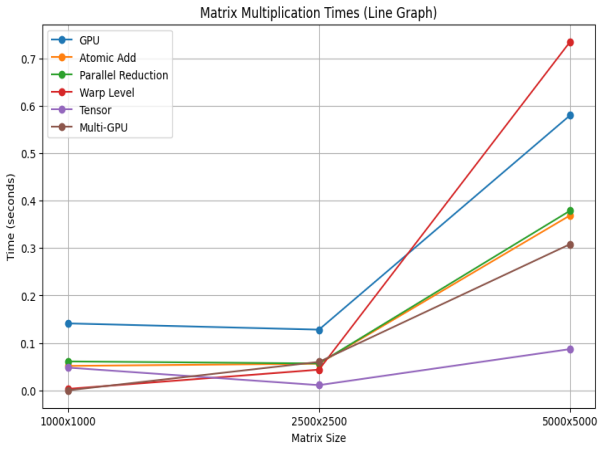

Figure 1: Performance comprising of different techniques

2.4 Synchronization and Result Aggregation

Following the kernel execution, in each GPU, the result must be broadcasted to all the GPUs to ensure all the GPUs possess the same result. It also ensures that all threads and GPUs have performed their computation through the use of the cudaDeviceSynchronize() function which halts the host until the device has made its computations. At synchronisation, the data computed from every segmented part of the matrix are read from every GPU to the host. These partial results represent matrix C of the final output where each of them corresponds to the results of each GPU.

Finally, this code successively gathers the partial results into the output matrix as follows: Since each GPU was calculating different rows of the output matrix, there is no extra computation needed; the results obtained are added to get the final matrix.

2.5 Performance Considerations

It also includes event with CUDA events for timing intervals of measures in matrix multiplication between the two GPUs. Consequently, this implementation demonstrates how multi-GPU parallelization can be employed for a basic operation central to many machine learning algorithms: matrix multiplication and etc. Through the application of the efficient memory management, effective partition of the work load and by the use of the burst parallelism at the warp level it is clear that the time taken is greatly reduced and in fact the code is faster than if implemented on one GPU. These methods are extremely useful for scaling the training of large models such as GPT-3 [1]. From Figure 1 we can see that using multi-GPU setting where 16 threads operate on one output element is significantly faster than using a single GPU where 16 threads operate on one output element. Although 2x performance improvement is expected theoretically, due to the inherent delays in communication between these GPUs, we do not see the expected 2x performance improvement over using a single GPU setting.

2.6 Parallelism to Support LLM Training

The PTD-P parallelism technique introduced by Narayanan et al. [3] in their work can be aimed at finding an efficient approach to train

large-scale language models of the Megatron-LM type on multi-GPU clusters. The basic concept of PTD-P is to utilize Pipeline Parallelism, Tensor Parallelism, and Data Parallelism in parallel yet asynchronously in an optimal manner in order to make the best use of the resources and achieve the highest training rate. This combination lets the training of such huge models with tens of trillions of parameters, across thousands of GPUs, scale nearly linearly while keeping computational costs under control.

2.7 Pipeline Parallelism

Pipeline parallelism refers to partitioning of the layers of a neural network across several GPUs. Batches of layers are assigned to individual GPUs and both the forward and the backward passes are interleaved across the GPUs. This technique is very effective for training huge models that are unable to fit into the memory of a single GPU. Pipelining the layers across the multiple devices guarantees that each of the devices processes part of the model at a time hence eliminating memory bounds. One major challenge in this approach are the pipeline bubbles, where some GPU is idle while others are still working on data.

2.8 Tensor Parallelism

Tensor parallelism is used to partition the computations which occur within separate layers of a model, for example matrix computations, across multiple GPUs. Tensor parallelism splits the data in each layer and divides the computation in the same way, which makes it possible to compute operations that would be too large to handle on a single GPU. The technique also helps in reducing the memory footprint of the model as the work is spread across different GPUs and in addition allows for parallel executions. Tensor parallelism is most effective within multi-GPU servers where the high-speed interconnects such as NVLink can be utilized to reduce the associated overhead.

2.9 Data Parallelism

Data parallelism is common in distributed training where several replicas of the model are stored in different GPUs and each replica works on a different portion of the data. Updates of the model are coordinated across all the copies of the gradient and they are synchronized periodically. Data parallelism is very scalable if many GPUs are to be used, but they incur communication overhead when syncing gradients, especially in large clusters.

2.10 The PTD-P Approach

In the PTD-P approach, pipeline, tensor, and data parallelism techniques are integrated in a way, which maximize the efficiency of each of the methods and minimize their drawbacks. PTD-P combines pipeline parallelism across multiple GPUs for different layers of the model, tensor-parallelism within each layer to distribute the computation as well as data parallelism to manage large datasets to allow for scaling to thousands of GPUs. This way of parallelization also helps to ensure that big models can be trained with low memory overhead and low communication overhead while still having high computational throughput.

Another key enhancement made in PTD-P is the 1F1B (One Forward One Backward) pipeline schedule which is more efficient than the previous pipeline schedules in that it minimizes the pipeline bubble. This schedule splits the layers' chunks to be processed by each GPU, making it possible to interleave computations and, therefore, making the pipeline flush happen earlier. In consequence, GPUs become more utilized, and overall throughputs are advanced by up to 10% to traditional pipeline schedules.

2.11 Performance and Scalability

In their experiments, Narayanan et al. [3] showed that PTD-P can be operationally scaled to tens of thousands of GPUs while achieving high levels of computational performance. For example, the approach obtained 52% of the peak device throughput during training of the trillion-parameter model using 3072 NVIDIA A100 GPUs. This level of efficiency makes PTD-P one of the most efficient methods within the training of large-scale models such as GPT-3 [1] within a reasonable time.

Overall, PTD-P represents a major advancement in distributed training for large language models. It provides a practical solution to the challenges involved in training extremely huge models on multi-GPU clusters through carefully optimized multiple forms of parallelism. This enables the training of trillion-parameter models within months—not years—which definitely is an essential tool to unlock capability at scale in AI systems.

3 COMPUTATIONAL REQUIREMENTS OF LLMS

Of all factors in the optimization of machine learning models, especially large-scale architectures like BERT [2], the relationship between parameters and FLOPs is very vital. While parameters relate to weights in a neural network learned during training, FLOPs are a metric that provides the total number of arithmetic operations done by the model in one forward pass. Both give information on how much resources are needed to train/run the model and how it will scale with model dimensions.

3.1 BERT Architecture Parameters and FLOPs

There are several kinds of layers in the BERT[2] architecture, all adding to the model's general parameter count and computational cost. The main layers in BERT [2] are the self-attention layers [4], feed-forward layers, layer normalization layers, and the embedding layer.

Self-Attention Layer: The self-attention [4] mechanism is inherent to the BERT [2] transformer architecture. This layer contains multiple weight matrices projecting the input embeddings into query, key, and value representations. While this is the case, the attention [4] layer parameters scale with the square of the hidden size because of weight matrices that are of dimension Hidden Size x Hidden Size. FLOPs in this layer increase quadratically with the sequence length because each token in the input sequence needs to attend to every other token.

Feedforward Layer: Following the attention [4] mechanism is the feedforward network, consisting of two fully connected layers. The number of parameters in these layers depends on the hidden size and the intermediate size, which is usually a multiple of the hidden size. The FLOPs for this layer also scale linearly with the

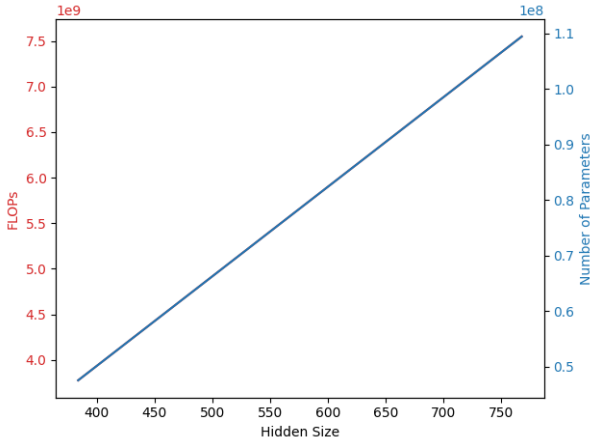

Figure 2: Scaling of BERT [2] model when manipulating the hidden_size with respect to the number of parameters and FLOPs

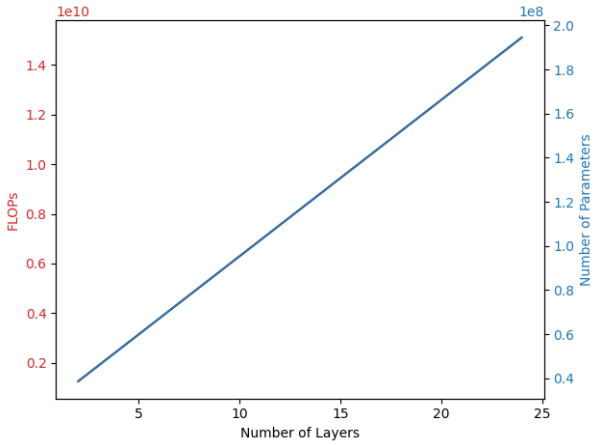

Figure 3: Scaling of BERT [2] model when manipulating the number of layers with respect the number of parameters and FLOPs

hidden size, intermediate size, and sequence length, as it performs dense matrix multiplications.

Layer Normalization: Each BERT [2] layer is followed by layer normalization, which helps stabilize training. Although the normalization layers contain relatively few parameters, they still contribute to the total FLOPs, as each element in the sequence needs to be normalized.

Embedding Layer: The embedding layer converts token indices into dense vectors. The number of parameters in this layer is proportional to the vocabulary size multiplied by the hidden size. While the embedding layer holds a significant portion of the model's parameters, it requires relatively few FLOPs since the embedding process is essentially a lookup operation.

3.2 Scaling with Hidden Size

The first plot, 2, shows how an increase in the hidden size affects FLOPs and the number of parameters of the BERT [2] model. According to the plot, when the hidden size increases, FLOPs and parameters scale linearly with it.

FLOPs: The FLOPs increase from about 4 billion when the hidden size is 384, to about 7.5 billion FLOPs when it is 768. This linear relationship comes from the fact that both attention [4] and feedforward require more computation as the hidden size goes up. In particular, all matrix multiplications involved in the attention [4] mechanism and feedforward network have operations that go with the square of the hidden size.

Parameters: At a hidden size of 384, the number of parameters is around 400 million; at a hidden size of 768, it is over 1.1 billion. This rise in the number of parameters is due to the larger matrices of weights that the self-attention [4] and feedforward layers require. Since the parameters are linear with respect to the hidden dimension for each layer, increasing the hidden dimension will add parameters linearly.

This linear scaling with the hidden size is expected, as both FLOPs and the number of parameters in BERT's[2] architecture depend a lot on the hidden size. The more sizable the hidden dimensionality, the more complex the model becomes, both in computational demand and memory requirements.

3.3 Scaling with the Number of Layers

The second plot 3 shows the influence of increasing the number of layers on FLOPs and parameters in the BERT [2] model. One can notice that with an increasing number of layers, FLOPs and parameters are growing linearly.

FLOPs: Moving from 2 layers to 24, the FLOPs grow from about 200 million to roughly 1.4 trillion FLOPs. Additional layers bring in more computations for both the self-attention [4] and feedforward networks; thus, this increase is direct and huge in total FLOPs. Since the architecture of BERT [2] involves similar operations on every layer, more layers increase the computational workload linearly.

Parameters: The number of parameters increases linearly from roughly 200 million for 2 layers to roughly 2 billion for 24 layers. With each additional layer, a constant number of parameters is added on top, which explains the linear trend one sees in the above plot. Those additional parameters simply correspond to the extra weight matrices in the self-attention [4] and feedforward networks of each layer.

3.4 Implications for Model Design

These scaling patterns illustrate how critical architectural decisions in BERT—particularly [2] the hidden size and the number of layers—directly impact the computational complexity and memory requirements of the model.

Increasing Hidden Size: Increasing the hidden size augments the model's capacity to capture more complex patterns in the data. A higher hidden size, however, vastly increases FLOPs with higher parameter counts, which increase memory usage and prolong training times.

Increasing the Number of Layers: More layers can deepen the model, allowing it to learn hierarchical representations that

possibly bring in better performance for harder tasks. However, as with increasing hidden size, this also linearly increases the number of parameters and FLOPs, hence requiring more computational resources.

Ultimately, what is important in applying BERT [2] in real-life scenarios is finding the appropriate balance between model size and computational efficiency. Such scaling plots offer valuable lessons for making practical choices when scaling BERT [2] up or down for various use cases, guaranteeing that the efficiency and manageability of the model are within the hardware constraints available.

4 CONCLUSION

Research into the computational requirements of LLMs like BERT [2] and GPT-3 [1] reveals all the complexities that go into designing and scaling such models. According to the report, the relationship between parameters, FLOPs, and essential architectural decisions defines resource demands for training and deployment of such models. Experiments and Analyses—it is demonstrated that all important parameters, like hidden size, the number of layers, and parallelism strategies, relate directly to computational and memory requirements for the model.

In particular, the multi-GPU matrix multiplication implementation further attested to the fact that parallel processing could seriously improve computational efficiency, especially when dealing with large models. It was also shown by the detailed architecture breakdown from BERT how parameters and FLOPs scale with different components, thus modeling the trade-offs between model complexity and resource consumption. This, in turn, demonstrates through scaling plots that increases in both are linear with hidden size and the number of layers, hence giving practical guidelines to practitioners when optimizing BERT[2] for certain applications.

Finally, the PTD-P parallelism technique has come up with a very promising solution about how to efficiently train an extremely large model across thousands of GPUs. In terms of maximizing resource utilization by merging different forms of parallelism, PTD-P is the way to go, enabling the training of models with a trillion parameters within a reasonable timeframe.

It is the final conclusion of the report: model design and computational efficiency should go side by side when working with LLMs. Combined with deep understanding of scaling behavior, advanced parallelization techniques will be required to push the boundaries of what can be achieved in machine learning, to ensure these models remain feasible to train and deploy on available hardware.

REFERENCES

[1] 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. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL] https://arxiv.org/abs/2005.14165

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[3] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. arXiv:2104.04473 [cs.CL] https://arxiv.org/abs/2104.04473

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762