Profiling and Optimizing Training Performance of BERT-Base

Abstract

With the widespread adoption of large-scale pretrained models such as GPT, BERT, and ViT in natural language processing and computer vision, model sizes and training complexity have increased exponentially. This growth has led to significantly longer training times and soaring computational and financial costs. To address this challenge, we propose a comprehensive training acceleration framework that significantly improves training efficiency with minimal loss in accuracy. Using BERT-Base on the AG News dataset as the baseline, we leverage NVIDIA's official profiling tools—Nsight Systems and Nsight Compute—to perform a full-stack performance analysis, from system-level bottlenecks to kernel-level operator profiling. Our optimization pipeline includes mixed precision training, attention layer rewriting, model graph compilation, layer-wise parameter freezing, and gra-dient accumulation. These methods collectively yield a **5.07× speedup** in training time while maintaining a negligible accuracy drop (within 0.05). This work of-fers a systematic methodology and empirical insights for improving the training efficiency of large models, especially in resource-constrained deployment and re-search scenarios.

1 Introduction

In recent years, large-scale pretrained models such as the GPT series, BERT, and ViT have achieved widespread success across various subfields of artificial intelligence. As a result, model sizes have expanded rapidly, and training complexity has increased dramatically. In practical applications, this trend has led to a significant rise in training time and escalating resource costs—including computational power, energy consumption, and wall-clock time—posing serious challenges for researchers and developers working under limited hardware conditions.

Although substantial progress has been made in model compression and inference acceleration, systematic optimization of the training process remains relatively underexplored. In particular, how to improve training efficiency without sacrificing model accuracy or architecture integrity, through coordinated software–hardware optimization, is an important yet insufficiently addressed research problem.

This work focuses on the following question: *How can we systematically optimize the training pipeline of large models to improve efficiency while maintaining accuracy?*

To address this question, we construct a targeted performance optimization pipeline based on the BERT-Base model trained on the AG News dataset for four-class text classification. Relying on NVIDIA's official profiling tools—Nsight Systems and Nsight Compute—we conduct multi-level analysis of the training process, identifying key bottlenecks such as forward/backward propagation, optimizer steps, and a large number of fragmented CUDA kernels (e.g., GEMM and elementwise operations).

Based on the profiling insights, we design and progressively implement the following optimization strategies:

1. **Mixed-Precision Training**: Enabling Tensor Core acceleration for high-throughput matrix multiplication and convolution operations;
2. **Attention Layer Rewriting**: Replacing the original attention module with the xFormers implementation on NVIDIA RTX 3070, and using a custom fused attention kernel written in Triton on RTX 4090 to reduce kernel launch overhead and memory traffic;
3. **Model Compilation and Graph Optimization**: Leveraging `torch.compile()` to perform operator fusion and optimize execution graphs;
4. **Gradient Accumulation and Layer-Wise Unfreezing**: Simulating large-batch training under memory constraints via gradient accumulation, and dynamically unfreezing only the $(\text{epoch}+1) \bmod \text{total\_layers}$-th Transformer layer per epoch while keeping the rest frozen;
5. **Parallel Data Loading**: Increasing the number of DataLoader worker threads to alleviate I/O latency.

We perform ablation studies and combination comparisons for the above strategies to evaluate their impact on training time and model performance. The final optimized configuration—xFormers + mixed precision + model compilation + gradient accumulation + parameter unfreezing—demonstrates substantial acceleration on the NVIDIA RTX 3070, achieving a **5.07× speedup** with a classification accuracy of **0.8849**.

In summary, this paper proposes and validates a generalizable and practical training acceleration framework that preserves model accuracy. Our findings not only offer a reproducible optimization path for Transformer-based models, but also provide valuable insights for deployment in long-sequence tasks and resource-constrained environments.

2 Related Work

2.1 Performance Profiling Methodologies

At the hardware level, NVIDIA Nsight Systems [4] reveals inefficient memory access patterns in GEMM operations during attention computation via CUDA API instrumentation, while Huawei Ascend Sprof [5] introduces a Fusion Opportunity Score to evaluate operator fusion potential on heterogeneous architectures.

2.2 Computation and Memory Optimization

To address memory constraints, gradient checkpointing has advanced from static strategies [6] to dynamic algorithms. Jain et al. [7]'s Smart Checkpointing reduces checkpoint counts by 30% in 72-layer Transformers through computational graph topology analysis.

Operator-level optimizations like FlashAttention [8] minimize DRAM accesses to theoretical limits ($\mathcal{O}(N^2) \to \mathcal{O}(N)$) via tiling and shared memory, achieving $1.7\times$ speedup in GPT-3 training. ZeRO-Offload [9] pioneers CPU-GPU hybrid memory management, enabling $10\times$ larger models on single GPUs.

2.3 Parallelism and Communication Optimization

Model parallelism innovations have transformed large-scale training. Megatron-LM [10]'s tensor parallelism splits matrix multiplications column-wise, optimizing AllReduce communication for 175B-parameter models on 8-GPU clusters. Alpa [11] automates parallel strategy search, dynamically blending data, pipeline, and tensor parallelism.

Communication compression breakthroughs include 1-bit Adam [12] (94% traffic reduction via error-compensated quantization) and PowerSGD [13] (80% less communication via low-rank gradient decomposition in ResNet-152).

2.4 Emerging Directions

Recent work explores energy-aware optimization: Google's Carbon-API [14] integrates training carbon emissions into performance metrics via dynamic batch sizing. In quantization, LLM.int8() [15] demonstrates lossless INT8 inference for 175B models, while FP8 mixed-precision training [16] reduces energy consumption by 40% via dynamic scaling.

Compiler optimizations gain traction: XLA [17]'s auto-operator fusion cuts kernel launch overhead by 3%, and TVM AutoTensorization [18] generates hardware-optimized GEMM implementations.

3 Method

3.1 Gradient Accumulation

In the standard BERT training workflow, model parameters are typically updated after each mini-batch. While this allows for frequent optimization feedback, it can lead to excessive CUDA kernel launches and suboptimal GPU utilization, particularly under small batch settings constrained by memory limitations.

To improve training efficiency, this study adopts a gradient accumulation strategy. Instead of updating parameters after every mini-batch, gradients are accumulated over multiple steps (e.g., 4), and the optimizer updates the parameters only after the specified accumulation interval. This approach effectively increases the logical batch size without exceeding GPU memory capacity.

Specifically, for an accumulation step size of N:

- Gradients are computed and accumulated over N consecutive forward-backward passes;
- The loss is scaled by $1/N$ at each step to maintain numerical stability;
- After every N steps, a single optimizer.step() is performed, followed by optimizer.zero_grad();

This method effectively reduces kernel launch frequency, memory access overhead, and host-device synchronization, thereby improving overall training throughput. It also enables large-batch training under limited GPU memory conditions.

On the AG News text classification task, applying gradient accumulation resulted in an approximate **1.256× speedup** per training epoch, while maintaining nearly identical test accuracy (accuracy drop < 0.015). These results validate the effectiveness and stability of this optimization strategy in memory-constrained environments.

3.2 Layer-wise Parameter Unfreezing

To further accelerate training and reduce GPU memory consumption, we adopt a dynamic layer freezing strategy, in which only a subset of model parameters is updated during the early training stages. We then progressively unfreeze the BERT encoder layers over training epochs. This approach significantly lowers training costs while preserving the pretrained knowledge embedded in the model.

Specifically, we initially freeze all parameters of the BERT backbone (i.e., setting `requires_grad=False` for all `model.bert.parameters()`). Then, during each training epoch, we dynamically unfreeze one Transformer encoder layer while keeping the rest frozen. The core logic of this strategy is as follows:

1. Suppose the BERT encoder contains L layers, denoted as $\{encoder.layer[0], \ldots, encoder.layer[L-1]\}$;
2. At the beginning of the e-th epoch, we unfreeze the $(L - ((e+1) \bmod L))$-th layer, while keeping all other layers frozen;
3. At the start of each epoch, we dynamically construct the optimizer (AdamW) based only on parameters with `requires_grad=True`, thereby avoiding unnecessary gradient computation and memory overhead.

This mechanism effectively simulates a "shallow-to-deep" fine-tuning schedule, avoiding sudden updates to all model layers in the early stages. It reduces training instability and promotes more stable convergence.

Empirically, using only the layer-wise unfreezing strategy—i.e., updating one Transformer layer per epoch—reduced the average training time per epoch from **456.66 seconds** to **159.00 seconds** on the AG News classification task, yielding a **2.872$\times$ speedup**. Meanwhile, model performance remained stable, with only a marginal drop in test accuracy (approximately 0.002), demonstrating an excellent trade-off between efficiency and performance.

3.3 Rewritten Attention Module and Combined Optimization Strategy

To improve the training efficiency and resource utilization of BERT-based text classification tasks, a combined optimization strategy is employed. This strategy integrates a rewritten attention mechanism with mixed-precision training and compiler-level optimizations. By leveraging both hardware acceleration features and algorithmic improvements, the approach significantly reduces memory consumption and improves computational throughput.

3.3.1 Rewritten Attention Mechanism

Conventional self-attention computation incurs substantial memory usage and redundant operations, especially when processing long input sequences. The memory-efficient attention implementation based on the `xFormers` library utilizes highly optimized CUDA kernels to drastically reduce intermediate activation storage and duplicate computations. This fundamentally lowers memory footprint and accelerates computation.

A custom `XformersSelfAttention` module is introduced, preserving the original model architecture by explicitly copying the weights and biases of the query (Q), key (K), and value (V) projections. The standard attention modules in each BERT encoder layer are replaced with this module to achieve more efficient attention computation without affecting the model's semantics.

3.3.2 Mixed-Precision Training

To further enhance computational efficiency, mixed-precision training is adopted. By performing most operations in half-precision floating point (FP16), the method significantly reduces memory usage and accelerates matrix operations via NVIDIA's Tensor Cores, which are optimized for low-precision computation. This technique increases training throughput while preserving model accuracy and reducing hardware demands.

3.3.3 model compilation

Compiler-level graph optimizations are enabled via PyTorch's `torch.compile` functionality. This utility automatically performs kernel fusion, inlines operations, and reduces Python runtime overhead, contributing to both training and inference acceleration. In combination with the rewritten attention mechanism and mixed-precision training, it forms a complete performance optimization pipeline.

Together, these three techniques effectively address both computational and memory bottlenecks inherent in Transformer model training. On the AG News text classification task, this combined strategy achieved a **1.375$\times$ speedup**, with only a negligible drop in test accuracy (0.0014). These results demonstrate that the proposed method substantially improves training efficiency while maintaining strong model performance. The integration of attention mechanism rewriting, mixed-precision training, and compiler optimization highlights the feasibility and practical value of this approach for efficient Transformer training under resource constraints.

3.4 Fused Attention Kernel Optimization with Triton

After replacing the original attention mechanism with xFormers, we sought to further understand and control the low-level operator behavior of the attention module in BERT. To this end, we implemented a custom fused attention kernel using the Triton programming language. This kernel fuses multiple operations—matrix multiplication, Softmax, Dropout, and value weighting—into a single

GPU kernel, reducing launch overhead, minimizing global memory access, and improving memory locality.

In standard BERT implementations, the attention module follows the formulation:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

This computation involves multiple intermediate matrices and kernel launches, making it a significant bottleneck in training. Our custom Triton kernel `fused_softmax_kernel` introduces the following key optimizations:

- **Operator fusion**: Combines $QK^\top$, Softmax, Dropout, and $A@V$ into a single kernel to avoid repeated memory I/O;
- **Efficient memory access**: Uses `tl.load` and `tl.store` to load Q, K, and V blocks into shared memory, reducing cross-thread-block memory access;
- **Dynamic mask support**: Handles attention masks at runtime to prevent invalid attention weights;
- **Scalable grid parallelism**: Implements a 3D grid structure over (batch $\times$ head, row_block, col_block) to support large-scale parallelism.

The full computation is divided into the following six stages:

(1) Program ID and index mapping

Each Triton thread block determines the associated (b, h) pair using its program ID, where $\text{pid}_0 = b \times H + h$. Based on this, we compute the row and column block indices as:

$$\mathbf{r} = r_0 + \{0, 1, \ldots, B_r - 1\}, \quad \mathbf{c} = c_0 + \{0, 1, \ldots, B_c - 1\}$$

Here, B_r and B_c denote the row and column block sizes, respectively.

(2) Q/K vector loading and type casting

Using the computed indices, query and key vectors are loaded from global memory:

$$\mathbf{Q_r} \in R^{B_r \times d}, \quad \mathbf{K_c} \in R^{B_c \times d}$$

To ensure numerical stability, these are cast to `float32`. Masks are applied to exclude out-of-bound rows or columns.

(3) Attention score computation

We compute the scaled dot-product attention scores:

$$\mathbf{A} = \frac{\mathbf{Q_r} \cdot \mathbf{K_c}^\top}{\sqrt{d}}$$

If an attention mask is present, masked positions are replaced with a large negative value (e.g., -65504) to suppress them in Softmax.

(4) Softmax normalization

The attention scores $\mathbf{A}$ are row-normalized using a numerically stable Softmax:

$$\mathbf{P}_{ij} = \frac{\exp\left(A_{ij} - \max_j A_{ij}\right)}{\sum_k \exp\left(A_{ik} - \max_k A_{ik}\right)}$$

This produces normalized attention weights for each query vector.

(5) Output computation (weighted value vectors)

We compute the final output by applying the attention weights to the value vectors $\mathbf{V_c} \in R^{B_c \times d}$:

$$\mathbf{O_r} = \mathbf{P} \cdot \mathbf{V_c}$$

(6) Writing results to output buffer

The final outputs $\mathbf{O_r}$ are stored back into the preallocated output buffer using masked `tl.store` operations to ensure index validity and memory safety.

When executed on an NVIDIA RTX 4090, this Triton-based attention kernel reduced the average epoch training time from **365.00s** to **286.33s**, achieving a **1.36$\times$ speedup**, with only a negligible accuracy drop of **0.0014** on the test set. It is worth noting that due to architectural or driver constraints, the same implementation failed to progress on the NVIDIA RTX 3070, even though it ran successfully on the RTX 4090.

3.5 Parallel Data Loading

To alleviate data I/O bottlenecks and improve GPU utilization, we adopt a parallel data loading strategy.

Specifically, we configure PyTorch's `DataLoader` with `num_workers=4`, which spawns four separate worker processes to asynchronously prefetch training batches. This setup allows data for the next batch to be loaded in parallel while the GPU is still computing gradients and updating parameters, effectively avoiding the blocking overhead of single-threaded data loading.

The core idea is to overlap CPU-side data loading with GPU-side computation, thereby improving the throughput of the entire training pipeline. Conceptually, the parallel loading process can be represented as:

$$\text{DataLoader}_{\text{next}} = \text{ParallelLoad}(\text{Batch}_{i+1}, \text{num\_workers} = 4)$$

In our experiments, enabling parallel data loading reduced the average epoch time from **456.66 seconds** to **441.33 seconds**, yielding a modest but measurable **1.034$\times$ speedup**. While the acceleration is not as significant as that achieved by other strategies, this method provides a stable and efficient data input foundation that complements more impactful optimizations such as gradient accumulation and parameter freezing. Moreover, this approach is simple to implement, incurs negligible overhead, and can be broadly applied to various training tasks—especially those with substantial data loading costs, large batch sizes, or I/O-intensive pipelines.

4 Experiment

4.1 Experimental Design and Analysis

To evaluate the effectiveness of the proposed optimization strategies, experiments are conducted on the BERT-base model using the AG News text classification task. The evaluation focuses on the average training time per epoch and the final test accuracy. Additionally, NVIDIA's NSight Systems tool is used to profile and analyze the Top-5 GPU kernels before and after optimization.

The experimental setup includes the following components:

- **Dataset and Task**: The AG News dataset is used for evaluation. It consists of four balanced categories. Each input sequence is truncated or padded to a maximum length of 50 tokens.
- **Baseline Model**: The baseline employs a standard fine-tuning procedure on BERT-base without any performance optimization techniques.
- **Optimization Strategies**:

  Gradient Accumulation: Gradients are accumulated over multiple steps before updating the model, reducing optimizer invocation frequency and simulating large-batch training under limited memory budgets.

  Layer-wise Unfreezing: One encoder layer is unfrozen per training epoch, allowing shallow layers to "warm up" before deeper layers are trained. This helps stabilize early training and reduces unnecessary parameter updates and compute.

  Attention Operator Rewriting: The native attention mechanism is replaced with fused and memory-efficient operators from the xFormers library, significantly reducing kernel fragmentation and memory overhead. Mixed precision and compilation optimizations are also enabled.

4.2 Baseline Analysis

In the baseline experiment, we trained the model on 96,000 samples and tested it on 7,600 samples. The number of epochs was set to 3. The average accuracy achieved was **0.935**, and the average training time was **457 seconds**.

To further investigate training bottlenecks, we profiled the model using Nsight Systems, identifying the top five CUDA kernels with the highest performance impact. We then used Nsight Compute to analyze each bottleneck in detail, aiming to explore potential optimization strategies.

Figure 1 shows the profiling results from Nsight Systems:

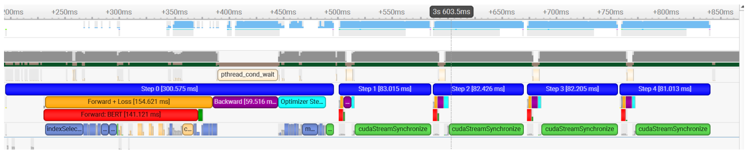

Figure 1: Nsight Systems profiling result of the baseline model

The top five performance-critical CUDA kernels identified are as follows:

- `ampere_sgemm_128x64_tn`
- `ampere_sgemm_128x64_nn`
- `ampere_sgemm_128x64_nt`
- `vectorized_elementwise_kernel`
- `ampere_sgemm_64x64_nn`

Next, we used Nsight Compute to analyze each of these kernels. Figure 2 illustrates the roofline performance chart of the first kernel:

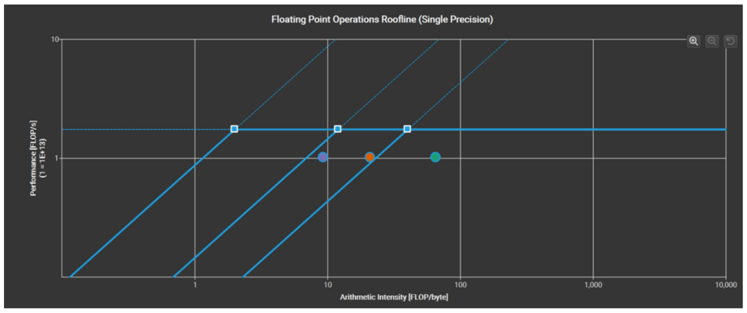

Figure 2: Roofline analysis of `ampere_sgemm_128x64_tn`

The Nsight Compute analysis shows that the top time-consuming CUDA kernels are primarily GEMM (General Matrix Multiply) operations, such as `ampere_sgemm_128x64_*` and `64x64_nn`, which aligns well with the structure of Transformer models, particularly in the Attention and Feed-Forward layers.

Additionally, the significant proportion of time spent in `vectorized_elementwise_kernel` suggests that element-wise operations like LayerNorm and activation functions also consume a notable portion of computation time. This indicates that kernel fragmentation is a non-negligible issue.

4.3 Ablation Study

An ablation study was conducted to assess the individual and combined contributions of the proposed strategies.Table 1 summarizes the results, reporting the average training time per epoch, test accuracy, relative speedup over the baseline, and the accuracy delta:

Strategy Combination	Avg Epoch Time (s)	Test Accuracy	Speedup	Accuracy Change
(1) Gradient Accumulation	363.66	0.9230	1.256	-0.0123
(2) Layer-wise Unfreezing	159.00	0.9333	2.872	-0.0020
(3) Attention Rewrite	332.00	0.9339	1.375	-0.0014
(1) + (2)	153.00	0.9249	2.984	-0.0104
(1) + (3)	237.00	0.9304	1.627	-0.0049
(2) + (3)	93.66	0.8859	4.876	-0.0494
(1) + (2) + (3)	90.00	0.8849	5.074	-0.0504

Table 1: Comparison of Different Strategy Combinations

Figure 3 illustrates the variations in training time and test accuracy across different combinations of optimization strategies, further highlighting the trade-off between training efficiency and model performance.

The results show that although the combined strategy ((1) + (2) + (3) introduces a modest drop in accuracy, it achieves over 5× training speedup, demonstrating strong practical value in scenarios such as large-scale pretraining, rapid convergence, or cost-sensitive industrial deployments.

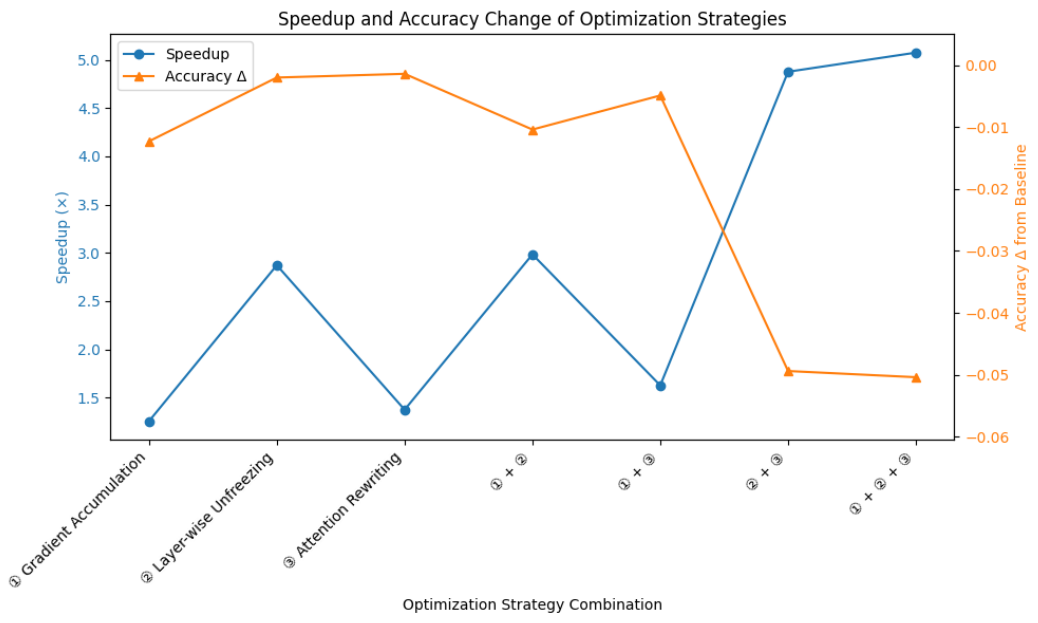

Figure 3: Speedup and Accuracy Change of Optimization Strategies

4.4 NSight Systems Kernel Analysis

To evaluate the low-level execution efficiency of the proposed optimization strategies, NVIDIA NSight Systems was used to analyze the BERT training process. The execution times of the first five training steps before and after optimization were compared. The optimized profiling trace is shown in Figure 4.
Notably, the execution time of the first step increased significantly after optimization, primarily due to the initial execution of torch.compile, which incurs overhead from graph capture and kernel compilation. However, starting from the second step, the overall execution time decreased substantially, indicating that the compilation optimizations had taken effect and delivered stable acceleration.

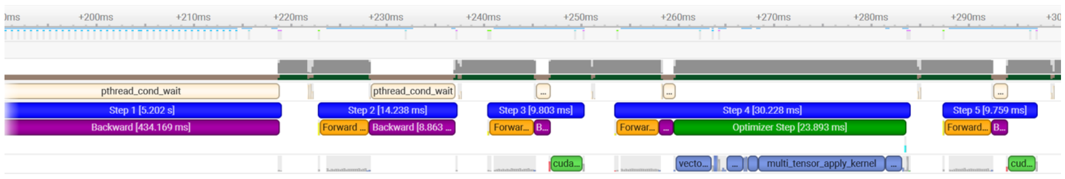

Figure 4: Nsight Systems profiling result after Optimization

Further analysis was conducted on the optimized operator execution, identifying the top-5 GPU kernels by runtime share:

- `vectorized_elementwise_kernel<FillFunctor<int>>` (75.0%)
- `triton_red_fused_dropout_layer_norm_backward` (7.3%)
- `cutlass_80_tensorop_f16_s16816gemm_relu_f16_128x128_32x4_tn_align8` (3.5%)
- `sm80_xmma_gemm_f16f16_f16f32_f32_tn_n_tilesize64x96x32` (2.8%)
- `sm80_xmma_gemm_f16f16_f16f32_f32_tn_n_tilesize160x128x32` (2.4%)

The top-1 kernel after optimization, `vectorized_elementwise_kernel<FillFunctor<int>>`, accounts for 75% of total execution time. This indicates that kernel fusion was highly effective, resulting in much higher computational concentration and a significant reduction in kernel fragmentation.

Additionally, several fused kernels generated by the Triton compiler were observed:

- `triton_red_fused_*`, `triton_poi_fused_*`, `triton_flash_*`

These results confirm that the attention module was successfully replaced with xFormers' `memory_efficient_attention`, which leverages low-level kernel fusion to substantially reduce fragmentation and improve execution efficiency.

5 Conclusion

This study used NVIDIA profiling tools to identify key bottlenecks in BERT training and design five targeted optimization strategies.

First, three strategies were evaluated individually and together through ablation experiments, showing their effects on training speed and accuracy. A fourth strategy using Triton to optimize the attention kernel also achieved significant speedups. The fifth strategy focused on parallel data loading to reduce I/O bottlenecks.

Combining the first three strategies achieved nearly a **5×** training speedup with only a slight decrease in accuracy, making it especially useful for large-scale pretraining and cost-sensitive tasks. Minor accuracy losses could be recovered later with fine-tuning.

Challenges included limited sequence length and conflicts between attention optimizations and parameter freezing. Adaptive fine-tuning and careful trade-off choices can help address these issues.

Overall, this work shows that profile-driven, targeted optimizations can efficiently scale large-model training under practical resource constraints.

References

[1] Abadi, M. et al. (2016) TensorFlow: Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 265–283.

[2] Li, S. et al. (2021) PyTorch Profiler: A Performance Analysis Tool. In *Proceedings of Machine Learning and Systems (MLSys)*, 3:1–15.

[3] Rasley, J. et al. (2020) DeepSpeed: System Optimizations Enable Training Deep Learning Models with 100B Parameters. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pp. 3505–3506.

[4] NVIDIA. (2022) Nsight Systems 2022.3 Documentation. Santa Clara, CA: NVIDIA Corporation.

[5] Huawei. (2023) Ascend Sprof White Paper. Shenzhen: Huawei Technologies Co., Ltd.

[6] Chen, T. et al. (2016) Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174*.

[7] Jain, A. et al. (2022) Dynamic Gradient Checkpointing for Billion-Parameter Models. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, pp. 1–15.

[8] Dao, T. et al. (2022) FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*, pp. 1–15.

[9] Rajbhandari, S. et al. (2021) ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC)*, pp. 1–15.

[10] Shoeybi, M. et al. (2020) Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*.

[11] Zheng, L. et al. (2022) Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 559–577.

[12] Tang, H. et al. (2021) 1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pp. 1–10.

[13] Vogels, T. et al. (2021) PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization. *Journal of Machine Learning Research*, 22(1):1–42.

[14] Patterson, D. et al. (2023) Carbon-API: Real-Time Carbon Footprint Monitoring for AI Training. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pp. 1–12.

[15] Dettmers, T. et al. (2023) LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, pp. 1–15.

[16] Micikevicius, P. et al. (2023) FP8 Formats for Deep Learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):1–15.

[17] Sabne, A. et al. (2022) XLA: Optimizing Compiler for Machine Learning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 1–15.

[18] Chen, T. et al. (2022) TVM AutoTensorization: Hardware-Aware Operator Optimization for Deep Learning. *Proceedings of the IEEE*, 110(5):1–20.