Optimizing Diffusion Transformers

Chuanyang Jin Saining Xie
New York University
{cj2133, sx352}@nyu.edu

Abstract

We investigate general methods and recent tools aimed at accelerating deep learning models, with a particular focus on Diffusion Transformers (DiT). We delve into optimized attention, Just-In-Time compilation, gradient checkpointing, mixed-precision training, feature pre-extraction, among other strategies, providing a comprehensive report on both successful and unsuccessful attempts. Consequently, we manage to accelerate DiT by 95% *and decrease memory usage by* 59%*. We provide code at this https URL.*

1. Introduction

Diffusion models are a class of likelihood-based models which have recently been shown to produce high-quality image [5] [8] [9] [10] [16]. These models generate samples by gradually removing noise from a signal, and their training objective can be expressed as a reweighted variational lower-bound. An innovative variant of these models, the Diffusion Transformer (DiT) [15], has recently been introduced. This model replaces the traditional U-Net [17] backbone with a transformer that operates on latent patches. The scalability of DiT is highly impressive and it has set a new benchmark with its state-of-the-art Frechet Inception Distance (FID).

A parallel architecture has been utilized in another paper, known as U-ViT [2]. Similar to DiT, U-ViT employs a transformer-based diffusion model with a few minor variations. These include the use of skip connections between blocks, a distinct conditioning method, time and positional embeddings, and final layers that serve the same purpose but are implemented differently. Since U-ViT outperforms DiT in terms of training speed, we extensively refer to U-ViT's strategies during our endeavor to optimize the DiT model, before eventually achieving superior performance.

2. Methods

2.1. Optimized Attention

Flash Attention. FlashAttention [4] offers an innovative solution to the memory bottleneck issues between GPU's High Bandwidth Memory (HBM) and on-chip Static Random-Access Memory (SRAM), a problem particularly pronounced in the processing of long sequences by memory-intensive Transformers. The key innovation of this approach is its tiling strategy, breaking down the input data into more manageable tiles or fragments. This fragmentation significantly reduces memory read/write operations and optimizes data processing within the GPU's memory hierarchy.

xFormers. The xFormers library [13] places substantial emphasis on memory efficiency, and strives to maintain a swift execution speed.

Check then Select. We specifically delve into the implementation of the attention mechanism in the library `timm.models.vision_transformer`. This library is widely used and regularly updated with the most recent methods. The standard implementation of DiT utilizes its `attention` class.

The class initially examines the attributes of the `torch.nn.functional` module to determine if it contains the "scaled dot product attention" function (requires Pytorch 2.0). If the function is unavailable, the attention is calculated using the conventional mathematical method. If present, the class automatically employs the function to compute the attention. The function supports three implementations of scaled dot product attention: FlashAttention (previously mentioned), xFormers (also mentioned earlier), and a PyTorch implementation defined in C++. This "check then select" approach, often used to optimize performance, is similarly employed in the U-ViT implementation. Its attention class similarly checks the dependencies before choosing from FlashAttention, xFormers, or the mathematical implementation, as appropriate.

2.2. JIT Compilation

The introduction of the `Torch.compile` function in PyTorch 2.0 has ushered in a significant enhancement in code execution performance. This function allows Just-In-Time (JIT) compilation [1] of PyTorch code into optimized kernels. JIT compilation is a technique that compiles the source code

during execution, as opposed to ahead-of-time, facilitating faster code execution by exploiting runtime characteristics.

The function `Torch.compile` offers convenient JIT compilation options. in various contexts. For instance, it can be used to compile arbitrary Python functions in the following manner: `opt_foo = torch.compile(foo)`. Alternatively, it can also be utilized as a decorator for Python functions, i.e., `@torch.compile` preceding the function definition. Moreover, Torch.compile can be applied directly to instances of `torch.nn.Module`, streamlining the performance optimization process for PyTorch models. A typical usage for compiling a model would be `model = torch.compile(model)`.

The `Torch.compile` function operates under three different modes to provide flexibility based on user requirements. The default mode prioritizes a balance between compilation efficiency and performance optimization, ensuring the compilation process neither consumes excessive time nor requires substantial memory overhead. The second mode, reduce-overhead, aims to significantly diminish framework overhead, albeit at the cost of additional memory consumption. Lastly, the max-autotune mode focuses on generating the fastest possible code by extensively autotuning the compilation process. This mode, however, requires a considerably longer compilation time.

2.3. Gradient Checkpointing

Training large-scale models, like diffusion models, presents significant challenges even on modern GPUs. These models can be so massive that the GPU memory often becomes insufficient, and the time it takes to train these models can become considerably long. One solution to mitigate these issues is gradient checkpointing, a technique designed to lessen the memory footprint by trading off computation time for memory.

In the context of deep learning, training a model involves computing forward and backward passes through the network. The forward pass computes the output for a given input, while the backward pass calculates gradients, which are necessary for updating the model's parameters. Traditionally, all intermediate activations of the forward pass are stored in memory as they are needed for the backward pass, which can be memory-intensive for large networks. Gradient checkpointing addresses this problem by not storing all intermediate activations during the forward pass. Instead, it saves a subset of these activations, known as checkpoints. During the backward pass, the non-saved activations are recomputed from these checkpoints as needed. While this approach does increase computation time slightly, it significantly reduces the memory usage, making it possible to train larger models on hardware with limited memory.

2.4. Mixed Precision Training

Mixed precision training [14] is a technique that enhances the speed and decreases memory usage during deep neural network training. As the size of neural networks expand to improve accuracy, so does the computational demand and memory requirements. To address these challenges, mixed precision training introduces the use of half-precision floating point numbers for storing weights, activations, and gradients during training, thereby offering considerable memory and computational benefits.

However, half-precision floating numbers inherently have a limited numerical range compared to single-precision numbers, which can lead to a loss of information. To counter this, mixed precision training employs two key strategies. Firstly, it maintains a single-precision copy of the weights, which accumulates the gradients after each optimizer step. This copy is then rounded to half-precision format during the training process, mitigating the potential for information loss. Secondly, it proposes scaling the loss appropriately to account for the information compromise with half-precision gradients.

This novel approach has been demonstrated to be effective across a range of models. Automatic Mixed Precision (AMP) training has been enabled in many libraries, such as `torch.cuda.amp` in Pytorch Version 2.0 or the Accelerate library.

2.5. Feature Pre-extraction

In training loops, there can sometimes be instances of repetitive computations where the parameters remain unchanged. In such cases, we can optimize our process by storing the results of these computations for future use. Our DiT model provide such a scenerio, where we utilize a pretrained variational autoencoder (VAE) model [11] to map images to their corresponding latent space [16] prior to deploying transformer-based Denoising Diffusion Probabilistic Models (DDPMs). The output from the VAE, referred to as "features," can be pre-computed and stored. In subsequent training iterations, these pre-calculated features can be directly accessed and utilized. This method greatly expedites the overall operation of the model by eliminating repetitive computations.

2.6. Other Methods

Beyond the methodologies previously outlined, there are several other common strategies to accelerate models. Model pruning [3], for instance, reduces computational complexity by selectively eliminating non-critical connections or weights based on specified criteria. Knowledge distillation [7] involves training a smaller, more efficient model (the student) to mimic the performance of a larger, more complex model (the teacher), thereby achieving comparable accuracy with fewer computational resources. Furthermore,

Epoch	1	2	3	4	5	6	7	8	9	10	Average
Experiment 1	0.69	0.65	0.66	0.69	0.74	0.75	0.74	0.74	0.75	0.75	**0.746**
Experiment 2	0.65	0.65	0.64	0.64	0.65	0.63	0.64	0.64	0.64	0.62	**0.639**
Experiment 3	0.62	0.61	0.61	0.62	0.62	0.62	0.62	0.62	0.62	0.62	**0.618**
Experiment 4	0.65	0.59	0.60	0.60	0.60	0.61	0.60	0.60	0.60	0.60	**0.605**

Table 1. **Stability of Performance.** We train a DiT-XL/2 using 4 A100 GPUs, and a global batch size of 256.

leveraging pre-trained models [6] [12] through either feature extraction or weight fine-tuning can expedite model training and enhance performance.

3. Basic Evaluations

3.1. Stability of Performance

Our experiments are carried out on the NYU High-Performance Computing (HPC) Greene clusters.To establish a robust baseline and facilitate a fair comparison of performance, we examine the consistency of the training speed, by replicating the training procedure multiple times under the same configuration. The outcomes are presented in Table 1.

The speed of training demonstrates considerable stability within the same experiment and only minor variations across different experiments. On average, the training speed is found to sustain a relatively consistent rate of approximately 0.62 steps per second.

3.2. Comparison between Models

We conduct a comparison of various model configurations of DiT and U-ViT, the results of which are presented in Table 2. These findings provide the baseline for further comparisons. Our analysis reveals that U-ViT outperforms DiT in terms of speed and memory efficiency.

It's noteworthy that the impact of augmenting the number of GPUs can vary across different models. For instance, when we double the number of GPUs, we observe a speed increase ranging from 35% to as much as 135%.

3.3. Plugging in DiT model into U-ViT

We endeavor to tune the model parameters and structure with the aim of enhancing the speed of DiT and reducing that of U-ViT, in order to understand the underlying reasons for the observed performance gap. However, no significant changes are noted, as illustrated in Table 3. This prompts us to consider whether the model itself is the root cause.

To test this, we integrate the DiT model directly into the U-ViT training protocol, maintaining the same optimizer, learning rate scheduler, and batch size. Interestingly, the speed of the model surpass that of the original U-ViT, while matching the performance of the original DiT. Therefore, we conclude that the performance gap is not attributable to the model structure, but rather to the training techniques employed, such as mixed precision training.

4. Experiments

4.1. Experimental Setup

In the our experiments for performance optimization, we train a DiT-XL/2 model with a global batch size of 128, and we test it on both a single A100 and a pair of A100s. Since the model speed stabilizes a few minutes post-initialization, we run each experiment for an hour and record the steady-state speed.

4.2. Optimized Attention

Our DiT model employs the attention mechanism in the `timm.models.vision_transformer` library. As previously discussed, it opts for FlashAttention, xFormers, or a conventional mathematical method, contingent on the dependencies available. In our standard environment, DiT defaults to the conventional mathematical method for attention computation, while U-ViT utilizes FlashAttention.

We experiment with modifying the DiT environment and enforce the use of either FlashAttention or xFormers. Despite the fact that xFormers facilitated a 17% acceleration in U-ViT's speed, neither FlashAttention nor xFormers significantly enhanced our performance. The maximum speed improvement observed was a modest 1% on the A100.

4.3. JIT Compilation

Initially, we apply Just-In-Time (JIT) compilation to the DiT module, observing no significant alteration in speed. Subsequently, we extend the JIT compilation to the DiT-Block and FinalLayer modules, but again, no substantial speed changes are detected. Moreover, when we opt to compile all the functions within the DiT model instead of the modules, we actually notice a decrease in speed. As a point of comparison, we replicate these experiments with a similar ViT model and find no appreciable speed increase.

Given our suspicion that our Distributed Data Parallel (DDP) might influence JIT compilation, we experiment with compiling the model post-DDP instead of pre-DDP. However, this adjustment fails to yield any speed improvements. Lastly, we attempt to switch to the "max-autotune" mode for JIT compilation. This method, however, consumes significantly more memory, compelling us to abandon this approach.

Model	GPU	# Parameters	Batch Size	Steps/Sec	Memory Usage	Sampling mode
DiT-XL/2	A100*1	675M	128	/	Out Of Memory	Discrete
DiT-XL/2	A100*2	675M	128	0.77	66259 / 81920	Discrete
DiT-XL/2	A100*4	675M	128	1.05		Discrete
DiT-XL/2	RTX8000*4	675M	128	0.33		Discrete
DiT-XL/2	A100*1	675M	256	/	Out Of Memory	Discrete
DiT-XL/2	A100*2	675M	256	/	Out Of Memory	Discrete
DiT-XL/2	A100*4	675M	256	0.62		Discrete
DiT-XL/2	RTX8000*4	675M	256	/	Out Of Memory	Discrete
DiT-L/2	A100*1	625M	128	/	Out Of Memory	Discrete
DiT-B/2	A100*1	272M	128	0.55	50510 / 81920	Discrete
U-ViT-H/2	A100*1	501M	128	0.8	24812 / 81920	Discrete
U-ViT-H/2	RTX8000*1	501M	128	0.28	22813 / 46080	Discrete
U-ViT-L/2	RTX8000*4	287M	128	1.53	12583 / 46080	Continuous
U-ViT-L/2	RTX8000*4	287M	1024	0.25	19127 / 46080	Continuous
U-ViT-L/2	A100*1	287M	1024	0.18		Continuous

Table 2. **Comparison between models.** Batch Size denotes the total batch size distributed across all utilized GPUs. Memory Usage refers to the amount of memory consumed per individual GPU.

Experiment	Steps/Sec
Original	1.62
mlp time embed = True	1.62
final layer conv = True	1.60
out blocks with nn.linear	1.71
discrete sampling	1.62
disabled grad checkpointing	2.04
disabled mixed precision	0.47

Table 3. **Tuning U-ViT parameters and structures.** We train a U-ViT-S using one RTX8000 with a small batch size of 32. Despite modifications in parameters and structures, the alterations are found to be minimal. The variations observed with disabled gradient checkpointing and deactivated mixed precision are anticipated and discussed in further details later.

4.4. Gradient Checkpointing

Gradient checkpointing trades computation time for memory by recomputing some of the forward pass during the backward pass. We employ a `ckpt wrapper` to encapsulate the forward pass of each `DiTBlock` module. This ensures that all subsequent calls employ checkpointing. Subsequently, we utilize `torch.utils.checkpoint` to incorporate gradient checkpointing. This approach successfully decreases GPU memory consumption by 55%.

4.5. Mixed Precision Training

When we strive to manually implement half-precision training by converting all data and model parameters from `torch.float32` to `torch.float16`, we confront a number of issues. One prominent challenge is that some PyTorch operations don't fully support half-precision formats. To overcome these hurdles, we turn to automatic mixed precision training facilitated by the Accelerate library. However, the Accelerate library also manages distributed training, leading to conflicts with our Distributed Data Parallel (DDP) process. Consequently, we overhaul the entire training procedure using the new paradigm.

Automatic mixed precision training accelerates DiT by 21% on a single A100, and 10% on two A100s. Additionally, this approach contributes to more significant enhancements when combined with the technique discussed later.

4.6. Feature Pre-extraction

We employ a pre-trained variational autoencoder (VAE) to map all $3 \times 256 \times 256$ images from ImageNet into features of shape $3 \times 32 \times 32$. These features, along with their class labels, are stored in advance, a process that takes under 10 hours on a single A100. In subsequent training sessions, we introduce a new `Dataset` and `Dataloader` to load these features. Once the features for each batch are concatenated, they are directed to the transformer-based Denoising Diffusion Probabilistic Models (DDPMs).

This methodology significantly speeds up the overall operation of the model by cutting out redundant calculations. It boosts the speed of DiT by 30% on a single A100, and 34% on two A100s. Moreover, this strategy enhances the speed of the mixed-precision DiT by 56% on two A100s.

5. Conclusion

In summary, we effectively accelerate DiT-XL/2 by 95% and reduce memory usage by 59% through the utilization of gradient checkpointing, mixed-precision training, and feature pre-extraction. The training loss and FID scores display

Model	Steps/Sec ↑ (A100)	Memory ↓ (A100)	Steps/Sec ↑ (Two A100s)	Memory ↓ (Two A100s)
Original	–	Out of Memory	0.77	66259 MB
GCP with TF32 Disabled	0.12	51991 MB	0.24	32167MB
Gradient Checkpointing (GCP)	0.43	44045 MB	0.77	30801 MB
GCP + AMP	0.52	40461 MB	0.85	31099 MB
GCP + Feature	0.56	29191 MB	1.03	26909 MB
GCP + AMP + Feature	0.84	27485 MB	1.33	27193 MB

Table 4. **Performance Optimization Experiments.** AMP refers to the Automatic Mixed Precision training. Feature refers to Feature Pre-extraction. The rates of changes are compared to the model with GCP.

identical patterns, confirming that our modifications do not compromise the model's capabilities.

In addition to refining my skills in High Performance Computing (HPC) and troubleshooting a variety of issues on the platform, this project impart several crucial insights:

Firstly, when computational resources are constrained - often leading to extended wait times - it's advisable to conduct exploratory, smaller-scale tests that require fewer GPUs or less time. This allows for more rapid debugging and performance assessment, setting the stage for more complex experiments and more refined code.

Secondly, there are times when we aspire to bridge the performance gap (in terms of speed or accuracy) between two similar methods. In our case (DiT vs. U-ViT), rather than solely focusing on accelerating the slower model, it might be more enlightening to think outside the box and explore ways to slow down the faster model. Integrating elements of the slower model into the faster one for testing can be a good strategy. This would allow for a more direct analysis of the underlying causes.

Finally, optimization methods can be highly specific, and a strategy that works for one model may not necessarily yield the same results for a closely related model. Furthermore, the outcomes can drastically differ depending on the hardware used.

References

[1] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35:97–113, 2003. 1

[2] Fan Bao, Shen Nie, Kaiwen Xue, Yue Cao, Chongxuan Li, Hang Su, and Jun Zhu. All are worth words: A vit backbone for diffusion models. 2022. 1

[3] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. What is the state of neural network pruning? *CoRR*, abs/2003.03033, 2020. 2

[4] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher R'e. Flashattention: Fast and memory-efficient exact attention with io-awareness. *ArXiv*, abs/2205.14135, 2022. 1

[5] Prafulla Dhariwal and Alex Nichol. Diffusion models beat gans on image synthesis. *CoRR*, abs/2105.05233, 2021. 1

[6] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Liang Zhang, Wentao Han, Minlie Huang, Qin Jin, Yanyan Lan, Yang Liu, Zhiyuan Liu, Zhiwu Lu, Xipeng Qiu, Ruihua Song, Jie Tang, Ji-Rong Wen, Jinhui Yuan, Wayne Xin Zhao, and Jun Zhu. Pre-trained models: Past, present and future. *CoRR*, abs/2106.07139, 2021. 3

[7] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *ArXiv*, abs/1503.02531, 2015. 2

[8] Jonathan Ho. Classifier-free diffusion guidance. *ArXiv*, abs/2207.12598, 2022. 1

[9] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *CoRR*, abs/2006.11239, 2020. 1

[10] Jonathan Ho, Chitwan Saharia, William Chan, David J. Fleet, Mohammad Norouzi, and Tim Salimans. Cascaded diffusion models for high fidelity image generation. *CoRR*, abs/2106.15282, 2021. 1

[11] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013. 2

[12] Nupur Kumari, Richard Zhang, Eli Shechtman, and Jun-Yan Zhu. Ensembling off-the-shelf models for gan training. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10641–10652, 2021. 3

[13] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, and Daniel Haziza. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022. 1

[14] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Frederick Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *ArXiv*, abs/1710.03740, 2017. 2

[15] William S. Peebles and Saining Xie. Scalable diffusion models with transformers. *ArXiv*, abs/2212.09748, 2022. 1

[16] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. *CoRR*, abs/2112.10752, 2021. 1, 2

[17] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. 1