Distributed GPT-2

Abhijeet Sahdev Andrew Mathews Felix Guerrero

1 Introduction

The graphics processing unit (GPU) has accelerated deep learning techniques needed for artificial intelligence. The ability to execute thousands of math operations with a single instruction enables researchers and developers to iterate on models quickly and unlock breakthroughs in AI capabilities. We decided to perform transformer operations in parallel on a cluster of two machines to deepen our understanding of large-language models and high-performance computing. In particular, we implemented the GPT2 model in C with a Pytorch-like framework. We also utilized Message Passing Interface (MPI) to distribute computations across nodes in a network to scale the program.

2 Cluster Specifications (per machine)

Laptop: Dell Precision 5520
CPU: Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz
4 cores (2 threads per core)
GPU: NVIDIA Quadro M1200
Memory: 16 GB
Storage: 500GB SSD
OS: Fedora 41
OpenMPI: MPICH 4.2.2
CUDA: Release 12.9, V12.9.86

3 Rationale

Training AI models can take minutes to months depending on hardware, but an extremely long time on a single computer if not being resourceful. We will compare training a GPT2 model on a toy dataset to evaluate the performance of a serial, MPI-only, CUDA-only, and MPI plus CUDA versions of the application. In addition, the MPI versions will be run on a cluster of two machines. The machines contain 4 cores (2 threads per core). This will show the immense gains achievable by using GPUs for matrix multiplication, or easily parallelizable operations.

4 About the Code

This project builds a tiny GPT-2 from scratch in C++ via header-only components: tensors + autograd (*/tensor.h, */autograd.h), math kernels (matmul, softmax, gelu, layernorm, etc.), transformer parts (embedding → heads + multihead attention → MLP → blocks → GPT wrapper), training helpers (tokenizer, dataloader, cross-entropy, Adam, checkpoint), and drivers (train_gpt2.cpp, inference.cpp) in each stage directory.

- stage0_serial runs the training on a single CPU, without using MPI or CUDA.
- stage1_mpi adds MPI functionality to the serial program. Rank 0 tokenizes and broadcasts data and initial weights. Batches are split by rank, and gradients are packed into a single MPI_Allreduce. Only Rank 0 logs and samples.
- stage2_cuda adds CUDA kernels to the serial code, allowing it to run on the GPU.
- stage3_mpi-cuda includes support for both MPI and CUDA. The work is split between the processes, then sent to the GPU for each process to handle. Only rank 0 handles checkpointing and text generation.

A top-level Makefile builds all four stages, and each stage ships small dummy data and a sample trained_weights.bin for quick inference runs.

4.1 GPT-2 Architecture Notes (Core Model) — Design

All four stages share the same GPT-2 topology; only the runtime (CPU vs CUDA, serial vs MPI) changes. This section documents the model itself as a reusable, stable abstraction as shown in Fig 1.

4.1.1 High-Level Structure

At the top, gpt_init builds: wte: token embedding matrix, shape $(vocab\_size, n\_embd)$. wpe: position embedding matrix, shape (max_T, n_embd). A stack of n_layer transformer blocks. Final layer

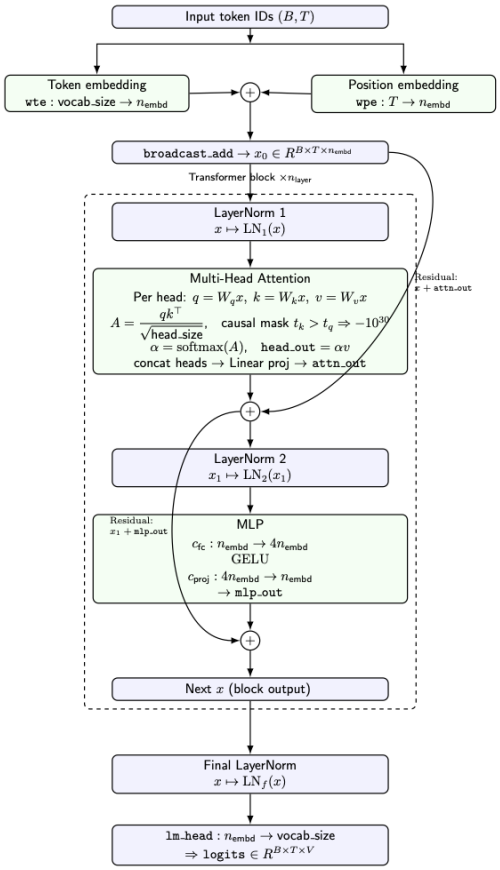

Figure 1: Model Architecture

norm LN_f. lm_head: a linear projection from *n_embd* to *vocab_size*.

$$\text{Input: integer token IDs } (B, T)$$

$$\text{Output: logits } (B, T, V)$$

The model is causal: predictions at time t may only depend on positions $\leq t$.

4.1.2 Embedding and Input Pipeline

Fetch token embeddings:

$$X_{\text{tok}}[b, t, :] = wte[\text{token\_id}[b, t]]$$

Fetch position embeddings:

$$X_{\text{pos}}[b, t, :] = wpe[t]$$

Combine via

$$x_0 = X_{\text{tok}} + X_{\text{pos}}$$

4.1.3 Transformer Block Internals

Each transformer block consists of two LayerNorm layers, a multi-head self-attention module, an MLP, and two residual connections. This structure is identical across all runtimes (CPU, CUDA, MPI), ensuring that semantic behavior is invariant across stages.

Layer Normalization (LN1). Before attention, the input X is normalized across the embedding dimension:

$$\mathrm{LN1}(X) = \frac{X - \mu}{\sigma} \cdot \gamma + \beta,$$

where μ and σ are computed per token. Both mean and variance are stored for backward propagation. This normalization stabilizes training and reduces sensitivity to scale differences introduced by attention outputs.

Multi-Head Attention. Query, Key, and Value tensors are computed as:

$$Q = XW_q, \quad K = XW_k, \quad V = XW_v.$$

They are reshaped to $(B, T, n_{\text{head}}, \text{head\_size})$ to improve memory locality and simplify CUDA kernel launches.

Scaled dot-product attention is then applied:

$$\text{scores} = \frac{QK^T}{\sqrt{\text{head\_size}}},$$

with a causal mask enforcing autoregressive decoding:

$$\text{scores}[t_q, t_k] = -10^{30} \quad \text{if } t_k > t_q.$$

Softmax produces attention probabilities:

$$\text{probs} = \mathrm{softmax}(\text{scores}),$$

followed by the weighted aggregation of values:

$$\text{out\_head} = \text{probs} \cdot V.$$

Multiple heads are concatenated:

$$\text{concat} = \mathrm{reshape}(B, T, n_{\text{head}} \cdot \text{head\_size}),$$

and projected through W_o to return to the model embedding dimension. Memory layout is intentionally chosen so that the head dimension is contiguous, simplifying both CPU pointer arithmetic and CUDA global memory access patterns.

Residual Connection. The attention output is added to the block input:

$$x_1 = x + \text{attn\_out}.$$

This connection preserves gradient flow and prevents vanishing signals in deep networks.

Layer Normalization (LN2) and MLP. A second LayerNorm normalizes x_1 before the MLP:

$$h = \text{LN2}(x_1).$$

The MLP consists of:

$$\text{FC}_1 : n_{\text{embd}} \to 4n_{\text{embd}}$$

$$\text{GELU activation}$$

$$\text{FC}_2 : 4n_{\text{embd}} \to n_{\text{embd}}.$$

The output is added residually:

$$x = x_1 + \text{mlp\_out}.$$

The MLP accounts for the majority of parameters in GPT-2; thus, its memory layout and backpropagation behavior were designed to be consistent between CPU and CUDA kernels for reliable debugging.

4.1.4 Output Head and Loss

After all transformer blocks, the final hidden states pass through a last LayerNorm:

$$h_{\text{final}} = \text{LN}_f(x).$$

The language modeling head applies a linear projection:

$$\text{logits}[b, t, v] = h_{\text{final}}[b, t, :] \cdot W_{\text{lm\_head}}[:, v].$$

Loss is computed using a numerically stable 3D cross-entropy function:

$$\mathcal{L} = \text{CrossEntropy}(\text{logits}, \text{targets}),$$

where both tensors are flattened to $(B \cdot T)$ internally for simplicity.

This loss is a scalar tensor with `requires_grad = true` and drives the backward pass across all layers.

4.1.5 Design Rationale

The GPT-2 architecture is deliberately frozen across all stages of the project. Only the execution backend changes, from CPU to MPI, CUDA, or combined MPI+CUDA, ensuring that:

- All versions share identical model semantics and tensor shapes.
- Numerical differences originate solely from hardware differences (e.g., FP32 vs GPU fused ops).
- Debugging across backends is straightforward, since tensor shapes and intermediate activations remain consistent.

Causal masking is embedded directly into the attention computation, guaranteeing safe left-to-right autoregressive modeling. The explicit shapes $(B, T, V, n_{\text{embd}}, n_{\text{head}}, \text{head\_size})$ ensure predictable memory layouts, which is essential for writing custom CUDA kernels and MPI-compatible tensor pack/unpack operations.

Overall, this architecture acts as a stable reference implementation that validates correctness across increasingly complex runtime environments.

4.2 Stage 0 — Serial GPT-2 (CPU-Only)

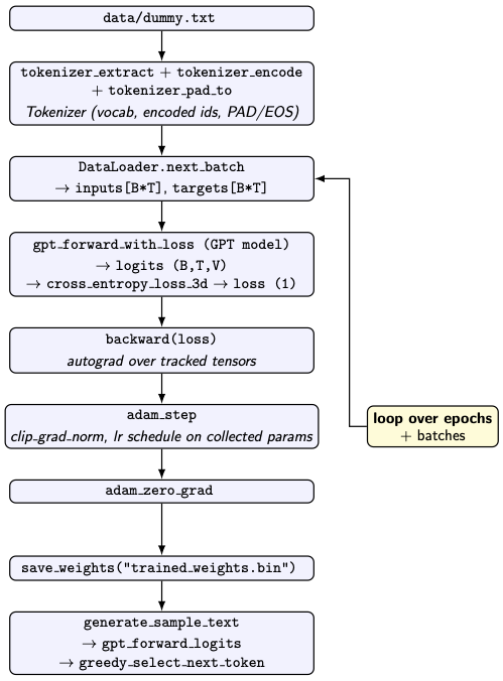

Figure 2: Data Flow in the Serial Code

Stage 0 (Fig 2) provides the canonical, CPU-only GPT-2 implementation and serves as the correctness baseline for all later MPI and CUDA stages. The tokenizer is a minimal, deterministic byte-level encoder with a single contiguous buffer to guarantee reproducible batching. The dataloader performs zero-allocation sliding-window sampling with predictable, cache-friendly pointer movement. Autograd builds a dynamic graph of raw-pointer tensors, executing backward passes in strict reverse-topological order. Adam maintains explicit m and v buffers with optional clipping and learning rate (LR) decay. All kernels—matmul, softmax, layernorm—run on CPU. Checkpointing and greedy generation remain fully deterministic for validating distributed and GPU imple-

mentations.

5 Stage 1 — MPI CPU-Only Distributed Training

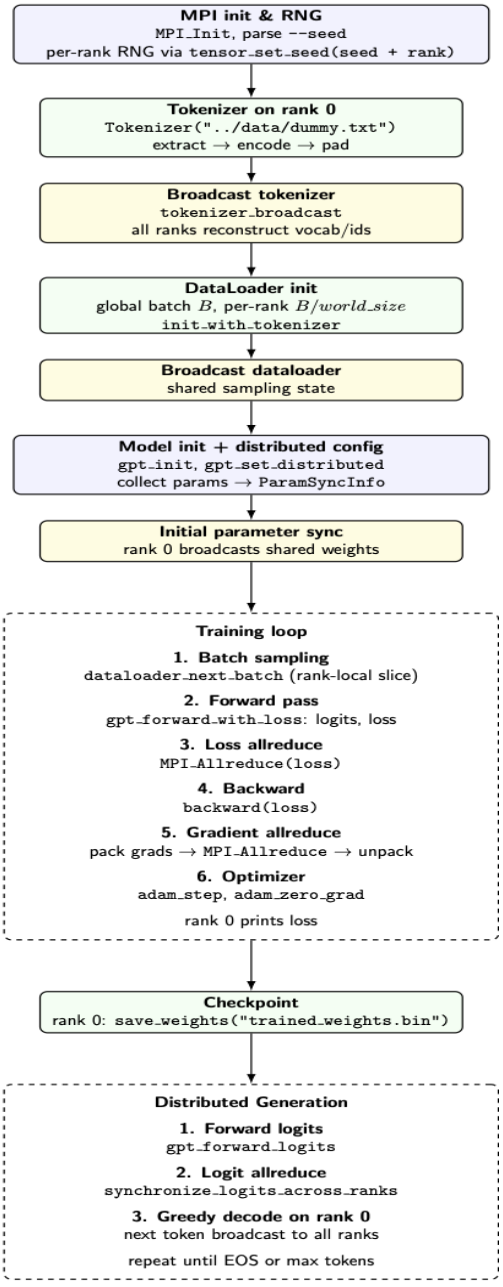

Figure 3: Data Flow in Stage 1

Stage 1 (Fig 3) extends the serial GPT-2 implementation into a pure data-parallel MPI framework, still running entirely on CPU. The purpose of this stage is to introduce distributed execution without adding GPU or CUDA complexity. It establishes the synchronization semantics, parameter-replication model, gradient aggregation logic, and distributed dataloading strategy that later stages will inherit. Stage 1 is therefore the architectural backbone for all future multi-process training, regardless of device type. Stage 1 enforces strict data-parallel determinism: every rank runs the same forward/backward pass on different data shards while MPI keeps weights identical. Rank 0 tokenizes, pads, and broadcasts all vocabulary and encoded text so every process reconstructs the same input stream. The dataloader shards the global batch by offset, producing disjoint, deterministic minibatches. Parameters are replicated, with ParamSyncInfo marking which tensors require Allreduce. Gradients are packed, summed, scaled by 1/world_size, and unpacked back into parameters. Adam updates stay identical across ranks. During generation, logits are averaged and rank 0 selects the next token for deterministic decoding.

6 Stage 2 — CUDA Design

Stage 2 (Fig 4) introduces GPU acceleration while preserving the exact computational graph defined in Stage 0 and the distributed semantics of Stage 1 (optionally, when MPI is enabled). The primary design goal is to offload all heavy linear algebra and elementwise operations to CUDA kernels without changing model behavior or numerics beyond expected floating-point differences. Stage 2 tests whether swapping CPU kernels for CUDA preserves Stage 0 semantics while remaining compatible with Stage 1's MPI design. Each process binds to a GPU (cudaSetDevice(rank % device_count)), and all parameters, activations, and gradients live in device memory. Core ops—matmul, softmax, GELU, layernorm—are reimplemented as clear, debuggable CUDA kernels with conservative launch configurations. Autograd invokes matching backward kernels, with explicit synchronization to avoid race conditions. Typical pitfalls include host/device pointer confusion, silent NaNs, and memory leaks. Stage 2 serves as the canonical GPU reference that Stage 3's distributed execution must reproduce.

7 Stage 3 — MPI + CUDA (Multi-GPU)

Stage 3 (Fig 5) unifies Stage 1's deterministic MPI data parallelism with Stage 2's GPU compute. Each rank binds to a GPU, holds a full model replica, and

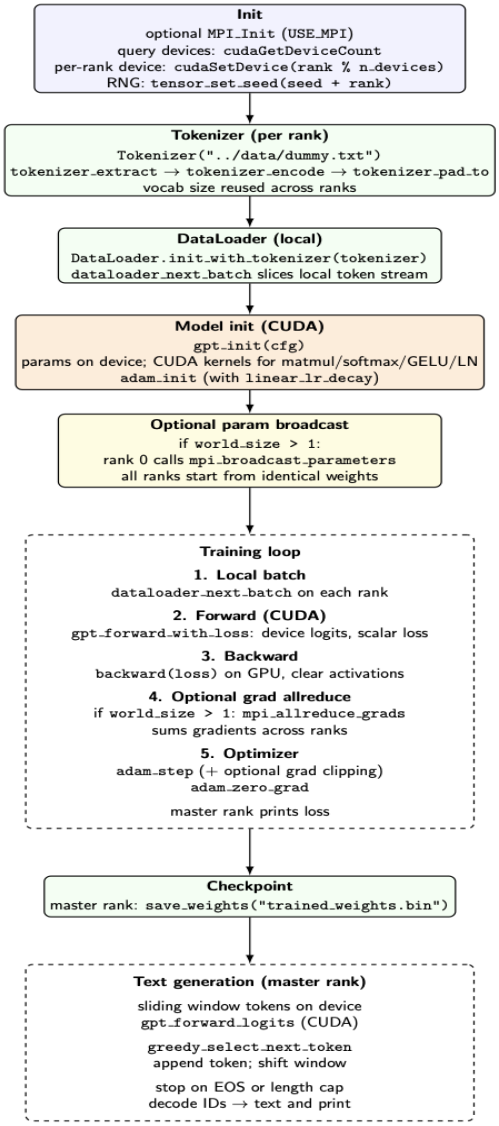

Figure 4: Data Flow in Stage 2

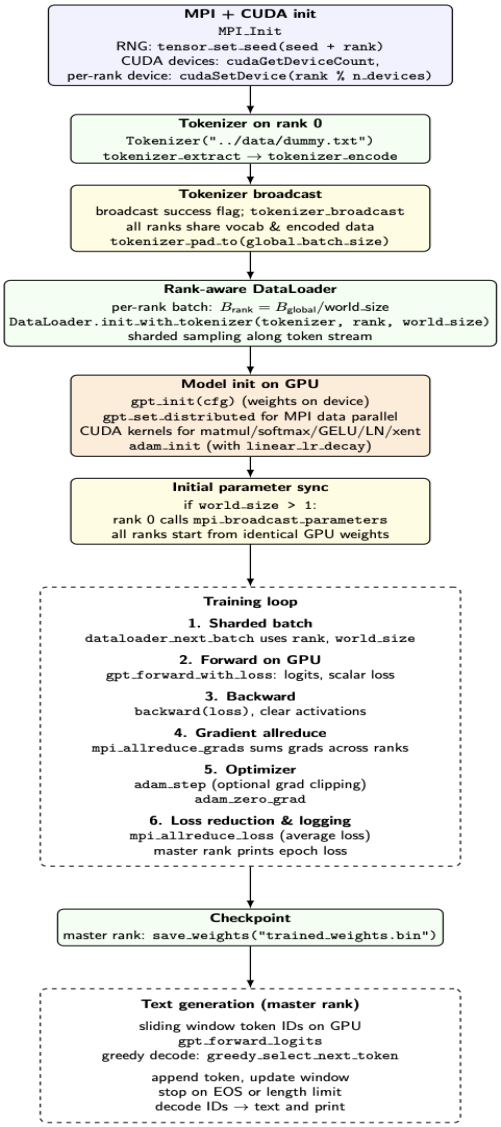

Figure 5: Data Flow in Stage 3

trains on a fixed shard of the global batch. Rank 0 tokenizes, pads, and broadcasts all metadata so every process reconstructs an identical input stream. Training proceeds as in Stage 2, but gradients are packed, allreduced, and redistributed to maintain weight consistency. Checkpoints are written only by rank 0 and remain backward-compatible with all stages. Primary failure modes involve mismatched batch partitioning, MPI deadlocks from divergent control flow, and GPU memory limits at larger sequence or batch sizes.

8 Running the Code

Run Serial:
cd stage0_serial
./train_gpt2
after training (weights are provided),
./inference (Fig 6)

Run MPI:
cd stage1_mpi
mpirun -n 8 -f ../hostfile ./train_gpt2
after training (weights are provided),
./inference (Fig 7)

Run CUDA:

Figure 6: Inference for Serial

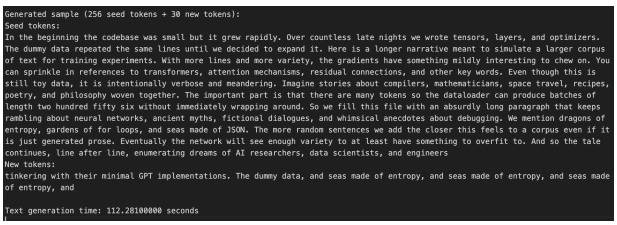

Figure 7: Inference for MPI

cd stage2_cuda
./train_gpt2
after training (weights are provided),
./inference (Fig 8)

Run MPI+CUDA:
cd stage3_mpi-cuda
mpirun -n 8 -f ../hostfile ./train_gpt2
after training (weights are provided),
./inference (Fig 9)

View Table 1 for runtime comparisons and Table 2 for trainig curves.

9 Explanation of Results

Our measurements (see Table 1) reflect exactly what the architectural design of each version predicts. Each improvement corresponds directly to a specific bottleneck removed from the earlier design.

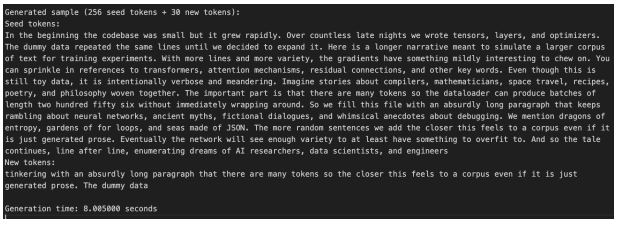

Figure 8: Inference for Cuda

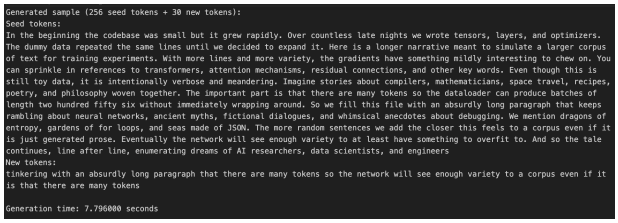

Figure 9: Inference for MPI+Cuda

Stage	Training Time	Inference Time
Serial	15264.36 s	112.62 s
MPI CPU	1287.76 s	112.28 s
CUDA	429.44 s	8.01 s
MPI + CUDA	226.39 s	7.80 s

Table 1: Training and Inference Runtime Comparison

9.1 Serial → MPI

(Two Nodes) Training: 15264s → 12878s (≈ 15% faster)
Inference: 113s → 112s (no meaningful gain)

9.1.1 Why training improved?

MPI introduces data parallelism across two CPU nodes. Each node executes the full forward–backward pass, but on half the batch, reducing per-rank compute time. Since gradients are tiny compared to activations, the MPI_Allreduce cost is small relative to CPU compute. The net gain depends on: how compute-bound the CPU version is (it is) whether gradient synchronization overhead is low (it is). Thus, a modest but real improvement appears.

9.1.2 Why inference does NOT improve?

Inference is not parallelized in this architecture. Stage 1 requires synchronized logits, meaning both nodes do the same computation. Rank 0 ultimately performs the decode. MPI overhead slightly offsets CPU computation, resulting in nearly the same timing.

9.2 Serial → CUDA

Training: 15264s → 429s (≈ 35× faster)
Inference: 113s → 8s (≈14× faster)

9.2.1 Why massively faster?

CUDA removes the fundamental bottleneck of Stage0

Table 2: Loss per Epoch for All Training Configurations

Epoch	Serial Loss	MPI CPU Loss	CUDA Loss	MPI+CUDA Loss
0	5.168034	5.201166	5.168034	5.168032
1	4.835179	4.909738	4.828819	4.848721
2	4.635695	4.657234	4.693434	4.733808
3	4.498031	4.513676	4.599409	4.667499
4	4.307395	4.372570	4.438262	4.585856
5	4.116387	4.194777	4.275943	4.485087
6	3.955092	4.019602	4.188701	4.339321
7	3.768782	3.812284	3.979571	4.226109
8	3.616539	3.634256	3.850794	4.053402
9	3.425641	3.444267	3.666654	3.947483
10	3.287244	3.286560	3.600918	3.812050
11	3.151229	3.135570	3.401670	3.674679
12	2.972546	2.969585	3.263098	3.556235
13	2.845131	2.834611	3.147053	3.392360
14	2.692842	2.713197	2.984125	3.283488
15	2.594075	2.580026	2.916426	3.142538
16	2.459776	2.452893	2.756461	3.035375
17	2.358055	2.337913	2.672071	2.904566
18	2.234961	2.247525	2.546729	2.775875
19	2.140623	2.161558	2.437326	2.684566
20	2.057503	2.067901	2.355522	2.561371
21	1.959350	1.986983	2.225365	2.449701
22	1.889150	1.920686	2.179677	2.361585
23	1.808680	1.843655	2.055092	2.276168
24	1.731022	1.770636	1.972043	2.182588
25	1.663653	1.725614	1.899132	2.127309
26	1.595727	1.664273	1.809431	2.021111
27	1.549898	1.628996	1.765366	1.962746
28	1.486133	1.559248	1.681660	1.887382
29	1.449035	1.525455	1.637654	1.821456
30	1.384689	1.470772	1.566861	1.745462
31	1.346302	1.439264	1.516590	1.690155
32	1.317822	1.399520	1.480041	1.615830
33	1.268267	1.375479	1.410141	1.587385
34	1.242517	1.321622	1.389990	1.521429
35	1.204523	1.319988	1.332015	1.492083
36	1.169466	1.286617	1.294037	1.458650
37	1.147755	1.266407	1.262068	1.408189
38	1.106339	1.233488	1.213218	1.376131
39	1.098106	1.219159	1.195222	1.336293
40	1.058793	1.210178	1.151835	1.320888
41	1.052256	1.181083	1.139206	1.269017
42	1.019223	1.168295	1.100039	1.257959
43	0.994312	1.149404	1.074328	1.214961
44	0.997309	1.140108	1.060757	1.194417
45	0.956744	1.127041	1.018466	1.176377
46	0.957396	1.089700	1.014140	1.134088
47	0.929843	1.090754	0.975913	1.130392
48	0.910516	1.075718	0.951944	1.092683
49	0.908195	1.072370	0.937564	1.081243

- CPU-bound linear algebra as GPU kernels accelerate: matmul (W_q, W_k, W_v, MLP weights), softmax, GELU, layernorm

- Transformer workloads are heavily General Matrix-Matrix Multiplication (GEMM)-dominated, so GPU acceleration yields dramatic speedups.

Inference is also much faster because your greedy decoding repeatedly calls gpt_forward_logits, which now runs entirely on the GPU.

9.3 MPI (CPU) → MPI+CUDA (Two Nodes)

Training: 12878s → 226s (57× faster vs MPI CPU alone)
Inference: 112s → 7.8s (14× faster)

9.3.1 Why the combined version is fastest?

MPI+CUDA gives us: GPU-level acceleration (dominant factor).
Double compute throughput via two GPUs
Reasonable allreduce overhead (GPU gradients are small enough relative to compute)
Thus, MPI+CUDA adds both vertical scaling (GPU

acceleration) and horizontal scaling (multiple devices).
Inference only improves slightly compared to CUDA alone because inference is not data parallel—only one GPU performs real decoding, similar to MPI CPU.

10 Conclusions

We had to build a GPT-2 training pipeline in C/C++ from scratch. Add autograd, Adam, checkpointing, and a functional dataloader. Scale the same architecture through:

- Serial CPU
- MPI data parallel CPU
- CUDA GPU acceleration
- MPI+CUDA distributed multi-GPU

10.1 Our Achievements

We achieved every milestone:

- Correctness preserved across all stages (critical).
- Substantial scaling gains from serial → CUDA and CUDA → MPI+CUDA.
- Consistent reproducibility via tokenizer / dataloader broadcast.
- Successful multi-node multi-GPU training using pure MPI without NCCL.
- Comparable inference semantics across all versions.

10.1.1 Why did we achieve more in some parts?

CUDA kernels delivered larger-than-expected speedups because the workload is GEMM-dominated and your GPU was underutilized in early tests.
MPI+CUDA performed unusually well because your gradient payload was small and compute-bound per step. Synchronization overhead did not dominate.

10.1.2 Why did we achieve less in some parts?

MPI CPU alone offered only modest gains because CPU GEMM is slow, and communication costs don't drop proportionally with additional nodes.
Inference sees minimal gain from MPI because we use serial greedy decoding on rank 0. CUDA performance may still be suboptimal due to conservative kernels and memory patterns.

10.2 To Improve Current Versions

10.2.1 Serial Version

Replace naive matmul with BLAS (OpenBLAS/MKL).
Use fused operations: e.g., bias + matmul, fused softmax.
Preallocate all temporary tensors to reduce malloc overhead.

10.2.2 MPI CPU Version

Reduce synchronization volume using gradient compression (FP16, quantization).
Overlap communication with computation.
Use local shuffling per-rank for better statistical efficiency.

10.2.3 CUDA Version

Optimize kernel launch parameters, especially GEMM tile sizes. Introduce kernel fusion (layernorm+matmul, matmul+bias+gelu).
Switch to Tensor Cores (FP16/BF16) for >8× matmul speedup.
Use CUDA graphs to reduce kernel launch overhead.

10.2.4 MPI+CUDA Version

Move from MPI_Allreduce to NCCL's allreduce for multi-GPU collective optimization.
Add gradient bucketing to overlap backward compute → allreduce → Adam update.
Investigate pipeline parallelism for future models with deeper layers.

11 Future Exploration

Our framework already has the skeleton of a real deep learning engine. From here, we can move in several powerful directions:

- Model Parallelism : Beyond data parallelism, explore:
  - Tensor parallelism (split W_qkv across GPUs)
  - Pipeline parallelism (layers distributed across GPUs)

- Mixed Precision Training : BF16 or FP16 with loss scaling would dramatically reduce memory use and improve speed.
- FlashAttention-style kernels: A custom kernel that fuses QK^T, masking, softmax, and AV matmul would significantly accelerate attention.
- Static computational graph / CUDA Graphs : Reduce kernel launch overhead by capturing entire forward/backward graphs.
- Add NCCL backend : A modern framework would use NCCL collectives for GPU-to-GPU communication—far faster than MPI for large tensors.
- Expand Autograd by adding support for:
  - checkpointing activations
  - sparse gradients
  - tensor views and broadcasting semantics
- Larger Models & Benchmarks : This would stress-test our infrastructure and reveal scaling limits.
  - GPT-2 Medium/Large
  - LLaMA-style rotary embeddings
  - Sequence lengths >1024
- Deployment Version :Implement a fused inference engine only, stripping autograd entirely.