Cycling Faster Through CycleGAN

Vincent La **Sahire Ellay** **Isabelle Rathbun**

Abstract

There has been a significant increase in the size of deep learning models, particularly with the rise of transformer-based architectures. Due to this growth, there has been significant dedication to distributed training strategies for scalable training and better hardware use. However, a large focus of these strategies has been for large language models, which do not effectively generalize to smaller, more heterogeneous image-based generative models with vastly different architectures and training mechanics. In this paper, we profile and attempt to optimize CycleGAN- a small but relatively popular image-based style transfer generative model. We perform a detailed profiling analysis to identify areas with significant bottlenecks in memory or compute leading to slowdowns. Based on this analysis, we implement various parallelism strategies to improve performance.

1 Introduction

Deep learning models have become dominant across many domains and applications in recent years.[1] With the development of transformers and the resulting expansion of transformer-based models, this dominance has continued to expand. These generative models have proven to excel at a variety of tasks including natural language processing, vision tasks in biological applications, and style transfer. Style transfer refers to taking an input image and transferring it to another style while keeping the same fundamental image - from a photograph to a sketch, turning a summer scene into winter or a day scene into night, etc. As more machine learning models have shifted to using transformer based encoders or decoders, the average size of models has exponentially increased. Prior to transformers, most networks used some convolutional neural network (CNN) or recurrent neural network (RNN) backbones, which usually are on the magnitude of thousands to millions of parameters.[2] Transformer based models, on the other hand, can range from millions to billions of parameters.[3] This means that many popular and commercial models are extremely large- they often cannot fit on a user's machine, cannot be trained on a single GPU, take very long to train (weeks to months), and require billions of instances of training data.[4] Due to this massive scale, there has been a large focus on distributed training methods.

One example of parallelizing the training is data parallelism,[5] which involves copying the entire model across multiple GPUs and splitting the extremely large training dataset across the GPUs, so each one holds a different subset of the dataset. The gradients are averaged and the weights are synchronized after every update. The limitation of this method for very large models is that it does not help address a situation where the model itself is too large to fit on a single GPU. On the other hand, model parallelism splits the model itself by either sharding the layers themselves (tensor parallelism) or separating sequential layers across multiple GPUs (pipeline parallelism). The limitations of this method are that it is difficult to implement, especially when the architecture of the model is complex, and complex latent space operations can be difficult to pipeline. One can also parallelize using hybrid parallelism,[6] which involves a combination of data and pipeline parallelism to try and balance the memory and communication needs of large-scale setups.

Many of the major large language models (LLMs), such as ChatGPT,[7] LLaMA,[8] or BERT,[9] have powerful scaling strategies, often developed specifically for their architectures and data, but there are not well-established, generalizable optimization methods for other generative models, especially

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

those that are image-based rather than language. Strategies that work well for LLMs do not generalize well to image-focused models due to fundamental differences in architecture, model size, training dynamics, and overhead cost tradeoffs.[10] Transformer-based language models are usually very deep networks consisting of uniform layers, which gives clear points to partition layers across multiple GPUs. Image-based generative models can have more heterogeneous architecture involving a variety of networks, blocks, convolutions and connections, and layers in a shallower network than an LLM, but much more varied. This makes it more difficult to uniformly or simply split a model across different GPUs since popular methods like tensor splitting or pipelining do not work when the model is not modular or sequential enough. Additionally, image models are often smaller than their LLM counterparts - where LLMs must be sharded because they are too large to fit on a single GPU (up to hundreds of billions of parameters), image models often do not exceed 5 billion parameters, which means they can fit on a single GPU, so the overhead cost of model parallelism does not provide much benefit. The training dynamics of these two model types also do not align - whereas LLMs are often trained once and deployed for use, image models often need to be retrained for each task or for each domain they are used in. This means that for an LLM, minimizing training time is the priority at nearly any cost, whereas for image models, there needs to be a focus on the cost of training and usability rather than solely on minimizing training time at any cost.

In this paper, we aim to optimize CycleGAN[11]- a basic style transfer generative image model- and hopefully create more generalized optimizations for generative image models from there based on our profiling results. We profile the model to identify computational bottlenecks in the architecture and work to develop some custom optimizations to parallelize aspects of our workload, especially working within the generator and discriminator setup of CycleGAN, and accounting for the fact that it is a fairly small model, so it meets all of the challenge points discussed above.

2 Related Work

2.1 Generative Adversarial Networks

Generative Adversarial Networks[12] (GANs) are a class of deep learning models introduced in 2014 consisting of at least two neural networks- a generator and discriminator. The generator takes noise and generates a data sample; the discriminator takes in an instance of real data and an instance of generated data and aims to distinguish which is the real and which is generated. The two networks are trained simultaneously against each other, commonly referred to as adversarial training. The generator generates more and more realistic data with the goal of fooling the discriminator into thinking it is real data, while the discriminator gets better at distinguishing the generated and real data. These models are susceptible to training collapse, which is when the generator starts producing identical outputs every time, and are highly sensitive to changes in their architecture.

2.2 CycleGAN

CycleGAN[11] is a specific GAN designed for style transfer without paired training data published in 2017 that we chose to investigate for this project. It transfers images from one style or domain into another without needing paired training data- for example, turning horses into zebras or summer scenes into winter. CycleGAN introduced the concept of cycle-consistency, which is when taking an image from style A to style B, you should get the original image if you go from style B back to style A. This is trained using cycle-consistency loss to ensure that the images transferred back and forth result in the original image. The architecture of the model is two GANs- one going from style A to style B and one going from style B to style A. Each GAN has its own generator and discriminator functioning like a standard GAN. There are four possible generator architectures that are inherently supported- two sizes of ResNet and two sizes of U-Net- and three possible discriminators- two variants of PatchGAN and one of pixelGAN. We exploit this to increase the overall size of the model for testing our optimizations by selecting the largest U-Net options for the generators to nearly double the model size.

3 Preliminary Profiling

We profile a CycleGAN workload from this implementation of the CycleGAN and pix2pix models on a single GPU using the Pytorch profiler with the Kineto backend across 20 batches and visualized

it using Holistic Trace Analysis. We collected data on where the GPU is spending time for its job (Temporal breakdown), what its idle time is attributed to (Idle time breakdown), how much time is spent in each kernel type (Kernel breakdown), how many "short" GPU kernels ran, how many instances are there of the CPU taking a long time to schedule a kernel, and how many kernels have a large delay between the CPU scheduling the kernel and the kernel executing (CUDA kernel launch statistics). Through our profiling, we found that we should approach optimization by attempting to reduce idle time to increase utilization.

3.1 Kernel Breakdown

Figure 2 shows the most time consuming kernels among memory kernels. Moving data around on device took the most time. It was unclear to us how this would impact forms of parallelism in distributed set ups. In the case of pipeline parallelism, we figured many of these kernels would switch to kernels that move data between devices, incurring a lot of additional communication costs.

Figure 3 shows the percentage of time taken by computation and memory kernels. CycleGAN spends the vast majority of time in compute kernels, so we find that CycleGan is very compute bound.

3.2 CUDA Kernel Launch Stats

Figure 4 shows how many GPU kernels there are that had a "long delay" (> 100 us) between being scheduled and execution. It shows how many such kernels there were for several launch delay durations. The large number of kernels with small launch delays may contribute to the high idle time as seen in the temporal breakdown plot (figure 7), The largest spike alone contributes at least 6% of CPU time. Although it is difficult to tell, since this plot doesn't say what proportion of kernels have a launch delay or how much of the total time launch delays contributes against the total time spent.

Figure 5 shows how many GPU kernels there are that took a "long time" (> 50 us) for the CPU to schedule. Figure 6 shows how many GPU kernels there are whose duration was shorter than it took for the CPU to schedule it. Both show how many such kernels there were for several durations. It was not clear to us how these stats related to other profiling results, so they are hard to interpret.

All three CUDA Kernel Launch Stat graphs were difficult to take anything actionable away from since they don't tell what portion of all kernels were launch delay outliers/runtime event duration outliers/short.

3.3 Temporal Breakdown

Figure 7 shows what portion of time the GPU spent idle (idle_time), what portion was spent doing computations (compute_time), and what portion of time the GPU was being used for communication or memory events (non_compute_time). The majority of the time the GPU was idle and was mostly spent on computation otherwise. There is little communication/memory event overhead, which bodes well for pipeline parallelism as it requires a lot of communication.

3.4 Idle Time Breakdown

Figure 8 shows how the GPU spent its idle time. It shows time spent waiting for kernels to enqueue (host_wait), the duration between kernels (kernel_wait), and wait time due to unknown causes (other). The vast majority of time was spent waiting for the host, which aligns with the launch delay outliers graph (figure 4).

4 Optimizations

From our preliminary profiling runs, we note that the memory footprint of the CycleGAN pipeline is relatively small when compared to the available RAM of an A100 gpu. As such, distributed setups are generally unnecessary and we focus on optimizations for a single gpu training setup.

The CycleGAN forward pass can roughly be broken down as:

```
% Cycle 1: a --> b' --> a'
```

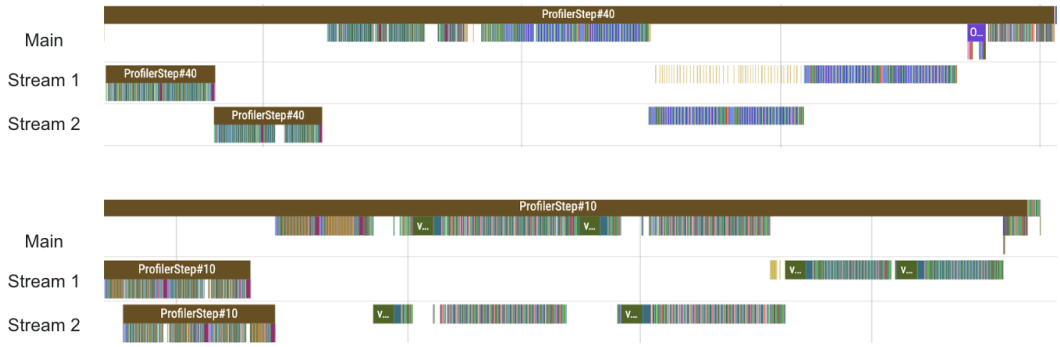

Figure 1: Comparison between our parallel forward pass implementation with batch size of 1 (top) versus batch size of 16 (bottom). The smaller workload does not actually achieve any sort of parallel compute in practice. This trace visualization is made using Perfetto.

```
fake_b = G_A(a)
rec_a = G_B(fake_b)  #G_B(G_A(a))

% Cycle 2: b --> a' --> b'
fake_a = G_B(b)
rec_b = G_A(fake_a)  #G_A(G_B(b))
```

where:

- G_A maps $A \to B$
- G_B maps $B \to A$
- A and B are the set of all images belonging to classes A and B, respectively
- $a \in A, b \in B$

Given that cycle 1 and cycle 2 operate on independent parts of the input batch, there is inherent room for parallelism. We parallelize the CycleGAN forward pass by launching each cycle on a separate CUDA stream. This allows for some automatic parallelism in the backward pass as each backward CUDA operation runs on the same stream as the corresponding forward operation. We profile our optimization in the same manner as our preliminary profiling experiments.

5 Results

Generator	Discriminator	Implementation	Avg. Forward Pass GPU Time (ms)
Resnet	PatchGAN	Baseline	734.42
Resnet	PatchGAN	Ours	809.28
U-Net	PatchGAN	Baseline	150.90
U-Net	PatchGAN	Ours	175.50

Table 1: Comparison of different CycleGAN setups and their performances. All runs are profiled with a batch size of 16 on a single A100 gpu.

We profile our parallel forward pass implementation in the same manner as our preliminary profiling experiments, on a single A100 GPU. We notice that using a small workload, such as the default batch size of a single pair of images, does not actually achieve any parallel computation. This is because the CUDA streams are still launched in a sequential manner, and with a small workload a single cycle computation is able to complete before the second stream is able to be launched. After increasing the batch size to 16, we did observe parallel computation across the separate CUDA streams, in both the forward and backward passes.

We observe that our parallel forward pass implementation exhibits a ~1.1x slowdown over the original baseline implementation. While increasing the problem size does achieve parallel computation on the GPU, the overall latency of our forward pass implementation is slower due to synchronization calls between the separate CUDA streams. We also experiment with changing the generator models' architecture to see if a larger model would have an impact on the overall performance of our implementation. The U-Net 256 generator model has ~23 M parameters in comparison to the default Resnet generator model's ~11.4 M. Despite this, the U-Net forward pass is empirically much faster than the Resnet forward pass. As such, the slowdown factor for our parallel forward pass implementation does not seem to change significantly across different generator architectures.

6 Conclusion

We attempt to optimize a CycleGAN pipeline which consists of training 4 separate models, two generators and two discriminators, in an adversarial training setup. First, we profile a single GPU setup. Through our initial profiling results, we found that CycleGAN is very compute bound with low memory event overhead and high GPU idle times. Based on this, we determined that we can increase throughput by increasing GPU utilization. Since the 4 models had an overall relatively low memory footprint, we strived to implement optimizations for a single GPU setup. We attempt to optimize CycleGAN's utilization on just a single GPU by parallelizing the forward pass execution which has embarrassingly parallelizable operations in the computation of two separate cycles that operate on separate data within a batch. Our implementation achieves parallelism by launching each cycle computation on a separate CUDA stream. We find that even with single GPU parallelism, when using small batch sizes, the forward pass compute is still almost entirely sequential because computation of a single cycle completes before the other CUDA stream is able to be launched. By increasing the batch size, we observe a higher overlap of computation on the GPU, though the overall forward pass is still slower than the baseline setup due to synchronization calls.

We experiment with using a larger U-Net generator architecture to amortize the synchronization calls across more computation, but find that the overall slowdown factor remains the same. Despite the U-Net generators being larger in parameter size, we find that the latency of the forward pass is faster than the ResNet generators. We leave the exploration of different generator and discriminator architectures for CycleGAN with more computationally intensive pipelines for future work. Additionally, in tandem with scaling up the model sizes, data parallel and distributed data parallel setups may be worth exploring for speeding up the overall processing of training batches.

7 Figures

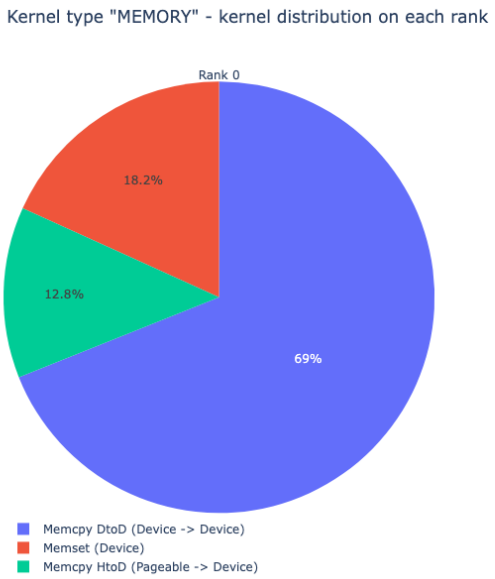

Figure 2: shows the most time consuming kernels among memory kernels. Memcpy DtoD (Device -> Device) kernels took 69% of time, Memset Device) kernels took 18.2% of time, and Memcpy (Pageable -> Device) kernels took the remaining 12.8% of time.

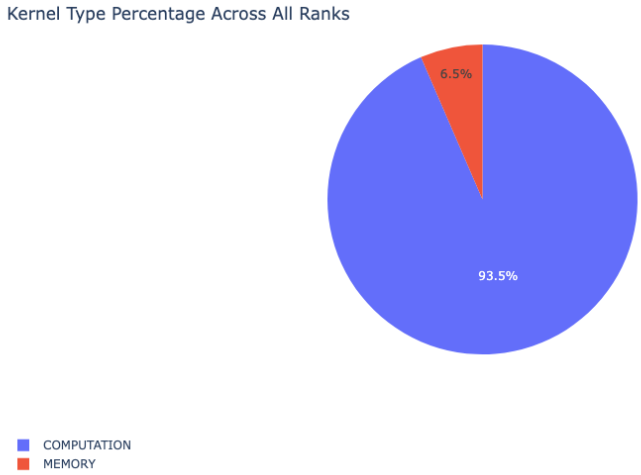

Figure 3: shows the percentage of time taken by computation and memory kernels. Computation kernels took 93.5% of time and memory kernels took 6.5% of time.

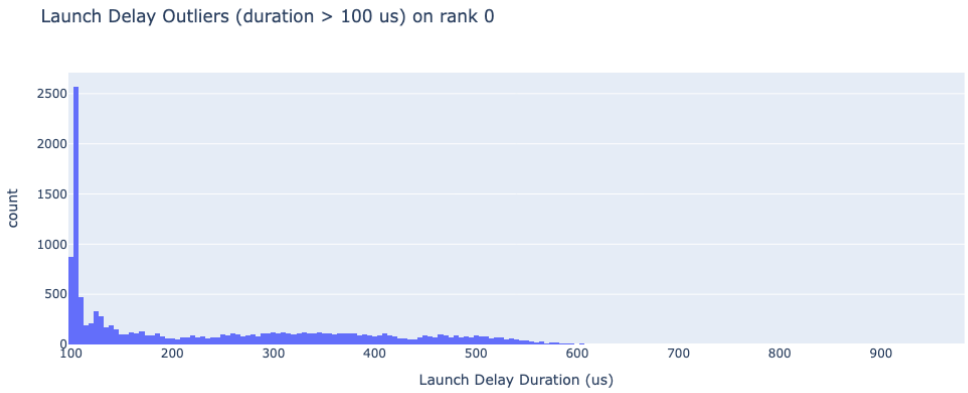

Figure 4: shows how many GPU kernels there are that had a “long delay” (> 100 us) between being scheduled and execution. It shows how many such kernels there were (y-axis) for several launch delay durations (x-axis, in microseconds)

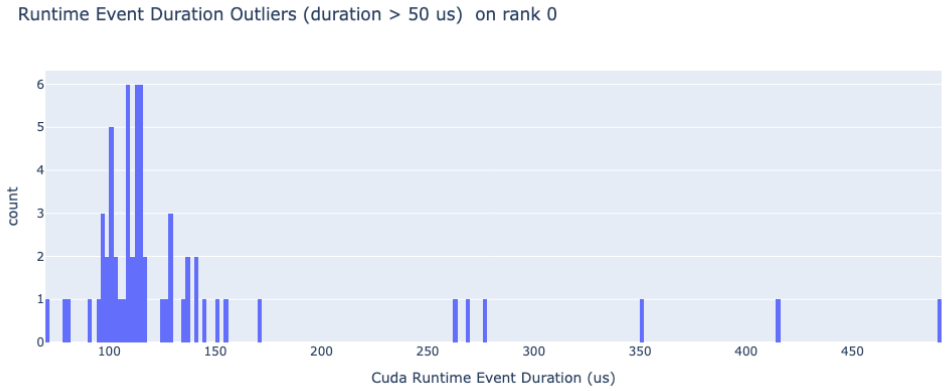

Figure 5: shows how many GPU kernels there are that took a “long time” (> 50 microseconds) for the CPU to schedule. It shows how many such kernels there were (y-axis) for several durations (x-axis, in microseconds).

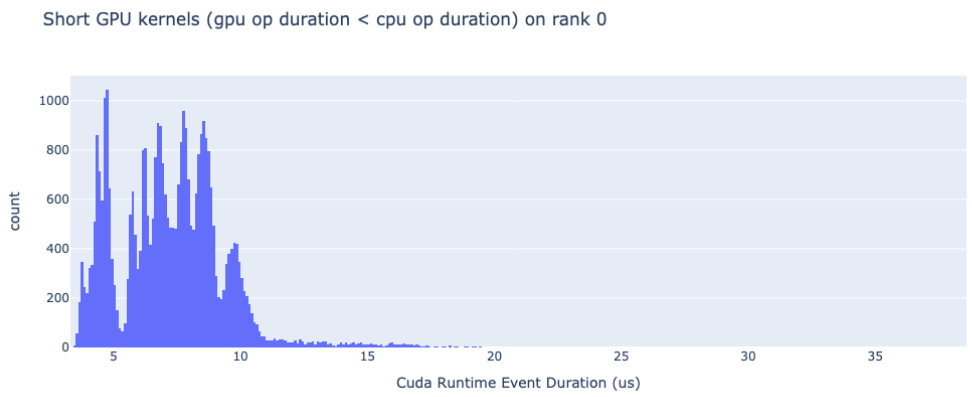

Figure 6: Shows how many GPU kernels there are whose duration was shorter than it took for the CPU to schedule it. It shows how many such kernels there were (y-axis) for several durations (x-axis, in microseconds).

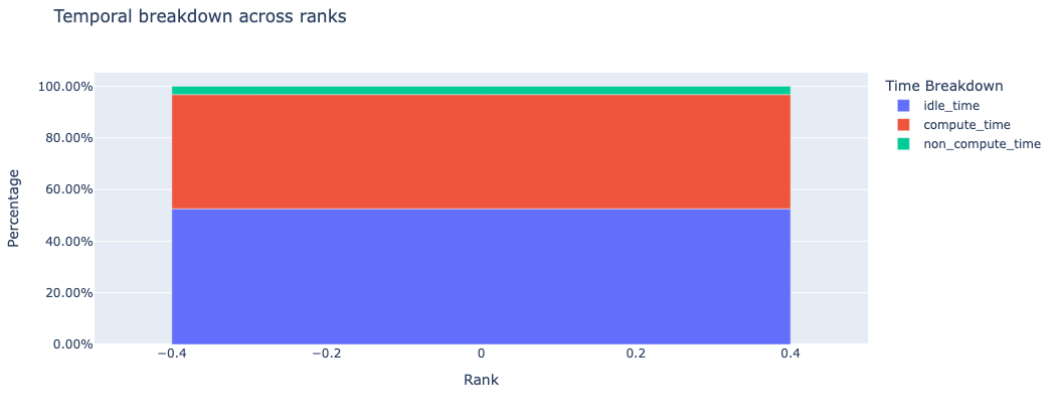

Figure 7: shows what portion of time the GPU spent idle (idle_time, the bottommost block), what portion was spent doing computations (compute_time, the middle block), and what portion of time the GPU was being used for communication or memory events (non_compute_time, the topmost sliver).

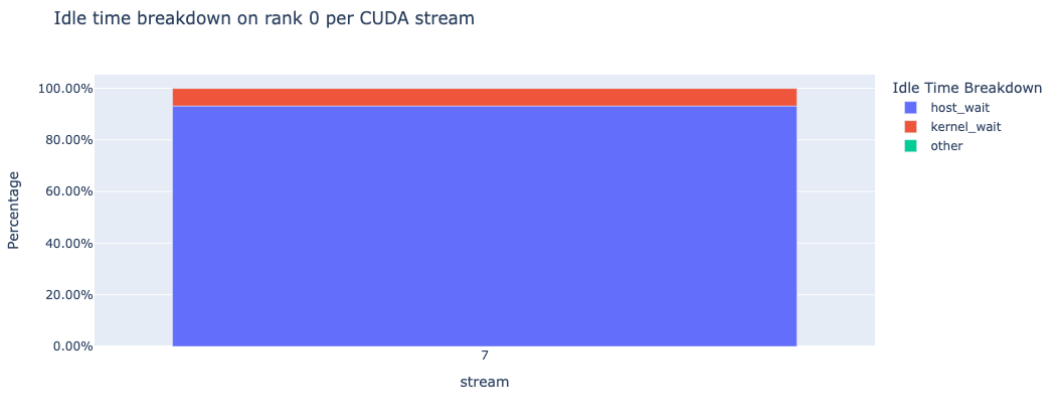

Figure 8: shows how the GPU spent its idle time. It shows time spent waiting for kernels to enqueue (host_wait, the largest, bottommost block), the duration between kernels (kernel_wait, the next largest, topmost block), and wait time due to unknown causes (other, so small it is not visible).

References

[1] Junhua Ding, Haihua Chen, Yunhe Feng, and Tozammel Hossain. Applications of Deep Learning Techniques. *Electronics*, 13(17):3354, January 2024. Number: 17 Publisher: Multidisciplinary Digital Publishing Institute.

[2] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A Survey of the Recent Architectures of Deep Convolutional Neural Networks. *Artificial Intelligence Review*, 53(8):5455–5516, December 2020. arXiv:1901.06032 [cs].

[3] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A Survey of Large Language Models, March 2025. arXiv:2303.18223 [cs].

[4] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning, April 2021. arXiv:2104.07857 [cs].

[5] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research*, 20(112):1–49, 2019.

[6] 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. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, August 2021. arXiv:2104.04473 [cs].

[7] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. 2018.

[8] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, February 2023. arXiv:2302.13971 [cs].

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. arXiv:1810.04805 [cs].

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, June 2021. arXiv:2010.11929 [cs].

[11] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired Image-To-Image Translation Using Cycle-Consistent Adversarial Networks. pages 2223–2232, 2017.

[12] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks, June 2014. arXiv:1406.2661 [stat].