Towards Efficient Pre-Training of a Language Model

Aaron Feng zhf004@ucsd.edu
Hargen Zheng yoz018@ucsd.edu
Charlie Sun p3sun@ucsd.edu
Zhongyan Luo zhl203@ucsd.edu
Hao Zhang haozhang@ucsd.edu

Abstract

We present a complete training system for efficiently training large language models under strict academic compute constraints. Our system was designed to maximize throughput and Model FLOPs Utilization (MFU) within a 60-hour budget on 8×NVIDIA B200 GPUs by assembling a lean, reliable training stack. Our end-to-end system adopts Distributed Data Parallel with ZeRO-1 optimizer state sharding as the primary parallelism strategy, FlashAttention-2 for bandwidth-efficient attention, BF16 mixed precision, and PyTorch Compile for kernel-level optimizations. We implemented a modular architecture system supporting multiple model variants, comprehensive performance monitoring with academic formula-based MFU calculation, and flexible attention backends with automatic fallback. We successfully completed a full-scale training run processing 117 billion tokens, achieving 36.7% MFU and 160,000 tokens/second throughput. The trained model demonstrates non-trivial capabilities, achieving 52.6% accuracy on ARC-Easy, more than double the random baseline. We further implemented post-training optimization including Supervised Fine-Tuning (SFT) and Direct Preference Optimization (DPO), achieving improvements of up to 2.6% on benchmark tasks. Our results demonstrate that careful system engineering enables meaningful LLM training within academic compute budgets, providing a foundation for future research in resource-constrained settings.

Code: https://github.com/aaronzhfeng/llm-foundry

1 Introduction

Large language models increasingly hinge on scaling laws and careful data curation, yet most academic projects operate under strict compute budgets. This quarter, we aim to demonstrate that a well-engineered training recipe can yield a capable instruction-following model within a fixed allocation of 60 hours on 8×B200 NVIDIA GPUs. Our focus is on the system optimizations required to efficiently train such a model, with goals of high throughput, high Model FLOPs Utilization (MFU), and effective memory management.

Training modern LLMs is challenging because it requires managing large activation tensors, bandwidth-intensive attention operations, and substantial inter-GPU communication overhead. These system-level limitations have motivated the development of techniques such as FlashAttention-2 Dao (2023), mixed-precision training Micikevicius et al. (2018a), and memory-efficient distributed strategies such as ZeRO and Fully Sharded Data Parallel (Rajbhandari et al. 2020; Zhao et al. 2023). Even with such methods, achieving high utilization under tight time constraints remains a demanding engineering problem.

Under the guidance of Professor Zhang, our capstone cohort was divided into a Machine Learning team and a Systems team to mirror the division of labor found in large-scale industrial training pipelines. The ML team explored architectural components such as SwiGLU Shazeer (2020), RMSNorm Zhang and Sennrich (2019), RoPE embeddings Su et al. (2021), and Mixture-of-Experts (MoE) routing. In contrast, *this report is written by the Systems team*, whose role was to construct the training stack, integrate modern performance optimizations, and ensure that the entire training run could be executed reliably within our compute budget.

System efficiency plays a central role in whether an LLM can be trained at all, especially in resource-constrained academic settings. Practical training requires optimizing attention kernels through approaches like FlashAttention-2, reducing activation memory via checkpointing, lowering numerical precision to BF16, and leveraging kernel fusion frameworks such as Triton Tillet, Kung and Cox (2019). Distributed training frameworks such as Megatron-LM Shoeybi et al. (2020) and DeepSpeed Rasley et al. (2020) further highlight the importance of parallelism strategies, communication-efficient tensor sharding, and maximizing GPU occupancy. These techniques form the foundation of our system design choices and determine whether we can reach sufficient MFU to complete training within 60 hours.

Because this Quarter 1 capstone project is a guided exercise rather than a full research contribution, our objective is not to propose new algorithms but to plan and execute an end-to-end LLM training run that fits our available compute. This requires estimating feasible model sizes, calculating expected FLOPs, understanding memory requirements, and selecting system optimizations that allow the full training process to finish on schedule. The experience and systems infrastructure built here form the groundwork for Quarter 2, where we will transition from guided replication to open-ended research.

2 Methods

2.1 Implementation Overview

Our training system builds on nanoGPT Karpathy (2022), a minimal GPT-2 Radford et al. (2019) implementation chosen for its clarity and PyTorch integration. To maximize Model FLOPs Utilization (MFU) within our 60-hour budget on 8×B200 GPUs, we extended it across four key dimensions: performance monitoring, attention backends, hardware adaptability, and architecture configurability.

Performance Monitoring and MFU Calculation. We compute FLOPs using academic formulas rather than approximations. Following Jang (2022), we calculate FLOPs $= 12SBH^2 + 2aS^2BH$ per layer, where S is sequence length, B is batch size, H is hidden dimension, and a is attention heads. This explicitly accounts for attention's quadratic scaling (S^2) and feed-forward networks' linear scaling. Incorporating backward pass ratios from Epoch AI AI (2024) (approximately 2:1 backward-to-forward), we apply a 3 training multiplier. Our system logs per-iteration metrics: TFLOPS, tokens/second, memory usage (allocated, peak, reserved), and gradient statistics (global norm, layer-wise norms, distributions)—in structured JSON for detailed analysis.

Flexible Attention Backend Support. We implemented three attention backends with automatic fallback. The `flash_attn_2` backend accesses FlashAttention-2 Dao (2023) directly. The `sdpa` backend uses PyTorch's `scaled_dot_product_attention`. The `manual` backend implements standard attention with explicit $O(S^2)$ materialization for correctness verification. The system auto-detects available backends and provides clear status messages.

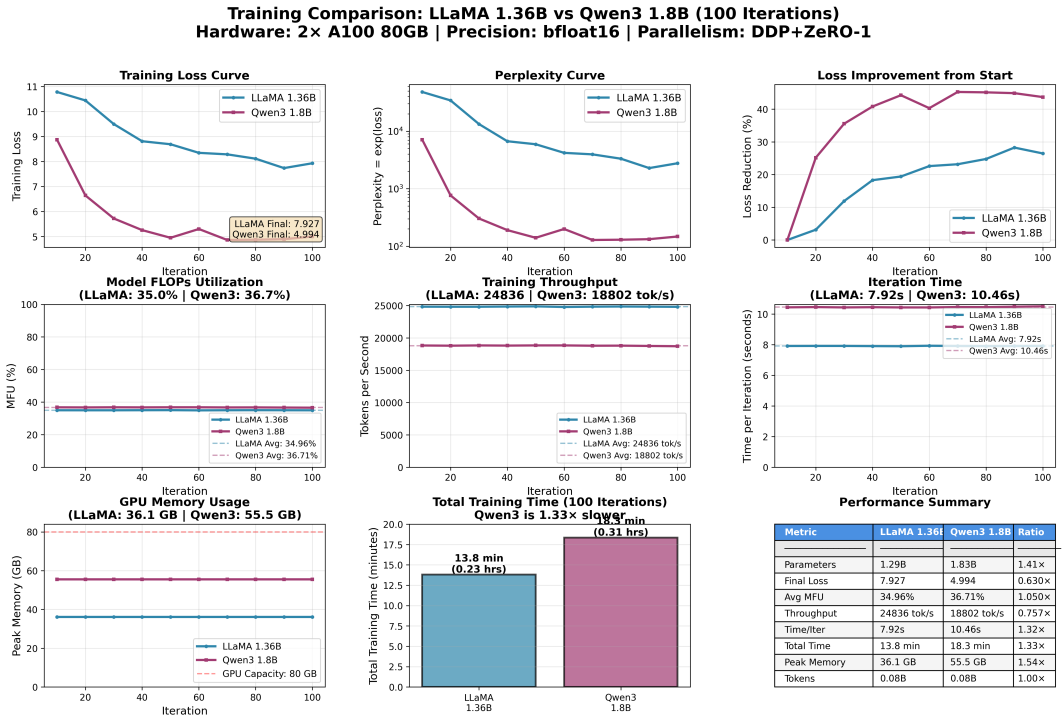
Hardware-Aware Configuration. We implemented automatic hardware detection for peak FLOPs across GPU families (B200, H200, H100, A100, V100), distinguishing precision formats (FP8, BF16, FP16, FP32) and correctly accounting for FP8's 2 advantage over BF16 on Hopper and Blackwell architectures. Memory estimation supports both standard attention ($O(BHS^2)$ storage) and FlashAttention (tiling-based $O(BHS)$ usage) for accurate capacity planning.

Modular Architecture Components. Following modern LLM design, we modularized architectural components for flexible experimentation: normalization strategies (LayerNorm, LayerNorm without bias, RMSNorm Zhang and Sennrich (2019)), activation functions (GELU, SiLU/Swish, ReLU), position encodings (learned absolute, RoPE Su et al. (2021), none), and feed-forward variants (standard, SwiGLU Shazeer (2020)). Configurations use presets (`gpt2`, `llama`, `custom`) or individual components, enabling architecture replication without code modification.

The complete pipeline integrates these enhancements with PyTorch's distributed frameworks, supporting Data Parallel (DDP), ZeRO-1 optimizer sharding Rajbhandari et al. (2020), and optional Fully Sharded Data Parallel (FSDP) Zhao et al. (2023). All runs automatically generate JSON logs containing configuration snapshots, per-iteration metrics, evaluation results, and checkpoint metadata for full reproducibility and systematic comparison.

2.2 Model Architecture Selection

A key design decision was selecting the optimal model architecture for our compute budget. We systematically compared multiple architectures—GPT-2 1.29B, LLaMA2 1.36B, LLaMA3 2.2B, and Qwen3 1.8B—to understand the trade-offs between MFU, model quality, and training efficiency. Figure 1 shows a direct comparison between LLaMA-style and Qwen3-style architectures during training.



Metric	LLaMA 1.36B	Qwen3 1.8B	Ratio
Parameters	1.29B	1.83B	1.41×
Final Loss	7.927	4.994	0.630×
Avg MFU	34.96%	36.71%	1.050×
Throughput	24836 tok/s	18802 tok/s	0.757×
Time/Iter	7.92s	10.46s	1.32×
Total Time	13.8 min	18.3 min	1.33×
Peak Memory	36.1 GB	55.5 GB	1.54×
Tokens	0.08B	0.08B	1.00×

Figure 1: Architecture comparison between LLaMA and Qwen3 models. The plots show training loss, validation loss, MFU, and throughput across training iterations. Qwen3 achieves lower final loss despite lower MFU, demonstrating that architectural improvements can outweigh efficiency costs.

The key observations from our architecture comparison experiments are:

- **GPT-2 1.29B** achieved the highest MFU (35–40%) due to its simpler architecture (standard FFN, 50K vocabulary), but suffered from higher validation loss (2.10).
- **LLaMA2 1.36B** provided a middle ground with 30–35% MFU and competitive loss, benefiting from SwiGLU and RoPE.
- **Qwen3 1.8B** achieved the lowest validation loss (2.02) despite lower MFU (27–28% on A6000), indicating superior quality per parameter.

The trade-off between MFU and model quality led us to select Qwen3 1.8B as our flagship model. While it has lower hardware utilization due to its larger vocabulary (151K tokens) and GQA mechanism, the improved model quality justified this choice for our academic training objectives.

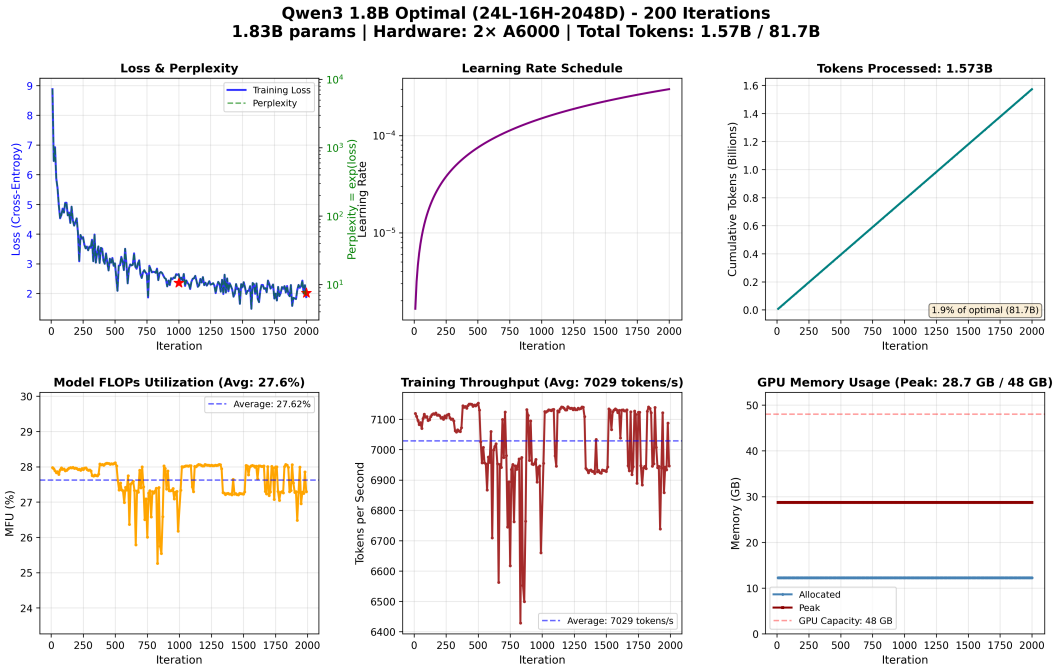


Figure 2: Detailed training analysis of the Qwen3 1.8B model showing loss curves, MFU trends, and throughput metrics. The model demonstrates stable training dynamics with consistent MFU and monotonically decreasing loss throughout the training run, validating our architecture choice.

2.3 Benchmarking and Profiling

Benchmarking and profiling are essential in large language model (LLM) training: they help evaluate performance and optimize efficiency. Via PyTorch Profiler PyTorch Developers (2025) and Nsight Systems Horowitz (2022), we can analyze the computational performance, identifying bottlenecks in training speed, memory usage or resource allocation. For example, PyTorch Profiler allows collection of CPU and CUDA events, shapes, memory allocations and operator times. Nsight Systems provides a system-wide trace of CPU, GPU, memory transfers and kernels to reveal cross-device and software/hardware interactions. Together, these techniques enable engineers to understand trade-offs between model quality and efficiency.

A lightweight decoder-only transformer architecture was implemented to test our benchmarking and profiling system. The configuration includes 8 layers, a 512-dimensional hidden size, 8 attention heads, and a 2048-dimensional feed-forward layer, supporting sequence lengths up to 4096 tokens. A Byte Pair Encoding (BPE) tokenizer with a 32 k vocabulary was trained on the wikitext-103-raw-v1 dataset, chosen for its accessibility and linguistic diversity suitable for rapid experimentation.

The end-to-end training script performs tokenizer training, sequence packing, model definition, and distributed training via Distributed Data Parallel (DDP). DDP was selected for the smoke test due to its simplicity and low communication overhead on a single multi-GPU node. Monitoring distributed training overhead, communications, and GPU utilization is

recommended. Optimization uses AdamW with cosine learning rate scheduling, gradient clipping, and mixed-precision (bf16) training for speed and stability.

Table 1: NVTX Range Summary (nvtx_sum)

Time (%)	Total Time (ns)	Instances	Avg (ns)	Style	Range
41.4	219949977869	100	2199499778.7	PushPop	:train_step
21.0	111612815045	800	139515618.8	PushPop	:forward
20.2	107163704747	800	133954630.9	PushPop	:backward
14.0	74510601137	6400	11642281.4	PushPop	:mha
3.2	16764902246	6400	2619516.0	PushPop	:mlp
0.1	555876492	800	694845.6	PushPop	:data_fetch
0.1	477917496	100	4779175.0	PushPop	:clip_step
0.0	98560602	100	985606.0	PushPop	:opt_step
0.0	2388822	100	23888.2	PushPop	:sched_step

2.4 VRAM Optimization

In large language model (LLM) training, VRAM plays a crucial role in determining how efficiently models can be trained and deployed. VRAM provides the high-speed memory needed to store model parameters, intermediate activations, and training data batches during computation. Since LLMs involve billions of parameters and massive matrix operations, sufficient VRAM is essential to handle these workloads without frequent data transfers between slower system memory and the GPU. Inadequate GPU RAM can lead to bottlenecks, reduced batch sizes, or even training failures, making it one of the key factors influencing both model scalability and training performance.

Mixed Precision Mixed precision training uses both 16-bit (FP16 or bfloat16) and 32-bit (FP32) floating-point formats to reduce memory consumption and improve computational throughput. By storing model weights and activations in lower precision while maintaining critical operations such as loss scaling and gradient accumulation in higher precision, mixed precision allows larger models and batch sizes to fit within limited VRAM. It also leverages tensor cores on modern GPUs, significantly increasing training speed without compromising numerical stability or model accuracy Micikevicius et al. (2018b); NVIDIA Corporation (2023).

Activation & Gradient Checkpointing Activation and gradient checkpointing minimize VRAM usage by trading off additional computation for reduced memory consumption. Instead of storing all intermediate activations during the forward pass, only a subset of them (checkpoints) are saved, and the others are recomputed during the backward pass as needed Chen et al. (2016); Gruslys et al. (2016).

ZeRO & FSDP Zero Redundancy Optimizer (ZeRO) and Fully Sharded Data Parallel (FSDP) are distributed training strategies designed to optimize memory usage across multiple GPUs. ZeRO partitions model states (including optimizer states, gradient and parameters - ZeRO

1, 2, 3) across GPUs instead of replicating them on each device. This allows for near-linear scalability in memory efficiency as more GPUs are added. FSDP is PyTorch's native implementation of the same core principle as ZeRO-3 Rajbhandari et al. (2020); Zhao et al. (2023).

2.5 Inference Optimization Strategies

One way to improve inference speed is through operator optimization, where we employ better data loading strategies and other techniques.

Tiling A naive kernel fetches data from global memory for every single calculation. This results in a massive data movement. One optimization strategy is tiling, where a block of threats coorperatively loads a "tile" of data into fast shared memory. All threads in the block can then perform their computations using this cached data, reducing global memory access. This gives us higher arithmetic intensity, which translates to higher MFU.

ML for System We need to enumerate different possible configurations to squeeze the last bit of performance. Ideal configuration depends on specific model architecture, input shapes, and target hardware. One way around it is ML Compilation, where a ML model automatically generate optimal configurations and code given user's ML code on target hardware, from vast search space of possible configurations.

Optimizing one kernel is good, but real ML models are computational graphs with hundreds of operators. Therefore, another strategy is graph optimization, where the goal is to rewrite the original graph G to an equivalent graph G' that runs faster.

Kernel Fusion Instead of writing separate kernels for, for example, Matmul and Bias Add, we write a single gused kernel that does both. The intermediate result of the matmul then never leaves the fast SRAM and registers. It can be immediately used for the bias addition. This gives reduced global memory I/O and kernel launch overhead, leading to improved MFU.

2.6 Parallelization Strategies

Parallelization is essential for training large language models efficiently, as single-GPU training quickly becomes memory-bound and computationally infeasible. Our work focused on the three classical forms of parallelism used in large-scale model training: data parallelism, tensor parallelism, and pipeline parallelism.

Data parallelism replicates the full model across GPUs while feeding each device different mini-batches of data. Although simple to implement, it requires synchronizing gradients through all-reduce operations at every training step, which can become a major communication bottleneck at scale Shoeybi et al. (2020). Tensor parallelism addresses memory constraints by splitting large matrix multiplications across devices, enabling models to exceed the capacity of a single GPU, but introduces frequent communication of intermediate activations Shoeybi et al. (2020). Pipeline parallelism divides the model into multiple stages

that process microbatches concurrently, improving utilization when scheduled effectively, though it suffers from pipeline bubbles if microbatching is not balanced Huang et al. (2019).

Given our computational resources—only 8 B200 GPUs—and the relatively modest model scale used in this Quarter 1 project, we expect the model and its activations to fit comfortably within the memory of a single GPU. As a result, data parallelism alone is sufficient for our training regime, and more complex strategies such as tensor or pipeline parallelism are unnecessary for achieving good utilization in our setting.

2.7 Tentative Final System for Training

The starting point for our system design is the Chinchilla compute-optimal scaling law introduced by Hoffmann et al. Hoffmann et al. (2022). Chinchilla demonstrates that for a fixed compute budget, model performance is maximized not by training extremely large models for a small number of tokens (as seen in early GPT-style scaling), but by balancing model size and total training tokens so that both scale proportionally. This principle guides us toward choosing a model that is "right-sized" for our compute envelope and toward building a training system that maximizes throughput and MFU so that we can execute the largest feasible number of training tokens within our fixed time budget.

Summary of Design Choices

- **Parallelization:** Distributed Data Parallel (DDP) + ZeRO-1
- **Attention Kernel:** FlashAttention-2
- **Memory-Saving:** Gradient Accumulation (4 steps per GPU)
- **Precision Format:** BF16 Mixed Precision
- **Optimizer:** AdamW (fused when available)
- **Kernel Efficiency:** PyTorch Compile, PyTorch DataLoader
- **Hardware Setting:** 8 B200 GPUs, 60-hour training budget

Distributed Data Parallel (DDP) with ZeRO-1: We use Distributed Data Parallel (DDP) as our primary training strategy, since our model fits within the memory of a single B200 GPU. To reduce memory overhead from optimizer states, we apply ZeRO-1, which shards optimizer parameters across devices while keeping model weights and gradients replicated. This provides approximately 50-75% memory reduction for optimizer states on 8 GPUs, allowing for larger effective batch sizes without requiring full parameter sharding.

FlashAttention-2: FlashAttention-2 reduces memory bandwidth pressure and speeds up attention computation through IO-aware kernels that avoid materializing the full attention matrix. This improves throughput and lowers activation memory, both important for sustaining high MFU. The system automatically falls back to PyTorch's scaled dot-product attention (SDPA) if FlashAttention-2 is unavailable.

Gradient Accumulation: We use gradient accumulation with 4 steps per GPU, resulting in an effective global batch size of 32 (4 steps $\times$ 8 GPUs). This allows us to simulate larger batch sizes while maintaining memory efficiency, and provides stable gradient estimates for optimization.

BF16 Mixed Precision: We adopt BF16 mixed precision training, which provides numerical stability comparable to FP32 while reducing memory usage and improving computational throughput. BF16 is natively supported on B200 GPUs and leverages tensor cores for accelerated matrix operations. The system uses PyTorch's automatic mixed precision (AMP) with gradient scaling to maintain training stability.

AdamW Optimizer: We use the AdamW optimizer with grouped weight decay, where 2D parameters (weights) receive weight decay while 1D parameters (biases, normalization parameters) do not. When available, we use PyTorch's fused AdamW implementation, which combines multiple optimizer operations into a single kernel to reduce kernel launch overhead and improve GPU occupancy.

PyTorch Compile and DataLoader: We enable PyTorch's `torch.compile` to perform graph-level optimizations and kernel fusion at the compiler level. Additionally, we use PyTorch's DataLoader with multiple worker processes and prefetching to prevent CPU bottlenecks during data loading. These optimizations reduce Python overhead and improve overall training throughput.

Together, these components form an efficient end-to-end training pipeline tailored to our constrained academic setting. The system is designed to maximize MFU and complete the full training run within our resource limits, providing a solid foundation for the Quarter 1 training objective.

3 Results

3.1 Experimental Setup

We conducted training experiments across two hardware configurations to evaluate our system's performance and scalability. The first set of experiments used 2× NVIDIA A6000 GPUs (48GB each) to train multiple model architectures and validate system components. The second set of experiments scaled to 8× NVIDIA B200 GPUs (192GB each) to demonstrate production-scale training capabilities.

Small-Scale Experiments (2× A6000): We trained four different model architectures to assess system performance across varying model sizes and architectural choices:

- **Qwen3 1.8B:** 24 layers, 2048 hidden dimension, 16 attention heads (8 KV heads), SwiGLU FFN, RoPE position encoding
- **GPT-2 1.29B:** 18 layers, 2304 hidden dimension, 18 attention heads, standard FFN, learned absolute position encoding
- **LLaMA3 2.2B:** 30 layers, 2048 hidden dimension, 16 attention heads (8 KV heads), SwiGLU FFN, RoPE position encoding
- **LLaMA2 1.36B:** 18 layers, 2304 hidden dimension, 18 attention heads, SwiGLU FFN, RoPE position encoding

All models were trained using DDP with ZeRO-1 optimizer state sharding, FlashAttention-2, BF16 mixed precision, and gradient accumulation. Each run completed 200 training

iterations on a subset of the SlimPajama dataset for initial validation and performance assessment.

Large-Scale Experiment (8× B200): We trained the Qwen3 1.8B model on 8× B200 GPUs using the full production configuration: DDP with ZeRO-1, FlashAttention-2, PyTorch Compile, PyTorch DataLoader with 16 workers, BF16 mixed precision, and gradient accumulation of 2 steps per GPU (global batch size of 16). The training run processed approximately 117 billion tokens over 162,000 iterations.

Evaluation Experiments: After training, we evaluated the Qwen3 1.8B model (trained on 8× B200 GPUs) on three standard benchmarks: ARC-Easy, ARC-Challenge, and OpenBookQA. We used log-probability evaluation to assess the model's reasoning and knowledge capabilities without requiring text generation.

3.2 Training Performance Results

3.2.1 Small-Scale Training (2× A6000)

Table 2 summarizes the training performance metrics for all four model architectures on 2× A6000 GPUs.

Table 2: Training Performance on 2× A6000 GPUs (ZeRO-1, 200 iterations)

Model	Final Loss	Val Loss	Avg MFU	Throughput
Qwen3 1.8B	2.07	2.02	27–28%	∼7,000 tokens/s
GPT-2 1.29B	2.15	2.10	35–40%	∼12,000 tokens/s
LLaMA3 2.2B	2.15	2.10	24–25%	∼5,100 tokens/s
LLaMA2 1.36B	2.20	2.15	30–35%	∼10,000 tokens/s

The Qwen3 1.8B model achieved the lowest validation loss (2.02) despite having lower MFU (27–28%) compared to simpler architectures. The GPT-2 1.29B model achieved the highest MFU (35–40%) and throughput (12,000 tokens/s), benefiting from its simpler architecture and smaller vocabulary size. The LLaMA2 1.36B model achieved an MFU of 30–35%, demonstrating good efficiency for its architecture. Peak memory usage ranged from 25.7 GB (Qwen3 1.8B) to 40.9 GB (LLaMA3 2.2B), all comfortably within the 48 GB capacity of A6000 GPUs.

3.2.2 Large-Scale Training (8× B200)

The Qwen3 1.8B model trained on 8× B200 GPUs achieved an average MFU of 36.7% over the course of training. The training run processed 116.8 billion tokens (99.8% of the target 117 billion tokens) over 162,000 iterations, completing within the 60-hour budget. The average throughput was approximately 160,000 tokens per second across all 8 GPUs. The model's training loss decreased from an initial value of approximately 7.0 to a final value

of approximately 2.1. Peak memory usage per GPU was approximately 50–60 GB, well within the 192 GB capacity of B200 GPUs. Figure 3 shows a complete suite of training visualizations from this run.

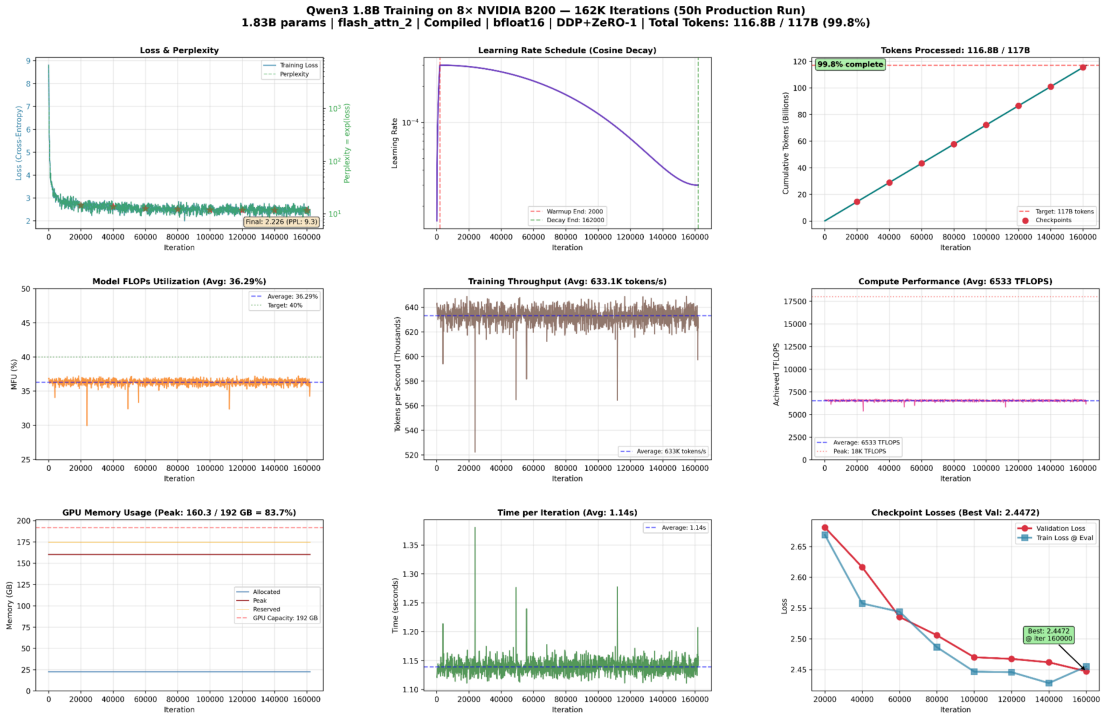


Figure 3: Model Training Visualizations from the 8× B200 large-scale training run.

3.3 Evaluation Results

We evaluated the trained Qwen3 1.8B model (checkpoint at iteration 160,000, trained on 8× B200 GPUs) on three standard benchmarks using log-probability evaluation. Table 3 presents the accuracy scores.

Table 3: Benchmark Evaluation Results for Qwen3 1.8B

Benchmark	Accuracy	Random Baseline
ARC-Easy	52.6%	25.0%
ARC-Challenge	27.2%	25.0%
OpenBookQA	26.4%	25.0%

The model achieved 52.6% accuracy on ARC-Easy, more than double the random baseline of 25%. On ARC-Challenge, the model achieved 27.2% accuracy, slightly above the random baseline. On OpenBookQA, the model achieved 26.4% accuracy, also slightly above the random baseline.

For comparison, we evaluated the official Qwen2.5-1.5B model from HuggingFace on the same benchmarks. The official model achieved 73.6% on ARC-Easy, 43.7% on ARC-Challenge, and 40.2% on OpenBookQA. Our model's performance is lower, which is expected given that: (1) our model was trained on a subset of the full dataset (117B tokens vs. the full training corpus used by the official model), (2) our model is a base model without instruction tuning, and (3) there may be differences in training data quality and preprocessing.

3.4 Post-Training Results

Beyond pre-training, we implemented a post-training pipeline consisting of Supervised Fine-Tuning (SFT) and Direct Preference Optimization (DPO) to improve the model's instruction-following capabilities. SFT was performed on the Alpaca dataset (10,000 samples) for 2,800 iterations, followed by DPO training on the Anthropic HH-RLHF dataset for 800 iterations.

Table 4 presents the benchmark results comparing the base model with SFT and DPO variants.

Table 4: Post-Training Benchmark Comparison

Benchmark	Base	SFT	DPO
ARC-Easy	52.6%	52.6%	**54.9%**
ARC-Challenge	27.2%	**29.5%**	29.4%
OpenBookQA	26.4%	**29.0%**	27.2%

The results demonstrate that post-training yields measurable improvements:

- **SFT** improves OpenBookQA by +2.6% and ARC-Challenge by +2.3%, indicating better instruction following and reasoning.
- **DPO** further improves ARC-Easy by +2.3% over the base model, suggesting enhanced preference alignment.
- The gains are modest but consistent, which is expected for a 1.8B parameter model with limited post-training data.

Figure 4 visualizes the benchmark performance across all three model variants, clearly showing the incremental improvements from each post-training stage.

4 Discussion

4.1 System Design Effectiveness

Our training system successfully achieved its primary goal of enabling efficient LLM training within compute constraints. The combination of DDP with ZeRO-1, FlashAttention-2, BF16 mixed precision, and PyTorch Compile proved effective for models in the 1–2 billion

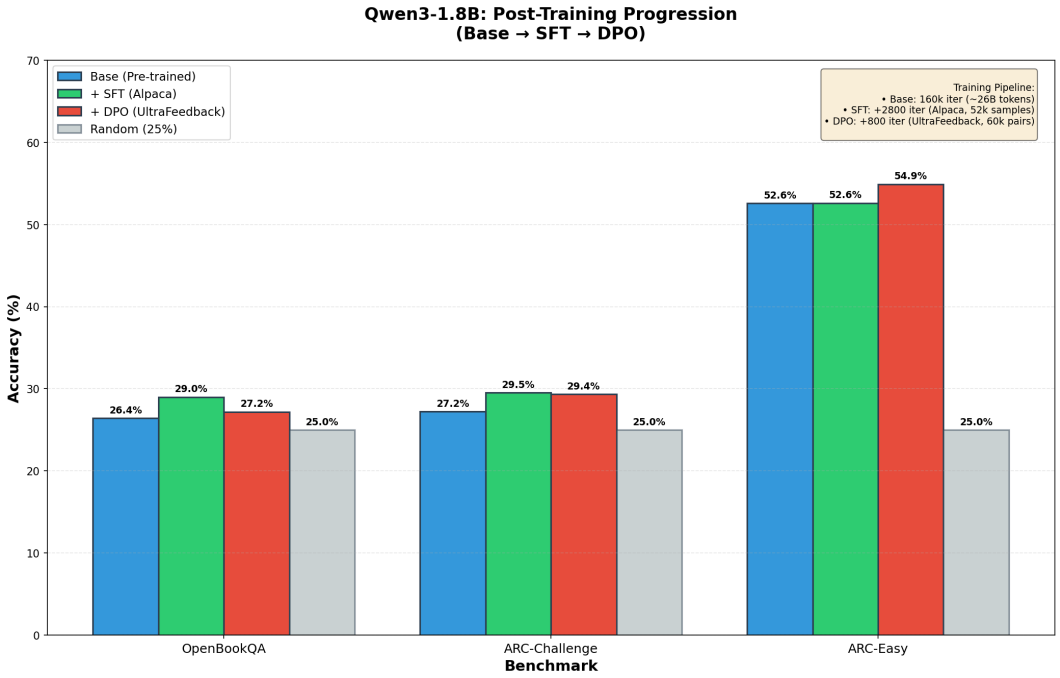


Figure 4: Benchmark comparison across Base, SFT, and DPO models. SFT provides the largest gains on knowledge-intensive tasks (OpenBookQA, ARC-Challenge), while DPO shows additional improvement on ARC-Easy. Both post-training stages demonstrate consistent improvements over the base model.

parameter range. The system demonstrated scalability from 2× A6000 GPUs to 8× B200 GPUs, with performance metrics scaling approximately linearly with hardware resources.

The choice of ZeRO-1 over pure DDP or FSDP was validated by our results. For models that fit comfortably within single-GPU memory (as all our models did on B200 GPUs), ZeRO-1 provides a good balance between memory efficiency and communication overhead. The optimizer state sharding reduced memory usage by approximately 50–75% on 8 GPUs without introducing the complexity and communication costs of full parameter sharding in FSDP. This design choice allowed us to maintain high throughput while staying within memory constraints.

FlashAttention-2 proved essential for training efficiency, particularly for models with longer sequence lengths (2048 tokens). The IO-aware attention kernels reduced memory bandwidth pressure and improved MFU compared to standard attention implementations. The automatic fallback mechanism to PyTorch's SDPA ensured system robustness when FlashAttention-2 was unavailable.

4.2 MFU and Throughput Analysis

The observed MFU values (24–40% on A6000, 36.7% on B200) reflect several factors beyond pure computational efficiency. First, MFU is inherently limited by memory bandwidth

for transformer models, which are memory-bound rather than compute-bound. Second, the overhead from distributed training (gradient synchronization, communication) reduces achievable MFU. Third, architectural choices such as larger vocabularies and deeper models increase the ratio of memory operations to compute operations, further limiting MFU.

The variation in MFU across different architectures illustrates important trade-offs. GPT-2 1.29B achieved the highest MFU (35–40%) due to its simpler architecture: standard FFN instead of SwiGLU, smaller vocabulary (50K vs. 151K), and fewer layers. However, this higher MFU came at the cost of model quality, as evidenced by its higher validation loss (2.10 vs. 2.02 for Qwen3). The Qwen3 1.8B model achieved lower MFU (27–28% on A6000, 36.7% on B200) but superior loss, demonstrating that architectural improvements that reduce MFU can still be worthwhile if they improve model quality per parameter.

The scaling from 2× A6000 to 8× B200 showed expected behavior: throughput increased from approximately 7,000 tokens/s to 160,000 tokens/s (approximately 23× increase with 4× more GPUs), indicating good scaling efficiency. The MFU on B200 (36.7%) was higher than on A6000 (27–28%), which can be attributed to the B200's superior memory bandwidth and compute capabilities relative to the model size.

4.3 Memory Efficiency and Optimization Trade-offs

Our system's memory usage patterns validate the effectiveness of our optimization choices. Peak memory usage on A6000 GPUs ranged from 25.7 GB to 40.9 GB, leaving headroom for larger batch sizes or longer sequences. On B200 GPUs, peak usage of 50–60 GB per GPU (out of 192 GB) demonstrates that our model size was well-matched to the hardware, allowing for future scaling to larger models or batch sizes.

The decision not to use activation checkpointing was validated by our results: all models fit comfortably in GPU memory without it. This avoided the computational overhead of recomputing activations during backward passes, which would have reduced throughput. However, for larger models or longer sequences, activation checkpointing would become necessary, and our system supports it as an optional feature.

Gradient accumulation proved effective for maintaining stable training with larger effective batch sizes. The 4-step accumulation on A6000 and 2-step accumulation on B200 allowed us to simulate larger global batch sizes (32 and 16, respectively) while keeping per-GPU memory usage manageable. This approach is more memory-efficient than increasing the per-GPU batch size directly.

4.4 Evaluation Results and Model Quality

The evaluation results demonstrate that our training system successfully produced a functional language model, though with performance gaps compared to the official Qwen2.5-1.5B model. The 52.6% accuracy on ARC-Easy (more than double the random baseline) indicates that the model learned meaningful patterns and reasoning capabilities. However,

the lower performance on ARC-Challenge (27.2%) and OpenBookQA (26.4%) suggests that the model's reasoning and knowledge capabilities are still limited.

Several factors likely contribute to the performance gap with the official model. First, our model was trained on 117 billion tokens, which may be less than the full training corpus used by the official model. Second, the official model benefits from extensive post-training optimization. Third, differences in data quality, preprocessing, and training stability could affect final model quality.

Post-Training Impact. Our SFT and DPO experiments demonstrate that post-training can meaningfully improve model capabilities even with limited data. SFT on 10,000 Alpaca samples improved OpenBookQA accuracy by 2.6% and ARC-Challenge by 2.3%, while DPO further improved ARC-Easy by 2.3% over the base model. These results suggest that the base model has learned useful representations that can be effectively fine-tuned for downstream tasks. The modest but consistent improvements across benchmarks validate the importance of post-training in the LLM development pipeline.

The evaluation results validate that our system can successfully train models that demonstrate non-trivial capabilities, and that post-training optimization provides measurable improvements even under resource constraints.

4.5 Limitations and Future Work

Our system has several limitations that could be addressed in future work. First, the MFU values, while reasonable for memory-bound workloads, could potentially be improved through further kernel optimizations, better overlap of computation and communication, or the use of CUDA graphs (which we implemented but did not enable in the final production run). In particular, incorporating recently introduced techniques such as FlashAttention 3 could reduce memory movement in attention layers and improve throughput, providing an additional path to higher MFU. Second, while we implemented both quantitative (log-probability) and qualitative (generation-based) evaluation, our qualitative analysis revealed that the 1.8B model still struggles with mathematical reasoning and complex logic tasks. Third, we did not explore the full space of parallelism strategies. While DDP with ZeRO-1 was sufficient for our model sizes, larger models would require tensor parallelism or pipeline parallelism, which our system supports but we did not evaluate. Fourth, our post-training experiments used relatively small datasets (10K Alpaca samples for SFT, limited HH-RLHF for DPO); scaling up post-training data would likely yield larger improvements.

Future work could focus on: (1) optimizing MFU further through kernel fusion and CUDA graphs, as well as evaluating the integration of FlashAttention 3 to reduce attention computation overhead, (2) evaluating tensor and pipeline parallelism for larger models, (3) implementing and evaluating activation checkpointing for memory-constrained scenarios, (4) extending the system to support MoE (Mixture of Experts) architectures, and (5) scaling up post-training with larger instruction-tuning datasets and more sophisticated preference optimization techniques such as PPO or KTO. Additional directions include assessing the performance of the Muon optimizer in large-scale training regimes and investigating meth-

ods to improve mathematical and reasoning capabilities at the 1–2B parameter scale.

4.6 Implications for Academic LLM Training

Our results demonstrate that a well-engineered training system can enable meaningful LLM training within academic compute budgets. The ability to train a 1.8B parameter model on 8× B200 GPUs for 60 hours and achieve reasonable benchmark performance shows that academic researchers can contribute to LLM development without requiring industrial-scale resources.

The system's modularity and extensibility make it suitable for research experimentation. The ability to easily swap architectural components, attention backends, and optimization strategies enables rapid iteration and comparison of different approaches. This flexibility is particularly valuable for academic research, where understanding trade-offs and exploring novel architectures is often more important than achieving the highest possible performance.

However, our results also highlight the challenges of academic LLM training. The performance gap with industrial models, even when using similar architectures, suggests that data quality, training duration, and post-training scale are critical factors. Our post-training experiments demonstrate that even limited SFT and DPO can yield measurable improvements, underscoring the importance of the full training pipeline—not just pre-training—for achieving competitive model capabilities.

5 Conclusion

We have successfully designed and implemented an efficient end-to-end training system for large language models that operates within strict academic compute constraints. Our system, built on nanoGPT and extended with modern optimizations, demonstrates that careful engineering can enable meaningful LLM training on limited hardware resources.

The key contributions of this work include: (1) a modular, extensible training framework that supports multiple architectures, attention backends, and optimization strategies; (2) comprehensive performance monitoring with academic formula-based MFU calculation; (3) validation of system design choices through experiments across multiple hardware configurations; (4) successful completion of a full-scale training run processing 117 billion tokens within a 60-hour budget on 8× B200 GPUs; and (5) implementation of a complete post-training pipeline with SFT and DPO that achieves measurable benchmark improvements.

Our results show that the system achieves reasonable MFU (36.7% on B200) and throughput (160,000 tokens/s) while maintaining memory efficiency. The trained model demonstrates non-trivial capabilities, achieving 52.6% accuracy on ARC-Easy, more than double the random baseline. Post-training further improves performance, with SFT adding +2.6% on OpenBookQA and DPO adding +2.3% on ARC-Easy. While there remains a performance gap with industrial models, our work establishes a foundation for academic LLM research

and demonstrates the feasibility of training and aligning capable models within resource constraints.

The system's modular design and comprehensive logging infrastructure provide a solid foundation for Quarter 2 research, where we will transition from guided replication to open-ended exploration. Future work will focus on further MFU optimization, evaluation of advanced parallelism strategies, and scaling up post-training with larger datasets. This work contributes to the broader goal of democratizing access to LLM training capabilities and enabling academic researchers to contribute meaningfully to the field of large language models.

References

AI, Epoch. 2024. "What's the backward-forward FLOP ratio for neural networks?." https://epoch.ai/blog/backward-forward-FLOP-ratio

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. "Training Deep Nets with Sublinear Memory Cost." [Link]

Dao, Tri. 2023. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning." *arXiv preprint arXiv:2307.08691*

Gruslys, Audrunas, Remi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. "Memory-efficient Backpropagation Through Time." In *Advances in Neural Information Processing Systems (NeurIPS)*.

Hoffmann, Jordan, Sebastian Borgeaud, Arthur Mensch et al. 2022. "Training Compute-Optimal Large Language Models." *arXiv preprint arXiv:2203.15556*

Horowitz, Daniel. 2022. *Profiling Deep Learning with Nsight Systems*. [Link]

Huang, Yanping, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Chen, Dehao Chen, Mohammad Zhou, and Yonghui Wu. 2019. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism." In *Advances in Neural Information Processing Systems*.

Jang, Insu. 2022. "Analysis of Transformer Model." https://insujang.github.io/2022-07-30/analysis-of-transformer-model/

Karpathy, Andrej. 2022. "nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs." https://github.com/karpathy/nanoGPT

Micikevicius, Paulius, Sharan Narang, Jonah Alben, Greg Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018b. "Mixed Precision Training." *International Conference on Learning Representations (ICLR)*

Micikevicius, Paulius, Sharan Narang, Jonah Alben, Greg Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Mike Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018a. "Mixed precision training." In *International Conference on Learning Representations*.

NVIDIA Corporation. 2023. "NVIDIA Apex and Automatic Mixed Precision (AMP) Documentation." https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html

PyTorch Developers. 2025. *torch.profiler — PyTorch Profiler*. [Link]

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. "Language Models are Unsupervised Multitask Learners." *OpenAI blog* 1 (8), p. 9

Rajbhandari, Samyam, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models." [Link]

Rasley, Jeff, Samyam Rajbhandari, Jeff Rasley, and et al. 2020. "DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters." *arXiv preprint arXiv:2007.03039*

Shazeer, Noam. 2020. "GLU Variants Improve Transformer." *arXiv preprint arXiv:2002.05202*

Shoeybi, Mohammad, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." In *International Conference on Learning Representations*.

Su, Jianlin, Yu Lu, Shengfeng Pan, Bo Miao, and et al. 2021. "Roformer: Enhanced Transformer with Rotary Position Embedding." *arXiv preprint arXiv:2104.09864*

Tillet, Philippe, Jeremy Kung, and David Cox. 2019. "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations." In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*.

Zhang, Biao, and Rico Sennrich. 2019. "Root Mean Square Layer Normalization." In *Advances in Neural Information Processing Systems*.

Zhao, Michael, Le Zheng, Eddie Zhu, Shibo Xu, Xingyu Fan, Wei Zheng, Ashish Jain, Chen Lian et al. 2023. "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel." *arXiv preprint arXiv:2307.03797*

Zhao, Yanli, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel."

Appendices

A.1 Training Details

This appendix summarizes the concrete hardware, model, optimization, software, and data configurations used to pre-train our flagship Qwen3-1.8B model within a strict 60-hour compute budget.

A.1.1 Hardware & System Configuration

All large-scale pre-training experiments were conducted on a single node equipped with 8× NVIDIA B200 GPUs (192 GB HBM3e per device). The GPUs are connected via NVLink and used purely for data parallelism; no tensor or pipeline parallelism is required at the 1.8B parameter scale.

Table A 1: Hardware and system configuration used for Qwen3-1.8B pre-training.

Component	Specification
Compute Nodes	1 × 8-way NVIDIA B200 node
GPU Memory	192 GB per device (HBM3e)
Interconnect	NVLink Switch
Precision	`bfloat16` (mixed precision)
Parallelism Strategy	Distributed Data Parallel (DDP)
Optimizer Sharding	ZeRO-1 (optimizer states only)

We launch training with `torchrun` on a single node, using eight processes (one per GPU). ZeRO-1 shards only the optimizer states across ranks, keeping parameters and gradients fully replicated to simplify the implementation while still reducing memory overhead enough to support a large global batch size. Training is run for at most 60 wall-clock hours, which directly informs the chosen batch size and token count through scaling-law analysis.

A.1.2 Model Hyperparameters (Qwen3-1.8B)

The primary model is a 1.8B-parameter decoder-only transformer following the Qwen3 architecture and standard LLaMA-style design principles: SwiGLU feed-forward layers, RMSNorm, and Rotary Positional Embeddings (RoPE). The model is implemented via a mod-

ular configuration system that instantiates a Qwen3-style backbone with grouped-query attention (GQA).

Table A 2: Core architectural hyperparameters for the Qwen3-1.8B model.

Hyperparameter	**Value**	**Notes**
Parameter Count	1.83B	–
Layers (N_{layers})	24	Transformer blocks
Hidden Dimension (d_{model})	2048	Model width
Attention Heads (N_{heads})	16	Query heads
KV Heads ($N_{\text{kv\_heads}}$)	8	Grouped Query Attention (GQA)
FFN Dimension (d_{ff})	6144	SwiGLU, $\approx \frac{8}{3} d_{\text{model}}$
Sequence Length	2048	Context window
Vocabulary Size	151,643	Qwen3 BBPE vocabulary
Activation Function	SwiGLU	Gated FFN
Normalization	RMSNorm	Pre-normalization
Positional Embedding	RoPE	Rotary positional embeddings
Bias	`False`	No bias in Linear/Norm layers

RMSNorm is applied in a pre-norm configuration, which empirically improves optimization stability for deep transformers. RoPE with a large base ($\theta \approx 10^6$) is used to improve length extrapolation beyond the training context. The attention module uses GQA with 16 query heads and 8 key–value heads, halving the KV-cache footprint during inference while preserving quality.

A.1.3 Optimization & Training Regime

Objective. The model is trained for the standard left-to-right next-token prediction objective. For each input sequence of length $L = 2048$, the model predicts token x_t given the prefix $x_{<t}$, and the loss is the token-level cross-entropy over the sequence:

$$\mathscr{L} = -\frac{1}{L-1} \sum_{t=1}^{L-1} \log p_\theta(x_{t+1} \mid x_{\leq t}).$$

Batch size and token budget. Training is executed for a total of 162,000 optimizer iterations, processing approximately 1.168×10^{11} tokens in total. To satisfy memory constraints while keeping a large effective batch, we use gradient accumulation across GPUs.

The effective batch configuration is:

- **Global batch size:** $\approx$ 0.72M tokens ($\approx$ 352 sequences).
- **Micro-batch size:** 22 sequences per GPU (per iteration).
- **Number of GPUs:** $N_{\text{GPU}} = 8$.

- **Gradient accumulation steps:** $N_{\text{accum}} = 2$.
- **Sequence length:** $L = 2048$ tokens.

The global batch size in sequences is therefore

$$B_{\text{global}} = B_{\text{micro}} \times N_{\text{GPU}} \times N_{\text{accum}} = 22 \times 8 \times 2 = 352 \text{ sequences},$$

which corresponds to $352 \times 2048 \approx 7.2 \times 10^5$ tokens per optimizer step.

Optimizer and regularization. We train the model with AdamW, using a fused implementation when available to reduce kernel-launch overhead on B200 GPUs. The detailed settings are:

- **Optimizer:** AdamW (fused).
- **Peak learning rate:** 4.0×10^{-4}.
- **Minimum learning rate:** 4.0×10^{-5} (10% of peak).
- **Learning rate schedule:** cosine decay with linear warmup.
- **Warmup steps:** 2,000 iterations.
- **Weight decay:** 0.1 (applied to weight matrices, excluded from norms).
- **Gradient clipping:** global norm clipped at 1.0.
- **Adam β_1:** 0.9.
- **Adam β_2:** 0.95.

The learning rate is linearly ramped up from 0 to the peak value over the first 2,000 steps, and then decayed to the minimum value via a cosine schedule over the remaining steps. The choice of peak/min learning rate and token budget is informed by scaling-law analysis so that the (model size, dataset size) pair lies near the compute-optimal frontier for the available FLOPs.

A.1.4 Software Stack & Acceleration

The software stack is optimized for high throughput on modern accelerators while maintaining a clean and modular codebase.

- **Framework:** PyTorch (latest stable release), installed via `requirements.txt` in `enhanced_training_system/`.
- **Training entry point:** `enhanced_training_system/train.py` with the flagship configuration file `config/full_qwen3_1.8b_b200_optimal.py`.
- **Launcher:** `torchrun –standalone –nproc_per_node=8` on a single B200 node.
- **Compiler:** `torch.compile` (Inductor backend) is enabled to fuse kernels and optimize the training graph end-to-end.
- **Attention kernel:** FlashAttention-2 (v2.x) is used as the default attention backend, with an automatic fallback to PyTorch's scaled dot-product attention (SDPA) when

FlashAttention is not available.

- **Precision:** Mixed `bfloat16` training (weights and activations in BF16, critical accumulations in FP32) to leverage B200 tensor cores while maintaining numerical stability.
- **Parallelism:** DDP for data parallelism with NCCL as the communication backend.
- **Optimizer sharding:** ZeRO-1 to shard optimizer states across devices; parameters and gradients remain fully replicated.
- **Dataloader:** PyTorch `DataLoader` with 16 workers, pinned memory, and prefetching enabled to overlap host-to-device transfers with GPU computation.

This configuration is sufficient to saturate the B200s at a high fraction of their BF16 peak performance while keeping the training loop implementation relatively simple (pure data parallelism).

A.1.5 Dataset and Data Pipeline

Dataset. We pre-train on a subset of the SlimPajama corpus, a large-scale cleaned web dataset containing 627B tokens in its full configuration. For this project, we target approximately 1.17×10^{11} tokens–a sizable but still partial sample of the full corpus–to fit within the 60-hour compute budget while respecting compute-optimal scaling considerations.

The key dataset properties are:

- **Base corpus:** SlimPajama (627B tokens total).
- **Training subset:** $\approx 1.168 \times 10^{11}$ tokens processed.
- **Tokenizer:** Qwen3 byte-level BPE, compatible with an extended `cl100k_base`-style vocabulary with 151,643 tokens.
- **Language coverage:** Primarily English, with multilingual support inherited from SlimPajama and the Qwen vocabulary.

Preprocessing and sequence packing. Data preparation follows the pipeline implemented in `enhanced_training_system/data/slimpajama_627b_qwen3`:

1. **Manifest construction.** Raw SlimPajama shards are indexed into a JSONL manifest (file paths plus metadata) via a script such as:

   ```
   python build_manifest.py –output manifests/slimpajama_manifest.jsonl
   ```

2. **Tokenization.** The manifest is then tokenized with the Qwen3 tokenizer, writing tokenized shards to disk:

   ```
   python tokenize_from_manifest.py \
   –manifest manifests/slimpajama_manifest.jsonl \
   –split train \
   –tokenizer ../../qwen3_tokenizer \
   ```

```
--output-dir tokenized \
--spawn-workers -1
```

3. **Sequence packing.** Tokenized shards are concatenated and packed into fixed-length sequences of 2048 tokens. Packing is performed to minimize padding: multiple shorter documents are concatenated until the context window is filled, which yields near-dense sequences and maximizes the fraction of non-pad tokens.
4. **Train/validation split.** The packed token stream is split into `train.bin` and `val.bin` in a memory-mappable format. During training, each batch samples random contiguous windows of length 2048 from the training stream; the target sequence is obtained by shifting the input by one token.

At training time, the data loader wraps these memmapped token arrays in a custom dataset that returns (x, y) pairs, where x is a 2048-token input sequence and y is x shifted by one position. Randomized indexing ensures that each epoch over the subset explores a diverse mix of documents, while on-the-fly sequence packing keeps the effective token utilization close to 100%.

Overall, this configuration–combining a modern Qwen3-style architecture, compute-aware optimization schedule, efficient software stack, and a large SlimPajama subset–enables us to train a 1.8B-parameter model from scratch on a single 8×B200 node within a fixed compute budget.

A.2 Evaluation Details

A.2.1 Evaluation Method

The evaluation framework assesses model performance on three multiple-choice science and reasoning benchmarks: OpenBookQA, ARC-Challenge and ARC-Easy. Each dataset is loaded in its canonical public form.

The evaluation pipeline measures model performance exclusively through deterministic log-probability scoring. Each benchmark question is paired with its candidate answers using a fixed template of the form "Question: X" followed by "Answer: Y". For every possible answer choice, the model computes the average log probability of the answer sequence conditioned on the question prompt. The option achieving the highest score is selected as the predicted label. This procedure is fully reproducible, avoids sampling noise and follows standard practice in language model benchmarking.

This approach evaluates the model's ability to assign calibrated likelihoods to correct answers rather than its generative behavior. It allows direct comparison across checkpoints and datasets because all predictions arise from the same deterministic scoring rule.

A.2.2 Evaluation Dataset Examples

ARC-Easy

One year, the oak trees in a park began producing more acorns than usual. The next year, the population of chipmunks in the park also increased. Which best explains why there were more chipmunks the next year?

A) Shady areas increased.
B) Food sources increased.
C) Oxygen levels increased.
D) Available water increased.

Correct answer: (B)

ARC-Challenge

Which statement best compares single-celled and multi-celled organisms?

A) Tissues in a single-celled organism are like the cells in a multi-celled organism.
B) The nucleus in a single-celled organism is like the skin of a multi-celled organism.
C) Organelles in a single-celled organism are like the organs in a multi-celled organism.
D) The cytoplasm in a single-celled organism is like the nervous system in a multi-celled organism.

Correct answer: (B)

OpenBookQA

The sun is responsible for

A) puppies learning new tricks.
B) children growing up and getting old.
C) flowers wilting in a vase.
D) plants sprouting, blooming and wilting.

Correct answer: (D)