The Little Book of llm.c

Version 0.1.1

Duc-Tam Nguyen

2025-09-24

Table of contents

Content **5**

The Book **9**

Chapter 1. Orientation . . . 9
1. What *llm.c* Is . . . 9
2. Repository Tour . . . 11
3. Makefile Targets & Flags . . . 14
4. Quickstart: CPU Reference Path (`train_gpt2.c`) . . . 16
5. Quickstart: 1-GPU Legacy Path (`train_gpt2_fp32.cu`) . . . 19
6. Quickstart: Modern CUDA Path (`train_gpt2.cu`) . . . 21
7. Starter Artifacts & Data Prep (`dev/download_starter_pack.sh`, `dev/data/`) 25
8. Debugging Tips & IDE Stepping (`-g`) . . . 27
9. Project Constraints & Readability Contract . . . 30
10. Community, Discussions, and Learning Path . . . 32

Chapter 2. Data, Tokenization, and Loaders . . . 34
11. GPT-2 Tokenizer Artifacts (`gpt2_tokenizer.bin`) . . . 34
12. Binary Dataset Format (`train.bin` and `val.bin`) . . . 37
13. Dataset Scripts in `dev/data/` . . . 39
14. DataLoader Design (Batching, Strides, Epochs) . . . 41
15. EvalLoader and Validation Workflow . . . 43
16. Sequence Length and Memory Budgeting . . . 46
17. Reproducibility and Seeding Across Runs . . . 48
18. Error Surfaces from Bad Data (Bounds, Asserts) . . . 50
19. Tokenization Edge Cases (UNKs, EOS, BOS) . . . 53
20. Data Hygiene and Logging . . . 55

Chapter 3. Model Definition and Weights . . . 56
21. GPT-2 Config: Vocab, Layers, Heads, Channels . . . 56
22. Parameter Tensors and Memory Layout . . . 59
23. Embedding Tables: Token + Positional . . . 62
24. Attention Stack: QKV Projections and Geometry . . . 64
25. MLP Block: Linear Layers + Activation . . . 67
26. LayerNorm: Theory and Implementation (`doc/layernorm`) . . . 69
27. Residual Connections: Keeping the Signal Flowing . . . 72
28. Attention Masking: Enforcing Causality . . . 74
29. Output Head: From Hidden States to Vocabulary . . . 76

30. Loss Function: Cross-Entropy over Vocabulary . . . 79
Chapter 4. CPU Inference (Forward Only) . . . 81
31. Forward Pass Walkthrough . . . 81
32. Token and Positional Embedding Lookup . . . 85
33. Attention: Matmuls, Masking, and Softmax on CPU . . . 88
34. MLP: GEMMs and Activation Functions . . . 92
35. LayerNorm on CPU (Step-by-Step) . . . 95
36. Residual Adds and Signal Flow . . . 99
37. Cross-Entropy Loss on CPU . . . 102
38. Putting It All Together: The `gpt2_forward` Function . . . 106
39. OpenMP Pragmas for Parallel Loops . . . 109
40. CPU Memory Footprint and Performance . . . 113
Chapter 5. Training Loop (CPU Path) . . . 115
41. Backward Pass Walkthrough . . . 115
42. Skeleton of Training Loop . . . 119
43. AdamW Implementation in C . . . 122
44. Gradient Accumulation and Micro-Batching . . . 125
45. Logging and Progress Reporting . . . 128
46. Validation Runs in the Training Loop . . . 130
47. Checkpointing Parameters and Optimizer State . . . 133
48. Reproducibility and Small Divergences . . . 136
49. Command-Line Flags and Defaults . . . 138
50. Example CPU Training Logs and Outputs . . . 141
Chapter 6. Testing and Profiling . . . 143
51. Debug State Structs and Their Role . . . 143
52. `test_gpt2.c`: CPU vs PyTorch . . . 145
53. Matching Outputs Within Tolerances . . . 147
54. Profiling with `profile_gpt2.c` . . . 149
55. Measuring FLOPs and CPU Performance . . . 151
56. Capturing Memory Usage on CPU . . . 153
57. Reproducing Known Loss Curves (CPU-only) . . . 155
58. Debugging Numerical Stability (NaNs, Infs) . . . 157
59. From Unit Test to Full Training Readiness . . . 159
60. Limitations of CPU Testing . . . 161
Chapter 7. CUDA Training (`train_gpt2.cu`) . . . 163
61. CUDA Architecture Overview (streams, kernels) . . . 163
62. Matrix Multiplication via cuBLAS/cuBLASLt . . . 166
63. Attention Kernels: cuDNN FlashAttention . . . 169
64. Mixed Precision: FP16/BF16 with Master FP32 Weights . . . 172
65. Loss Scaling in Mixed Precision Training . . . 174
66. Activation Checkpointing and Memory Tradeoffs . . . 177
67. GPU Memory Planning: Parameters, Gradients, States . . . 179
68. Kernel Launch Configurations and Occupancy . . . 181

69. CUDA Error Handling and Debugging . . . 184
70. `dev/cuda/`: From Simple Kernels to High Performance . . . 187
Chapter 8. Multi-GPU and Multi-node training . . . 189
71. Data Parallelism in *llm.c* . . . 189
72. MPI Process Model and GPU Affinity . . . 192
73. NCCL All-Reduce for Gradient Sync . . . 195
74. Building and Running Multi-GPU Trainers . . . 197
75. Multi-Node Bootstrapping with MPI . . . 200
76. SLURM and PMIx Caveats . . . 202
77. Debugging Multi-GPU Hangs and Stalls . . . 204
78. Scaling Stories: GPT-2 124M → 774M → 1.6B . . . 206
79. NCCL Tuning and Overlap Opportunities . . . 208
80. Common Multi-GPU Errors and Fixes . . . 211
Chapter 9. Extending the codebase . . . 213
81. The `dev/cuda` Library for Custom Kernels . . . 213
82. Adding New Dataset Pipelines (`dev/data/*`) . . . 215
83. Adding a New Optimizer to the Codebase . . . 217
84. Adding a New Scheduler (cosine, step, etc.) . . . 219
85. Alternative Attention Mechanisms . . . 221
86. Profiling and Testing New Kernels . . . 223
87. Using PyTorch Reference as Oracle . . . 225
88. Exploring Beyond GPT-2: LLaMA Example . . . 227
89. Porting Playbook: C → Go/Rust/Metal . . . 229
90. Keeping the Repo Minimal and Clean . . . 232
Chapter 10. Reproduction, community and roadmap . . . 234
91. Reproducing GPT-2 124M on Single Node . . . 234
92. Reproducing GPT-2 355M (Constraints and Tricks) . . . 236
93. Reproducing GPT-2 774M (Scaling Up) . . . 238
94. Reproducing GPT-2 1.6B on 8×H100 (24h Run) . . . 241
95. CPU-only Fine-Tune Demo (Tiny Shakespeare) . . . 243
96. Cost and Time Estimation for Runs . . . 245
97. Hyperparameter Sweeps (`sweep.sh`) . . . 247
98. Validating Evaluation and Loss Curves . . . 249
99. Future Work: Kernel Library, Less cuDNN Dependence . . . 251
100. Community, GitHub Discussions, and Suggested Learning Path . . . 253

Content

Chapter 1 — Orientation

1. What *llm.c* Is (scope, goals, philosophy)
2. Repository Tour (folders, files, structure)
3. Makefile Targets & Flags (CPU, CUDA, options)
4. Quickstart: CPU Reference Path (`train_gpt2.c`)
5. Quickstart: 1-GPU Legacy Path (`train_gpt2_fp32.cu`)
6. Quickstart: Modern CUDA Path (`train_gpt2.cu`)
7. Starter Artifacts & Data Prep (`dev/download_starter_pack.sh`, `dev/data/`)
8. Debugging Tips & IDE Stepping (`-g`, gdb, lldb, IDEs)
9. Project Constraints & Readability Contract
10. Community, Discussions, and Learning Path

Chapter 2 — Data, Tokenization, and Loaders

11. GPT-2 Tokenizer Artifacts (`gpt2_tokenizer.bin`)
12. Binary Dataset Format (`.bin` with header + tokens)
13. Dataset Scripts in `dev/data/` (Tiny Shakespeare, OpenWebText)
14. DataLoader Design (batching, strides, epochs)
15. EvalLoader and Validation Workflow
16. Sequence Length and Memory Budgeting
17. Reproducibility and Seeding Across Runs
18. Error Surfaces from Bad Data (bounds, asserts)
19. Tokenization Edge Cases (UNKs, EOS, BOS)
20. Data Hygiene and Logging

Chapter 3 — Model Definition & Weights

21. GPT-2 Config: vocab, layers, heads, channels
22. Parameter Tensors and Memory Layout
23. Embedding Tables: token + positional
24. Attention Stack: QKV projections and geometry
25. MLP Block: linear layers + activation

26. LayerNorm: theory and implementation (`doc/layernorm`)
27. Residual Streams: skip connections explained
28. Loss Head: tied embeddings and logits
29. Checkpoint Loading from PyTorch
30. Parameter Counting and Sanity Checks

Chapter 4 — CPU Inference (Forward only)

31. Forward Pass Walkthrough
32. Token and Positional Embedding Lookup
33. Attention: matmuls, masking, softmax on CPU
34. MLP: GEMMs and activation functions
35. LayerNorm on CPU (step-by-step)
36. Residual Adds and Signal Flow
37. Cross-Entropy Loss on CPU
38. Putting It All Together: The `gpt2_forward`
39. OpenMP Pragmas for Parallel Loops
40. CPU Memory Footprint and Performance

Chapter 5 — Training Loop (CPU Path)

41. Skeleton of Training Loop
42. AdamW Implementation in C
43. Learning Rate Schedulers (cosine, warmup)
44. Gradient Accumulation and Micro-Batching
45. Logging and Progress Reporting
46. Validation Runs in Training Loop
47. Checkpointing Parameters and Optimizer State
48. Reproducibility and Small Divergences
49. Command-Line Flags and Defaults
50. Example CPU Training Logs and Outputs

Chapter 6 — Testing, Profiling, & Parity

51. Debug State Structs and Their Role
52. `test_gpt2.c`: CPU vs PyTorch
53. `test_gpt2cu.cu`: CUDA vs PyTorch
54. Matching Outputs Within Tolerances
55. Profiling with `profile_gpt2.cu`
56. Measuring FLOPs and GPU Utilization
57. Reproducing Known Loss Curves

58. Common CUDA Pitfalls (toolchain, PTX)
59. cuDNN FlashAttention Testing (`USE_CUDNN`)
60. From Unit Test to Full Training Readiness

Chapter 7 — CUDA Training Internals (`train_gpt2.cu`)

61. CUDA Architecture Overview (streams, kernels)
62. Matrix Multiplication via cuBLAS/cuBLASLt
63. Attention Kernels: cuDNN FlashAttention
64. Mixed Precision: FP16/BF16 with Master FP32 Weights
65. Loss Scaling in Mixed Precision Training
66. Activation Checkpointing and Memory Tradeoffs
67. GPU Memory Planning: params, grads, states
68. Kernel Launch Configurations and Occupancy
69. CUDA Error Handling and Debugging
70. `dev/cuda/`: From Simple Kernels to High Performance

Chapter 8 — Multi-GPU & Multi-Node Training

71. Data Parallelism in *llm.c*
72. MPI Process Model and GPU Affinity
73. NCCL All-Reduce for Gradient Sync
74. Building and Running Multi-GPU Trainers
75. Multi-Node Bootstrapping with MPI
76. SLURM and PMIx Caveats
77. Debugging Multi-GPU Hangs and Stalls
78. Scaling Stories: GPT-2 124M $\rightarrow$ 774M $\rightarrow$ 1.6B
79. NCCL Tuning and Overlap Opportunities
80. Common Multi-GPU Errors and Fixes

Chapter 9 — Extending the Codebase

81. The `dev/cuda` Library for Custom Kernels
82. Adding New Dataset Pipelines (`dev/data/*`)
83. Adding a New Optimizer to the Codebase
84. Adding a New Scheduler (cosine, step, etc.)
85. Alternative Attention Mechanisms
86. Profiling and Testing New Kernels
87. Using PyTorch Reference as Oracle
88. Exploring Beyond GPT-2: LLaMA Example
89. Porting Playbook: C $\rightarrow$ Go/Rust/Metal

90. Keeping the Repo Minimal and Clean

Chapter 10 — Reproductions, Community, and Roadmap

91. Reproducing GPT-2 124M on Single Node
92. Reproducing GPT-2 355M (constraints and tricks)
93. Reproducing GPT-2 774M (scaling up)
94. Reproducing GPT-2 1.6B on 8×H100 (24h run)
95. CPU-only Fine-Tune Demo (Tiny Shakespeare)
96. Cost and Time Estimation for Runs
97. Hyperparameter Sweeps (`sweep.sh`)
98. Validating Evaluation and Loss Curves
99. Future Work: Kernel Library, Less cuDNN Dependence
100. Community, GitHub Discussions, and Suggested Learning Path

The Book

```
Small file, giant dream,
llm.c whispers tokens,
worlds unfold in text.
```

Chapter 1. Orientation

1. What *llm.c* Is

Imagine you wanted to peek inside a modern AI model-not by reading thousands of lines of optimized C++ or CUDA hidden inside a giant framework, but by opening a small, neat folder and seeing the entire training pipeline laid out in front of you. That is what *llm.c* gives you.

At its heart, *llm.c* is a reference implementation of how to train and run a GPT-2 style language model, written in pure C (and CUDA). The key word is *reference*: the code is meant to be minimal, readable, and educational. You don't need to wade through abstraction layers or device-specific macros. Instead, you get a version that looks almost like pseudocode, but still compiles and runs on your computer.

Why This Project Exists

Deep learning frameworks like PyTorch and TensorFlow are amazing for getting models to work quickly, but they hide most of the actual mechanics. Under the hood, there's a lot happening: tensors are allocated in memory, gradients are computed through backpropagation, optimizer states are updated, and schedulers adjust the learning rate. Most of us never see those details, because the framework handles them for us.

llm.c flips this around. It says: *what if we removed the black box and showed you exactly how a GPT-2 model is trained, line by line?* It's not about speed or production deployment. It's about clarity, education, and demystifying how large language models work.

Key Characteristics

- Minimalism: The CPU version (`train_gpt2.c`) avoids complicated optimizations so that beginners can follow the logic. Even the CUDA version tries to stay simple, with only necessary calls to cuBLAS/cuDNN.
- Self-contained: No external frameworks. The code defines its own tokenizer, dataloader, optimizer, and scheduler. Everything you need is in the repository.
- Parallels to PyTorch: Each function in the C/CUDA implementation has a counterpart in PyTorch. The repo even ships with Python test files to prove that the outputs match within tolerance.
- Step-by-step scalability: You can start with a tiny model on CPU and, once you understand the basics, switch to GPU, multi-GPU, or even multi-node training. The structure remains the same, just faster.

What You Can Do With It

1. Train GPT-2 from scratch: Start with a small dataset (like Tiny Shakespeare) and see the model learn patterns in language.
2. Experiment with configurations: Change number of layers, sequence length, or hidden size, then watch how memory and training time scale.
3. Learn GPU training internals: Move from CPU to CUDA, and later to multi-GPU with MPI/NCCL, to see how real distributed training works under the hood.
4. Profile performance: The repo includes profiling tools so you can measure FLOPs, memory bandwidth, and kernel execution times.
5. Reproduce big models: With enough hardware, you can actually retrain GPT-2 124M or larger versions, using the exact same setup described in the README.

Why You Should Care

If you've ever used a large language model and thought *"this feels like magic"*, *llm.c* is your chance to peel back the curtain. You'll see:

- How text is converted into numbers (tokenization).
- How those numbers are fed into matrices and transformed.
- How attention actually looks in code.
- How gradients are computed and weights updated.
- How scaling from one GPU to many is managed.

It's not magic anymore-it's math plus code you can read.

The Big Picture

Think of *llm.c* as a teaching lab for modern AI:

- On day one, you can run a small demo on CPU and see loss values dropping as the model learns.
- On day ten, you might be profiling CUDA kernels or debugging NCCL synchronization across GPUs.
- At every step, the code remains approachable because it avoids unnecessary complexity.

This philosophy makes *llm.c* unique: it's not trying to be the fastest or the most feature-rich. It's trying to be the clearest window into how large language models actually work when stripped down to their essentials.

The Takeaway: *llm.c* is a minimal, transparent re-implementation of GPT-2 training and inference, designed to teach you how language models operate under the hood. It's not about production performance-it's about giving you a guided tour through the machinery of modern AI, one C file at a time.

Would you like me to now expand Section 2 ("Repository Tour") in the same friendly beginner tone, showing what's inside each folder and file?

2. Repository Tour

When you first open the *llm.c* repository, it may look like a random collection of `.c`, `.cu`, `.py`, and shell scripts. But there's a clear structure hiding underneath. Each part has a purpose, and once you know where things live, navigating the repo feels natural. Let's take a guided walk through the main pieces.

Root Directory: The Entry Points

At the top level, you'll find the core training programs. These are the files you actually compile and run:

- `train_gpt2.c` - The CPU reference implementation. This is the simplest, most readable version of GPT-2 training. It avoids special optimizations so you can follow the math and logic step by step.
- `train_gpt2.cu` - The CUDA implementation. Faster, uses GPU kernels, cuBLAS, and optional cuDNN FlashAttention. This is the version you'd use for serious training runs.
- `train_gpt2_fp32.cu` - A legacy CUDA path, using plain FP32 precision instead of mixed precision. It's slower but useful as a debugging baseline.
- `train_gpt2.py` - The PyTorch reference. This is the oracle: a tiny script in Python/PyTorch that trains the same GPT-2 so you can compare outputs and verify correctness.

Other important root-level files:

- `Makefile` - Defines how to build different versions. Targets like `make train_gpt2` or `make train_gpt2cu` are your entry points.
- `README.md` - The main guide for running experiments, installing dependencies, and reproducing models.

`llmc/` Directory: Utilities and Building Blocks

This folder holds reusable C utilities that the main training files include:

- `utils.h` - Safety wrappers (`fopenCheck`, `mallocCheck`) and helper functions.
- `tokenizer.h` - Implements GPT-2's tokenizer in C: encoding text into token IDs and decoding back to text.
- `dataloader.h` - Defines how training batches are loaded and served, handling dataset splits and iteration.
- `rand.h` - Random number utilities, mirroring PyTorch's `manual_seed` and normal distributions.
- `schedulers.h` - Learning rate scheduling, like cosine decay with warmup.
- `sampler.h` - Implements softmax sampling for text generation and helper RNG.
- `logger.h` - Minimal logging functionality for tracking progress.

Think of `llmc/` as the library that keeps the main files clean and readable. Instead of cluttering `train_gpt2.c` with helpers, everything is modularized here.

`dev/` Directory: Scripts and Extras

This folder is full of supporting tools that make experiments easier:

- `dev/download_starter_pack.sh` - Fetches the GPT-2 124M weights, tokenizer, and datasets. This is the quickest way to get started.
- `dev/data/` - Contains scripts for preparing datasets like Tiny Shakespeare or OpenWebText in the binary format that *llm.c* expects.
- `dev/cuda/` - A place for experimenting with standalone CUDA kernels. This is where you'd go if you want to tinker with custom GPU code beyond the main trainer.

`doc/` Directory: Learning Resources

Documentation that digs deeper into specific topics. For example:

- `doc/layernorm/layernorm.md` - A tutorial-style explanation of Layer Normalization, complete with math and code. It helps you understand one of GPT-2's core components before diving into the C implementation.

This folder is a learning aid. Whenever a concept feels too dense, check here for a more gentle walkthrough.

Test Files

Testing is taken seriously in *llm.c*, because the goal is to prove that the C/CUDA implementation is correct compared to PyTorch:

- `test_gpt2.c` - Runs forward passes and training steps on CPU and compares outputs to PyTorch.
- `test_gpt2cu.cu` - Same idea but for CUDA, including both FP32 and mixed-precision runs.

These files keep everything honest: you can always verify that your build produces the same results as the canonical PyTorch model.

Profiling Tools

For performance deep dives:

- `profile_gpt2.cu` - A CUDA profiling harness that benchmarks kernels and measures throughput.
- `profile_gpt2cu.py` - Python-side profiler for analyzing GPU utilization, memory bandwidth, and FLOPs.

If you're curious about where time is being spent in training, these files show you how to measure it.

Datasets and Artifacts

When you run `download_starter_pack.sh`, you'll get:

- `gpt2_tokenizer.bin` - GPT-2's byte-pair encoding tokenizer, serialized in binary.
- Dataset `.bin` files - Training and validation sets, tokenized and ready for the dataloader.

These files are not in the repo by default but are downloaded or generated locally.

Putting It Together

The repository is structured like a teaching lab:

- Root files are the main experiments.
- `llmc/` is the library of building blocks.
- `dev/` provides extra tools and scripts.
- `doc/` explains tricky concepts in tutorial form.
- Tests and profilers make sure everything matches PyTorch and runs efficiently.

Once you see the pattern, the repo feels less intimidating. Every file has a role in telling the story of how a GPT-2 model is built from scratch in C and CUDA.

3. Makefile Targets & Flags

Every C or CUDA program needs a build system, and in *llm.c* that role is handled by a simple but powerful Makefile. If you've never used `make` before, think of it as a recipe book: you type `make <target>` in your terminal, and it follows the instructions for compiling the code into an executable. In *llm.c*, this file is your control center for choosing which trainer to build, whether to enable GPUs, and which optional features to turn on.

Why a Makefile?

Instead of memorizing long `gcc` or `nvcc` compile commands with dozens of flags, the Makefile captures those instructions once and gives them a short name. For example, building the CPU trainer is as easy as:

```
make train_gpt2
```

Behind the scenes, this calls `gcc`, sets optimization flags, includes the right headers, and links everything together. The same applies to CUDA builds with `nvcc`.

Core Targets

Here are the most important build targets you'll find:

- `train_gpt2` - Builds the CPU-only reference trainer. Uses `gcc` (or `clang`) and links against OpenMP for parallel loops.
- `train_gpt2cu` - Builds the CUDA trainer with mixed precision and optional cuDNN FlashAttention. Uses `nvcc`.

- `train_gpt2_fp32` - Builds the legacy CUDA trainer that stays in pure FP32 (slower but simpler).
- `test_gpt2` - Compiles the CPU test program to compare results against PyTorch.
- `test_gpt2cu` - Compiles the CUDA test program to check GPU parity with PyTorch.
- `profile_gpt2.cu` - Compiles the CUDA profiler harness, used to benchmark kernels and FLOPs.

Each of these produces a binary you can run directly, for example:

```
./train_gpt2
./train_gpt2cu
./test_gpt2
```

Key Flags You Can Toggle

The Makefile also exposes several switches that let you customize the build. You set them when running `make`, like this:

```
make train_gpt2cu USE_CUDNN=1
```

Here are the most important flags:

- `USE_CUDNN` - Enables cuDNN FlashAttention if your system has cuDNN installed. This can give big speedups for attention, but it's optional. By default, it's off.
- `OMP=1` - Tells the CPU trainer to compile with OpenMP enabled. This allows multi-threaded execution, making CPU runs much faster. Usually on by default if OpenMP is detected.
- `DEBUG=1` - Compiles with debugging symbols (`-g`) instead of maximum optimization. Useful when stepping through code in an IDE or using a debugger.
- `PROFILE=1` - Adds profiling hooks, helping you analyze execution time and performance.

Optimization Choices

The default build uses `-O3` optimization, which makes the code run fast but sometimes harder to debug. If you're just learning and want clarity, you can switch to:

```
make train_gpt2 DEBUG=1
```

This creates a binary that runs slower but lets you step through line by line in a debugger. For performance benchmarking, stick with the optimized default.

Multi-GPU and MPI Support

When building the CUDA trainer, the Makefile can also link against MPI and NCCL if they're installed. That's what enables multi-GPU and multi-node training. You usually don't need to change anything-the Makefile automatically detects these libraries and includes them if available.

Putting It All Together

Think of the Makefile as a switchboard for the whole project:

- Want to run the simple CPU demo? → `make train_gpt2`
- Want to train faster on GPU? → `make train_gpt2cu`
- Want to debug kernels? → `make train_gpt2cu DEBUG=1`
- Want to test parity with PyTorch? → `make test_gpt2` or `make test_gpt2cu`

With just a few keystrokes, you control whether you're running a beginner-friendly CPU demo, a high-performance GPU build, or a debugging session.

The takeaway: The Makefile is your control center. It abstracts away complicated compiler commands and gives you a clean menu of options: CPU vs GPU, FP32 vs mixed precision, debug vs optimized, and single vs multi-GPU. Mastering it is the first step to feeling comfortable experimenting inside *llm.c*.

4. Quickstart: CPU Reference Path (`train_gpt2.c`)

The simplest way to begin exploring *llm.c* is with the CPU-only reference implementation. This file, `train_gpt2.c`, is deliberately designed to be minimal, readable, and approachable. It doesn't hide complexity behind libraries or macros. Instead, it shows you exactly how a GPT-2 model is trained, step by step, using plain C and a sprinkle of OpenMP for speed.

Why Start with CPU?

- Clarity first: GPUs add layers of complexity (CUDA kernels, memory transfers, cuBLAS). On CPU, you can focus on the core algorithm without distraction.
- Portability: Any machine with a C compiler can run it-no special hardware required.
- Debuggability: Errors are easier to trace, and you can single-step through the code in an IDE.

The CPU version is slower, but that's a feature here-it forces you to really see what's happening under the hood.

Building the CPU Trainer

From the root of the repository, you just type:

```
make train_gpt2
```

This compiles `train_gpt2.c` into an executable named `train_gpt2`. If your system has OpenMP, the Makefile will detect it and add the right flags.

Running Your First Training Run

Before running, download the starter pack (tokenizer, dataset, configs):

```
./dev/download_starter_pack.sh
```

Now launch training:

```
./train_gpt2
```

You'll see output like:

```
[GPT-2]
max_seq_len: 1024
vocab_size: 50257
padded_vocab_size: 50304
num_layers: 12
num_heads: 12
channels: 768
num_parameters: 124475904
train dataset num_batches: 1192
val dataset num_batches: 128
num_activations: 73347840
val loss 5.325529
step 0: train loss 4.677779 (took 1987.546000 ms)
step 1: train loss 5.191576 (took 1927.230000 ms)
...
```

Each line tells you:

- Model size and config (sequence length, vocabulary size, layers, heads, channels).
- Dataset stats (how many batches for training and validation).
- Activation memory size (a measure of how big the intermediate states are).
- Training progress (step number, train loss, validation loss, time per step).

Inside the Training Loop

Although you don't need to dive into the code yet, here's the high-level flow in `train_gpt2.c`:

1. Load tokenizer and dataset → turns text into tokens.
2. Initialize model parameters → embeddings, attention weights, MLPs, norms.
3. For each batch:
    - Forward pass → compute logits and loss.
    - Backward pass → compute gradients.
    - Update parameters → optimizer step.
4. Log progress → print losses, occasionally run validation.

This mirrors exactly what happens in PyTorch, just spelled out in C.

Performance Notes

On CPU, don't expect speed. Training GPT-2 124M can take days or weeks. But that's not the point. The CPU reference path is like a glass box: everything is visible, no shortcuts. You'll use this to learn the mechanics and to verify that your GPU runs match the same results.

If you want to speed things up slightly, you can:

- Increase OpenMP threads:
  ```
  OMP_NUM_THREADS=8 ./train_gpt2
  ```
- Use a smaller dataset (Tiny Shakespeare) to see faster progress.
- Reduce model size by changing config values (fewer layers, smaller channels).

When to Move On

Once you're comfortable with how training looks on CPU-loss values going down, checkpoints being written, logs appearing-you'll be ready to graduate to the GPU version (`train_gpt2.cu`). That's where performance and scaling come in, but the CPU run gives you the conceptual foundation.

The takeaway: Running `train_gpt2.c` is your first hands-on encounter with GPT-2 training in *llm.c*. It's slow, transparent, and designed for learning. You'll see every piece of the model at work, one step at a time, before diving into the complexity of CUDA.

5. Quickstart: 1-GPU Legacy Path (`train_gpt2_fp32.cu`)

Once you've seen the CPU trainer in action, the natural next step is to try training on a GPU. The file `train_gpt2_fp32.cu` is the simplest GPU entry point. It predates the more advanced mixed-precision trainer (`train_gpt2.cu`), and it runs everything in full 32-bit floating point (FP32) precision. That makes it easier to follow and debug, even though it's slower than modern approaches. Think of it as the "training wheels" for GPU training in *llm.c*.

Why This Path Exists

Modern GPU training almost always uses mixed precision (FP16/BF16 for speed and memory savings, FP32 for stability). But mixed precision introduces extra complexity: scaling losses, maintaining master weights, checking for overflows. For beginners, all that can be distracting.

The FP32 path avoids those complications:

- Every tensor (activations, weights, gradients) is stored as 32-bit floats.
- No special handling of loss scaling is needed.
- Debugging mismatches with PyTorch is straightforward.

The trade-off is performance-this version runs significantly slower and uses more memory.

Building the FP32 CUDA Trainer

From the root of the repository:

```
make train_gpt2_fp32
```

This invokes `nvcc` (the NVIDIA CUDA compiler) and links against cuBLAS for matrix multiplications. The output is an executable named `train_gpt2_fp32`.

Running It

Just like the CPU version, make sure you've downloaded the starter pack first:

```
./dev/download_starter_pack.sh
```

Then launch training on your GPU:

```
./train_gpt2_fp32
```

If CUDA is installed correctly, the program will detect your GPU and start training. You'll see logs that look similar to the CPU trainer's, but with much shorter step times. For example, a training step that took ~2 seconds on CPU might take ~50 milliseconds on GPU.

Under the Hood

Although the training loop looks the same on the surface, a lot changes under the hood when running on GPU:

1. Tensors are allocated in GPU memory (not system RAM).
2. Matrix multiplications (the core of attention and MLP layers) are executed by cuBLAS, NVIDIA's high-performance linear algebra library.
3. Kernels for elementwise operations (like adding residuals, applying softmax, or normalizing) are written in CUDA or use built-in primitives.
4. Gradients and optimizer states are updated entirely on the device, with minimal CPU GPU transfers.

This makes training dramatically faster, but the structure of the code is still recognizable compared to the CPU version.

When to Use FP32 vs Mixed Precision

- Use FP32 (this path) when:
  - You're learning how GPU training works step by step.
  - You want a clean comparison with the CPU trainer.
  - You're debugging correctness issues without worrying about loss scaling.
- Use Mixed Precision (`train_gpt2.cu`) when:
  - You want real performance (2–4× faster training).
  - You're training larger models (774M, 1.6B parameters) where memory efficiency matters.
  - You're aiming to reproduce published GPT-2 runs on modern GPUs.

Common Pitfalls

1. CUDA not installed → If `nvcc` isn't found, the Makefile will fail. You'll need the CUDA Toolkit installed.
2. Driver mismatch → Your NVIDIA driver must match the CUDA version.
3. Out of memory errors → FP32 uses more GPU memory, so you may need to lower batch size if you're on a smaller GPU.

Why This Step Matters

The FP32 trainer is like a bridge:

- On one side is the CPU reference path, slow but crystal-clear.
- On the other is the mixed-precision CUDA path, fast but more complex.

By walking across this bridge, you learn how GPU acceleration works without being overwhelmed by optimizations.

The takeaway: `train_gpt2_fp32.cu` is your first taste of real GPU training in *llm.c*. It skips advanced tricks and shows you a clean, one-GPU, full-precision implementation. It's not the fastest, but it's the friendliest way to understand how training moves from CPU to GPU.

6. Quickstart: Modern CUDA Path (`train_gpt2.cu`)

This is the high-performance trainer most people use day to day. It runs on a single NVIDIA GPU (and also forms the basis for multi-GPU), uses mixed precision (FP16/BF16 where safe, FP32 where needed), and can optionally enable cuDNN FlashAttention for fast attention. Compared to the FP32 legacy path, it's significantly faster and uses less memory, while keeping the training loop easy to follow.

What "mixed precision" means (in plain words)

- Weights & activations: stored/processed in FP16 or BF16 for speed and lower memory.
- Master weights: a FP32 copy of parameters kept for stable updates.
- Loss scaling: multiply the loss before backward to avoid underflow; unscale the grads before the optimizer step.
- Autocast-like behavior: the code picks safe dtypes for each op (GEMMs in tensor cores, reductions in FP32, etc.).

You get 2–4× speedups on many GPUs, and the same final accuracy when configured properly.

Build the modern CUDA trainer

```
make train_gpt2cu
```

Common variants:

- With cuDNN FlashAttention (if available):
  ```
  make train_gpt2cu USE_CUDNN=1
  ```
- With debug symbols (slower, but easier to step through):
  ```
  make train_gpt2cu DEBUG=1
  ```

This produces an executable named `train_gpt2cu`.

One-time data & artifacts

If you haven't already:

```
./dev/download_starter_pack.sh
```

This fetches the tokenizer and a small dataset so you can run immediately.

Run your first GPU training session

```
./train_gpt2cu
```

You should see a config header (model dims, vocab, sequence length) followed by step-by-step loss prints. Step times will be much shorter than CPU and noticeably faster than FP32, especially on tensor-core GPUs (Turing and newer).

Speed tips right away:

- Use a larger global batch if memory allows-it improves GPU utilization.
- Set environment threads for any CPU preprocessing:
  ```
  OMP_NUM_THREADS=8 ./train_gpt2cu
  ```

What's different under the hood vs. FP32

- Tensor cores: GEMMs run in FP16/BF16 paths via cuBLAS/cuBLASLt for big throughput.
- Scaled loss & unscale pass: Forward computes the loss, multiplies it by a scale factor; backward divides gradients by the same factor before updates.
- Master FP32 copy: Optimizer (AdamW) updates this copy, then casts back to low precision for the next forward.
- Fused/fast attention (optional): With `USE_CUDNN=1`, attention may route through cuDNN FlashAttention backends.

You still recognize the same loop: load batch → forward → loss → backward → AdamW step → log.

Choosing FP16 vs. BF16

- FP16: best speed, needs loss scaling; widely supported.
- BF16: more numerically forgiving (often needs little/no scaling), requires hardware support (Ampere+); slightly larger memory than FP16 but often simpler.

The trainer picks what your GPU supports or what the code defaults to; you can expose a flag later if you want to force one.

Common command patterns

- Small GPU (less VRAM):

```
./train_gpt2cu --batch_size 4 --micro_batch_size 1 --seq_len 512
```

- Faster warmup with cosine schedule:

```
./train_gpt2cu --warmup_steps 1000 --lr 6e-4 --scheduler cosine
```

- Periodic eval to sanity-check:

```
./train_gpt2cu --eval_interval 200 --eval_batches 50
```

(Flag names above mirror typical patterns; adjust to match the binary's printed help.)

Validating correctness (highly recommended)

- Run the CUDA test binary to compare against the PyTorch reference on small batches:

```
make test_gpt2cu
./test_gpt2cu
```

- Check that logits/loss match within a small tolerance. If mismatches happen, recompile without optimizations or disable cuDNN fast paths (`USE_CUDNN=0`) to isolate the issue.

Enabling FlashAttention (when available)

```
make train_gpt2cu USE_CUDNN=1
./train_gpt2cu
```

Good signs: faster attention time and lower step latency. If you hit build/runtime errors, ensure your CUDA, cuDNN, and driver versions are compatible; fall back to `USE_CUDNN=0` while you sort it out.

Memory & performance tuning checklist

- Batching: Increase `micro_batch_size` until you reach ~90% GPU utilization without OOM.
- Sequence length: Longer sequences increase compute quadratically in attention; reduce `--seq_len` if memory is tight.
- Grad accumulation: Keep global batch size large by accumulating over multiple micro-batches.
- Pinned host memory & async copies: Already used where sensible; keep CPU GPU transfers minimal.
- Profiler: Once it runs, profile hotspots to confirm GEMMs dominate (as expected) and attention isn't a bottleneck unless FlashAttention is off.

Troubleshooting

- `cudaErrorNoKernelImageForDevice`: Toolkit too new/old for your GPU; rebuild with proper `-arch=` or update drivers.
- `CUBLAS_STATUS_ALLOC_FAILED` / OOM: Lower batch size, sequence length, or switch to BF16 if supported.
- Diverging loss with FP16: Increase loss scale (if configurable) or try BF16; confirm master-weight updates are in FP32.

- cuDNN errors: Rebuild without `USE_CUDNN` to verify the base path works, then revisit versions/paths.

The takeaway: `train_gpt2.cu` is the practical, fast trainer: mixed precision, optional FlashAttention, and ready to scale. You keep the same readable training loop while tapping your GPU's tensor cores for large speedups and much better memory efficiency.

7. Starter Artifacts & Data Prep (`dev/download_starter_pack.sh`, `dev/data/`)

Before you can actually train or test a model in *llm.c*, you need a few essential artifacts: the tokenizer, a dataset, and a config. These files aren't stored in the repo directly (they're too large and often under different licenses), so the project provides scripts to fetch or generate them. This is where the `dev/` folder comes into play.

The Starter Pack Script

The easiest way to get going is with:

```
./dev/download_starter_pack.sh
```

This script pulls down a ready-made bundle containing:

- `gpt2_tokenizer.bin` - The GPT-2 byte-pair encoding (BPE) tokenizer in binary format.
- `train.bin` / `val.bin` - Pre-tokenized training and validation datasets, often based on OpenWebText or Tiny Shakespeare for demos.
- Model configs - A JSON or header file that sets hyperparameters like layers, hidden size, and number of heads for GPT-2 124M.

Think of this as your "starter kit": it contains just enough to run a demo and see training loss decreasing without setting up a full-scale dataset pipeline yourself.

The Tokenizer File (`gpt2_tokenizer.bin`)

This is a binary representation of GPT-2's tokenizer vocabulary. It maps raw text (like `"Hello world"`) into integer token IDs, which are the actual inputs to the model.

- Why binary? It's faster to load in C than parsing a text-based vocabulary.
- Size? ~500 KB, representing ~50,000 tokens.
- Role in training? Used in both the dataloader (to prepare inputs) and the sampler (to decode outputs).

Without this file, the model can't understand text at all-it would just be manipulating meaningless numbers.

Dataset Files (`train.bin`, `val.bin`)

Each dataset file is a binary blob containing:

1. A header (about 1 KB) describing sequence length, vocab size, and other metadata.
2. A stream of token IDs (`uint16`), representing the text corpus already tokenized.

This design means the C dataloader can simply `fread()` chunks of tokens into memory, without needing to tokenize text on the fly. It's fast and memory-efficient, perfect for a lean project like *llm.c*.

The script usually fetches two versions:

- Training set (`train.bin`)
- Validation set (`val.bin`)

That way, the training loop can occasionally switch to validation mode and report a validation loss, helping you track overfitting.

The `dev/data/` Folder

If you want to generate your own datasets, this is where you'll find the tools:

- Scripts for Tiny Shakespeare, OpenWebText, or other corpora.
- Utilities to tokenize text using the GPT-2 tokenizer and write out the `.bin` format.
- Small Python snippets to check dataset statistics (like number of tokens or average sequence length).

For example, if you wanted to try fine-tuning GPT-2 on your own text files, you'd:

1. Run a preprocessing script in `dev/data/` to tokenize and save your corpus.
2. Point `train_gpt2.c` or `train_gpt2.cu` to your new `train.bin` and `val.bin`.
3. Kick off training as usual.

Why Preprocessing Matters

Tokenization and dataset preparation can be surprisingly heavy in Python, especially for large corpora. By precomputing everything into compact `.bin` files, *llm.c* keeps the runtime training loop as simple as possible-just reading arrays of integers and feeding them into the model.

This separation of concerns (preprocessing vs. training) is what makes the training code clean and focused.

Quick Sanity Check

After running `download_starter_pack.sh`, you should see these files in your working directory:

```
gpt2_tokenizer.bin
train.bin
val.bin
```

If any are missing, re-run the script. Without them, the trainer will exit with a file-not-found error.

The takeaway: The starter pack is your ticket to running *llm.c* right away. It gives you a tokenizer and datasets in exactly the format the C code expects. Later, when you're ready to train on your own text or scale up, the `dev/data/` folder shows you how to prepare custom datasets the same way.

8. Debugging Tips & IDE Stepping (`-g`)

Even though *llm.c* is designed to be small and readable, training a transformer model is still a big program with lots of moving parts. When something goes wrong-whether it's a segmentation fault, `NaN` losses, or unexpected results-you'll want to be able to debug effectively. That's where debug builds and IDE stepping come in.

Why Debug Mode Exists

By default, the Makefile compiles with heavy optimization (`-O3`). That makes the code run fast, but it also makes debugging harder:

- Variables may be optimized away.
- Functions might get inlined so you can't step through them clearly.
- The debugger may jump around unpredictably.

Adding the `-g` flag (enabled with `DEBUG=1`) tells the compiler to include extra information in the binary so you can see exactly what the code is doing at runtime.

Building a Debug Binary

To build with debug info:

```
make train_gpt2 DEBUG=1
```

This produces a slower executable, but one that works seamlessly with tools like:

- gdb - the classic GNU debugger.
- lldb - default on macOS.
- VS Code / CLion / Xcode - IDEs with integrated debuggers and GUI interfaces.

Using gdb on Linux/macOS

Start your program under gdb:

```
gdb ./train_gpt2
```

Inside gdb:

- Run the program: `run`
- Set a breakpoint at `main`: `break main`
- Step through line by line: `step` or `next`
- Inspect variables: `print loss`, `print i`
- Quit: `quit`

This is the fastest way to see exactly where a crash happens.

Using an IDE

If command-line debugging feels intimidating, you can use an IDE like VS Code or CLion:

- Open the project folder.
- Configure the debugger (choose `gdb` or `lldb` backend).
- Add breakpoints by clicking next to line numbers.
- Run the debug build (`train_gpt2` with `DEBUG=1`).
- Step through forward pass, backward pass, or optimizer updates.

This way, you can visually watch variables update with each step.

Debugging CUDA Code

CUDA debugging is a bit trickier, but still possible:

- `cuda-gdb` - NVIDIA's GPU debugger, works like gdb but supports stepping into kernels.
- Nsight Systems / Nsight Compute - graphical profilers/debuggers that let you trace kernel launches, memory transfers, and GPU utilization.

If your CUDA code crashes with cryptic messages like `illegal memory access`, `cuda-gdb` can help pinpoint the kernel and even the exact line.

Debugging Common Issues in llm.c

1. File not found → Make sure `gpt2_tokenizer.bin`, `train.bin`, and `val.bin` are downloaded.
2. Segfault at malloc/fread → Check file paths and dataset sizes.
3. Loss becomes NaN →
    - On CPU: check for division by zero in normalization.
    - On GPU: check loss scaling (mixed precision) or try FP32 path for comparison.
4. Mismatch with PyTorch tests → Run `test_gpt2` or `test_gpt2cu` and compare outputs; this usually isolates whether the bug is in forward pass, backward pass, or optimizer.

Logging & Sanity Checks

When debugging, it helps to add extra logging. The repo already has a lightweight logger, but you can also sprinkle `printf`s (on CPU) or `cudaDeviceSynchronize(); printf(...)` (on GPU) to track values. For example:

```
printf("Step %d: loss=%f\n", step, loss);
```

Sometimes the quickest fix is just to print and see what's going on.

Best Practices for Beginners

- Start with the CPU build when learning-it's easier to debug than CUDA.
- Always keep a small dataset (like Tiny Shakespeare) for fast iteration.
- Compare against the PyTorch reference for the same batch to catch subtle errors.
- Use `DEBUG=1` whenever you hit strange behavior-you'll trade speed for clarity, which is usually worth it when learning.

The takeaway: Debug builds (`-g`) turn *llm.c* from a black box into a step-through learning tool. With gdb, lldb, or an IDE, you can pause at any line, inspect variables, and understand exactly how GPT-2 training works inside C or CUDA. It's slower, but it's the clearest way to learn and fix issues.

9. Project Constraints & Readability Contract

The *llm.c* project isn't trying to be the fastest or most feature-rich GPT-2 trainer. Instead, it has a very deliberate set of constraints-rules the author imposes on the codebase to keep it approachable and educational. You can think of these as the "contract" between the code and the reader: certain things are kept simple on purpose, even if they cost some performance.

Minimalism over Optimizations

- No processor-specific intrinsics: You won't see AVX, NEON, or other hardware-tuned assembly calls in the CPU path.
- No fancy template metaprogramming: Unlike in C++ frameworks, here you get plain C structs and functions.
- No exotic libraries: Aside from cuBLAS/cuDNN for GPU acceleration, most functionality is implemented directly.

This means the code runs almost anywhere, and you don't need to understand deep compiler tricks to follow what's going on.

Transparency over Abstraction

- Every operation is visible in the source. For example, instead of calling a framework function like `nn.CrossEntropyLoss`, you'll find an explicit forward and backward pass coded in C.
- Data loading, tokenization, optimizer steps, and schedulers are all implemented as separate, small modules in `llmc/`.
- You don't need to guess what's happening-if you're curious, you can open the corresponding `.h` file and see the exact code.

The guiding idea: if something is central to training GPT-2, you should be able to read and understand it.

Performance Where It Matters (but No More)

- OpenMP pragmas are allowed in CPU builds, because they give large speedups with minimal extra code.
- cuBLAS/cuDNN are used for GPU matmuls and attention, because re-implementing them would be a distraction and would make the project impossibly large.
- But the project avoids unnecessary complexity-no kernel fusion, no elaborate caching layers, no half-implemented "framework" abstractions.

This balance ensures you can still run experiments at a reasonable speed, but the code never sacrifices readability.

Educational First

The code is written to teach, not to win benchmarks. That means:

- Variable names are descriptive, not cryptic.
- Comments explain not just *what* happens, but also *why*.
- Files are kept small and focused, rather than sprawling across dozens of layers of abstraction.
- There's a matching PyTorch reference implementation so you can always check your understanding against a familiar baseline.

Limitations You Should Expect

- Training is slower than PyTorch/XLA/JAX or DeepSpeed-tuned runs.
- Multi-GPU scaling is functional but not heavily optimized.
- Only GPT-2 architectures are covered-don't expect GPT-3 or transformer variants.
- Features like dataset streaming, checkpoint sharding, or advanced distributed tricks are intentionally left out.

These are not bugs-they're conscious trade-offs to keep the codebase small, sharp, and didactic.

Why This Matters for You

If you're learning how transformers work, this contract is a gift:

- You won't get lost in performance hacks.
- You won't fight through an abstraction jungle.
- You'll always know that what you're reading is close to the "pure" algorithmic idea.

On the flip side, if you're aiming for production-grade speed, you'll need to layer more on top. But that's outside the mission of *llm.c*.

The takeaway: *llm.c* is bound by a readability contract: clarity over raw speed, transparency over abstraction, minimalism over complexity. These constraints keep the project small enough to fit in your head, while still powerful enough to reproduce GPT-2 training. It's a teaching lab, not a racing car-and that's exactly why it's valuable.

10. Community, Discussions, and Learning Path

The last piece of the quickstart isn't about code at all-it's about the people and resources around the project. *llm.c* has grown into more than just a single repository; it has become a meeting point for learners, tinkerers, and researchers who want to strip large language models down to their essentials. Understanding this community layer is just as important as understanding the code itself.

Discussions and Issues on GitHub

The project's Discussions tab is full of valuable context:

- Developers asking about build errors on different platforms (Linux, macOS, Windows).
- Explorations of how to extend *llm.c* to train larger GPT-2 models (355M, 774M, 1.6B).
- Reports on multi-GPU and MPI runs, including troubleshooting NCCL hangs and performance bottlenecks.
- Debates on mixed precision vs FP32 vs BF16 stability.

Reading these threads is like looking over the shoulders of hundreds of other learners. You'll see not only the official answers but also the thought process of people solving problems in real time.

Roadmap and Contributions

The README and issues sometimes hint at where the project might grow:

- Making the CUDA kernels more modular in `dev/cuda/`.
- Simplifying multi-GPU startup for clusters.
- Adding small tutorial-style docs (like the LayerNorm walkthrough).

The project is open to contributions, but it follows the same minimalist philosophy. If you're thinking of contributing, remember: the goal is clarity first, performance second.

External Learning Resources

While *llm.c* is self-contained, it pairs nicely with outside material:

- The PyTorch reference implementation in `train_gpt2.py` is your canonical oracle for correctness.
- The GPT-2 paper gives the architecture background.
- CUDA and cuBLAS/cuDNN docs explain the GPU APIs that the project calls into.
- Community blog posts often walk through specific sections of the code in plain English, making it easier to digest.

By combining the code, the paper, and these resources, you can triangulate a much deeper understanding.

A Suggested Learning Path

If you're coming to *llm.c* as a beginner, here's a natural progression:

1. Run the CPU trainer (`train_gpt2.c`) on Tiny Shakespeare. Watch the loss decrease.
2. Step through the code with `DEBUG=1`, confirming that you understand forward, backward, and optimizer steps.
3. Move to the FP32 CUDA trainer to see how the same loop runs on GPU.
4. Switch to the modern CUDA trainer (`train_gpt2.cu`) and learn how mixed precision works.
5. Experiment with dataset scripts in `dev/data/`-try your own text corpus.
6. Read the LayerNorm doc in `doc/` to deepen your theory-practice connection.
7. Explore multi-GPU runs with MPI/NCCL if you have access to multiple GPUs.
8. Follow GitHub Discussions for real-world debugging and scaling stories.

Why This Matters

Code alone is not enough. The community context, the discussions, and the learning path make *llm.c* a living project. By engaging with them, you avoid the feeling of learning in isolation. You'll see others wrestling with the same challenges, and you'll have a clearer sense of what to try next.

The takeaway: Beyond the files and scripts, *llm.c* is a community-driven learning environment. GitHub issues, discussions, reference docs, and external tutorials all form part of the "extended classroom." If the code is the lab bench, the community is the set of lab partners who help you figure things out along the way.

Chapter 2. Data, Tokenization, and Loaders

11. GPT-2 Tokenizer Artifacts (`gpt2_tokenizer.bin`)

A language model like GPT-2 doesn't directly understand English, Vietnamese, or any other natural language. Instead, it understands numbers. These numbers are called tokens. A tokenizer is the tool that translates between human text and tokens. In *llm.c*, the GPT-2 tokenizer is stored in a small file called `gpt2_tokenizer.bin`. This file is the key that lets the model read input text and produce output text that we can understand.

What This File Contains

The file `gpt2_tokenizer.bin` is a binary version of GPT-2's tokenizer. It includes:

Component	Purpose
Byte vocabulary (0–255)	Makes sure every possible character can be represented.
Merge rules (BPE)	Combines frequent sequences like "ing" or " the" into single tokens for efficiency.
Vocabulary size (~50,257)	Defines how many distinct tokens GPT-2 can work with.
Mapping IDs text	Lets the program turn model outputs (numbers) back into human-readable strings.

Instead of being written as JSON or text, the tokenizer is stored in binary form. This allows *llm.c* to load it very quickly using a simple file read, which keeps the code clean and fast.

Where It Comes From

You don't need to build this file by hand. The repository provides a script to download it, along with small training and validation datasets:

```
./dev/download_starter_pack.sh
```

After running the script, you should see `gpt2_tokenizer.bin`, `train.bin`, and `val.bin` in your working directory. If the tokenizer is missing, the program cannot run because it won't know how to interpret text.

How the Code Uses It

During training, the tokenizer is not active because the datasets (`train.bin` and `val.bin`) are already pre-tokenized into integers. This keeps the training loop fast and simple.

During sampling or evaluation, the tokenizer becomes important again. After the model predicts a sequence of token IDs, the tokenizer translates those numbers back into text that you can read on your screen.

The C API for the tokenizer, defined in `llmc/tokenizer.h`, provides just three main functions:

```
int tokenizer_init(Tokenizer *t, const char *filename);
int tokenizer_decode(Tokenizer *t, const int *ids, int n, char *out);
void tokenizer_free(Tokenizer *t);
```

This is all you need: initialize the tokenizer from the file, decode tokens into text, and free memory when done.

Example Workflow in Practice

1. Initialize the tokenizer:

   ```
   Tokenizer t;
   tokenizer_init(&t, "gpt2_tokenizer.bin");
   ```

2. Decode a sequence of tokens back to text:

   ```
   char buf[512];
   tokenizer_decode(&t, tokens, ntokens, buf);
   printf("%s\n", buf);
   ```

3. Clean up memory when you no longer need it:

```
tokenizer_free(&t);
```

This small cycle is enough to turn model outputs into readable sentences.

Why It Matters

Without the tokenizer, the model cannot communicate. The tokenizer is like a shared dictionary between humans and the neural network. If you give the model text, the tokenizer converts it into numbers the model understands. When the model responds, the tokenizer converts its numbers back into text. If the tokenizer does not match the dataset, the model's predictions will come out as gibberish. Keeping the tokenizer and dataset in sync is essential for correct training and evaluation.

Try It Yourself

Here are a few small exercises you can do to understand the tokenizer better:

1. Check that the file exists: After running the starter pack script, verify that `gpt2_tokenizer.bin` is in your directory. Try running the trainer without it and observe the error message.
2. Inspect vocab size: Run the trainer and look for the line that prints `vocab_size: 50257`. Compare this with `padded_vocab_size: 50304`. Why do you think padding helps GPUs?
3. Decode a sequence manually: Write a short C program that loads the tokenizer and decodes a fixed list of token IDs (for example `[464, 3290, 318]`). Observe what text you get.
4. Mismatch experiment: If you build your own dataset with a different tokenizer (say, a custom vocabulary), try decoding it with `gpt2_tokenizer.bin`. Notice how the output becomes meaningless, showing why consistency matters.
5. Dataset + tokenizer link: Open `train.bin` in a hex viewer. You'll see it's just numbers. Use the tokenizer to decode the first few hundred tokens and see real text emerge.

The Takeaway

`gpt2_tokenizer.bin` is a tiny but vital file. It is the bridge that allows the model and humans to speak the same language. Training is efficient because all data is pre-tokenized, and when you want to see what the model has written, the tokenizer turns raw numbers back into words. Without it, the entire system would be silent.

12. Binary Dataset Format (`train.bin` and `val.bin`)

Just like the tokenizer turns text into numbers, the datasets in *llm.c* are stored as numbers too. Instead of reading plain text files like `.txt`, the training and validation data are kept in simple binary files: `train.bin` and `val.bin`. These files are the fuel for the training loop.

What These Files Look Like

At first glance, `train.bin` and `val.bin` look like unreadable blobs if you open them in a text editor. That's because they are not meant to be human-readable. They contain:

Part	Description
A tiny header (about 1 KB)	Stores metadata such as sequence length and vocab size.
A stream of token IDs (`uint16`)	Every tokenized word piece from the dataset, saved as 16-bit integers.

Each integer represents one token from the tokenizer's vocabulary. Since GPT-2 has a vocabulary of about 50,000 tokens, 16-bit integers (`uint16_t`) are enough to store them all.

Why Binary Format?

- Efficiency: Instead of re-tokenizing text every time, the data is pre-tokenized once and stored as numbers. The trainer just reads them directly.
- Speed: Reading integers from a file is faster than parsing and processing raw text.
- Simplicity: The training loop only has to deal with arrays of integers-no string handling, no parsing, no surprises.

This choice makes the training code in *llm.c* much cleaner and faster.

How the Dataloader Uses Them

When training starts, the dataloader reads chunks of numbers from `train.bin`. Each chunk corresponds to one batch of size B $\times$ T:

- B = batch size (number of examples in a batch).
- T = sequence length (number of tokens per example).

For example, if `B = 8` and `T = 1024`, the dataloader will read `8 × 1024 = 8192` token IDs from the file, reshape them into sequences, and feed them to the model.

The validation file (`val.bin`) works the same way but is only used occasionally during training to measure validation loss. This helps detect overfitting.

Workflow in Code

Inside the repo, you'll see functions like these in `llmc/dataloader.h`:

```
int dataloader_init(Dataloader *loader, const char *filename, int B, int T);
int dataloader_next_batch(Dataloader *loader, int *inputs, int *targets);
void dataloader_reset(Dataloader *loader);
void dataloader_free(Dataloader *loader);
```

Here's what happens step by step:

1. Initialize with the binary file and batch/sequence sizes.
2. Next batch reads the next B × T tokens into an input array and a target array.
3. Reset allows re-reading from the beginning when you start a new epoch.
4. Free cleans up resources when training ends.

The target array is simply the same sequence shifted by one token-because language modeling predicts the *next* token.

Why It Matters

The dataset format is what makes *llm.c* practical. Without it, the code would need to handle messy text, encodings, and tokenization during every training step. By storing clean arrays of token IDs, the training loop becomes very short and easy to follow. It's a design decision that keeps the project minimal yet faithful to real training pipelines.

Try It Yourself

1. Check file size: Run `ls -lh train.bin` and notice how large it is compared to a plain `.txt` file. Why is it smaller or larger?
2. Peek inside: Use a hex viewer (`xxd train.bin | head`) to see raw numbers. They won't look like text, but they are the tokens the model trains on.
3. Count tokens: Write a short Python or C script to count how many token IDs are stored in `train.bin`. This gives you a sense of dataset size.
4. Mini-dataset: Try generating your own dataset from a small `.txt` file using the scripts in `dev/data/`. See how the `.bin` file is created.
5. Validation experiment: During training, reduce the validation set to only a few batches and observe how the validation loss stabilizes or fluctuates compared to training loss.

The Takeaway

`train.bin` and `val.bin` may look like gibberish, but they are carefully prepared binary files containing token IDs. They make training faster, simpler, and more reproducible. The dataloader in *llm.c* reads these numbers in neat chunks and serves them directly to the model, letting you focus on learning how transformers work instead of wrestling with raw text parsing.

13. Dataset Scripts in `dev/data/`

The repository doesn't just give you ready-made binary datasets like `train.bin` and `val.bin`. It also provides scripts inside the `dev/data/` folder that show you how to create your own. These scripts are important because they demonstrate how raw text gets transformed into the binary format that the dataloader in *llm.c* expects.

What's Inside `dev/data/`

This folder contains small Python scripts that:

Script	Purpose
`prepare_shakespeare.py`	Turns the Tiny Shakespeare dataset into `train.bin` and `val.bin`.
`prepare_openwebtext.py`	Prepares a large-scale dataset similar to the one GPT-2 was trained on.
Other helpers	Tokenize raw `.txt` files, split them into train/val, and save to binary.

Each script follows the same basic recipe:

1. Read raw text from a source file.
2. Apply the GPT-2 tokenizer to turn text into token IDs.
3. Split the tokens into training and validation portions.
4. Write the IDs into binary files that *llm.c* can read directly.

Why Preprocessing Happens Outside C

In C, handling text files with Unicode, punctuation, and different encodings is messy. Instead, preprocessing is done once in Python, where tokenizers are easier to use. The results are saved in a simple binary format (`uint16` IDs). From then on, C only has to deal with arrays of integers-clean and efficient.

This design keeps the training loop minimal: no text parsing, no string handling, just numbers.

Example: Tiny Shakespeare

One of the simplest datasets is Tiny Shakespeare, about 1 MB of text from Shakespeare's plays. The script `prepare_shakespeare.py` will:

- Read `input.txt` (the raw text).
- Use the GPT-2 tokenizer (`gpt2_tokenizer.bin`) to turn every word and symbol into token IDs.
- Split 90% of the data into `train.bin` and 10% into `val.bin`.

After running the script, you'll have small binary files that let you train GPT-2 from scratch in minutes on CPU or GPU.

Example: OpenWebText

The script `prepare_openwebtext.py` shows how to tokenize a much larger dataset, closer to what GPT-2 was originally trained on. This is heavier and requires more disk space, but it's useful if you want to try scaling up training to bigger models.

Why It Matters

These scripts are more than convenience tools-they are examples of how to adapt llm.c to your own data. If you have a collection of emails, poems, or programming code, you can:

1. Put them into a single `.txt` file.
2. Modify one of the scripts in `dev/data/`.
3. Generate new `train.bin` and `val.bin` files.
4. Train GPT-2 on your own text.

By separating dataset creation from training, *llm.c* keeps the C code small and makes experimentation flexible.

Try It Yourself

1. Run the Shakespeare script:

   ```
   python dev/data/prepare_shakespeare.py
   ```

   Then check that `train.bin` and `val.bin` were created.

2. Open the binary files with a hex viewer and confirm that they contain only numbers.

3. Modify the script to tokenize a different text file (for example, your own writing).

4. Compare dataset sizes: Tiny Shakespeare is tiny (MBs), OpenWebText is huge (GBs). Observe how training speed changes depending on dataset size.
5. Re-run training with your custom dataset and watch how the model starts generating text in your style.

The Takeaway

The `dev/data/` scripts are the bridge between raw human text and the binary datasets used in training. They let you prepare small demo datasets or scale up to larger corpora. By experimenting with these scripts, you learn how to bring your own data into *llm.c* and train a GPT-style model on anything you like.

14. DataLoader Design (Batching, Strides, Epochs)

Now that the datasets are prepared as `.bin` files, we need a way to feed them into the model during training. This is the job of the DataLoader in *llm.c*. You'll find its interface in `llmc/dataloader.h`, and its purpose is very simple: take a big stream of token IDs from `train.bin` or `val.bin`, cut it into manageable chunks, and serve those chunks to the training loop as batches.

The Core Idea

Training a language model requires two arrays for every batch:

- Inputs: a sequence of token IDs, like `[The, cat, sat, on]`
- Targets: the same sequence shifted by one, like `[cat, sat, on, the]`

The model learns to predict each next token in the sequence. The DataLoader automates slicing these arrays out of the giant dataset file.

The Interface

In the code you'll see function declarations like these:

```
int dataloader_init(Dataloader *loader, const char *filename, int B, int T);
int dataloader_next_batch(Dataloader *loader, int *inputs, int *targets);
void dataloader_reset(Dataloader *loader);
void dataloader_free(Dataloader *loader);
```

Here's what each does:

- `dataloader_init`: opens the dataset file, remembers batch size `B` and sequence length `T`.
- `dataloader_next_batch`: returns the next chunk of `B × T` tokens (inputs) and their shifted version (targets).
- `dataloader_reset`: rewinds to the start of the file when an epoch ends.
- `dataloader_free`: closes the file and releases memory.

This design keeps the training loop clean: just call `next_batch` and you get the data ready for forward/backward passes.

B × T Explained

The two most important parameters are:

Symbol	Meaning	Example
B	Batch size (how many sequences per step)	16
T	Sequence length (how many tokens per sequence)	1024

So one batch contains `B × T` tokens. For example, with `B = 16` and `T = 1024`, each batch holds 16,384 tokens. The DataLoader simply reads that many numbers from the binary file and arranges them in memory.

Strides Through the Dataset

As you call `dataloader_next_batch`, the loader moves forward through the dataset by `B × T` tokens each time. When it reaches the end of the dataset file, it either:

- Resets back to the beginning (`dataloader_reset`), or
- Switches from training to validation, depending on the training loop's needs.

This stride-based reading is efficient: no random access, just sequential reads from a file.

Epochs and Shuffling

In deep learning, an epoch means one full pass through the dataset. The DataLoader in *llm.c* is simple: it goes linearly from start to finish. It doesn't shuffle data like PyTorch's `DataLoader`. Why? Because language data is already very diverse, and the project values minimal code over extra features. If you want shuffling, you can preprocess the dataset differently before creating `.bin` files.

Why It Matters

The DataLoader is the quiet workhorse of training. It ensures that every step sees a fresh batch of token sequences, always with matching inputs and targets. By separating dataset reading from the training loop, the code stays clean and focused. This design also makes it easy to swap datasets-once you generate a `.bin` file, the loader doesn't care where it came from.

Try It Yourself

1. Print the first batch: Modify the code to print the first 20 input tokens and their targets. See how each input token aligns with the next target token.
2. Experiment with B and T: Set `B = 2` and `T = 8` and observe how the loader slices the dataset into tiny chunks. Then try larger values and see how memory usage changes.
3. Check epoch length: Write a small loop to count how many batches you get before `dataloader_reset` is called. Does this match the total tokens divided by `B × T`?
4. Validation check: Observe how often the training loop switches to `val.bin`. How does validation loss compare to training loss over time?
5. Custom stride: Modify the code so the DataLoader skips some tokens between batches. What effect does this have on training?

The Takeaway

The DataLoader in *llm.c* is intentionally simple. It streams token IDs in fixed-sized batches, moves forward stride by stride, and resets when done. This straightforward design avoids complexity and keeps the focus on the model itself, while still teaching you the essential mechanics of batching and sequence handling in language model training.

15. EvalLoader and Validation Workflow

Training a model isn't just about watching the training loss go down. To know whether the model is actually learning patterns that generalize-and not just memorizing the training data-you need to run validation. In *llm.c*, validation is handled by a component called the EvalLoader, which works just like the DataLoader but reads from the validation dataset (`val.bin`) instead of the training dataset (`train.bin`).

Why We Need Validation

Imagine teaching a student only by drilling them with the same math problems over and over. They might get really good at those problems, but fail completely when given new ones.

Validation is like giving the student a pop quiz with unseen questions. If they do well, you know they've actually learned the concepts.

For language models, validation helps detect overfitting: when the training loss keeps improving but the validation loss stays flat or even gets worse.

How EvalLoader Works

EvalLoader lives in the same code file as the DataLoader (`llmc/dataloader.h`), but it points to a different dataset file. Its workflow is nearly identical:

1. Open `val.bin` and prepare for reading.
2. Serve up batches of size `B × T` (batch size × sequence length).
3. Provide inputs and targets the same way as the training DataLoader.
4. Reset after one full pass through the file.

The training loop typically calls the EvalLoader at intervals-for example, every few hundred steps-so you get a snapshot of validation loss during training.

What Happens During Validation

When validation is triggered:

1. The current model parameters are frozen (no gradient updates).
2. A few batches are read from `val.bin`.
3. The model runs forward passes only, computing the loss on each batch.
4. The losses are averaged and reported as validation loss.

This doesn't take long because it usually samples just a subset of the validation dataset, not the entire file.

Training Loop with Validation

In pseudocode, the loop looks like this:

```
for (step = 0; step < max_steps; step++) {
    dataloader_next_batch(&train_loader, inputs, targets);
    forward_backward_update(model, inputs, targets);

    if (step % eval_interval == 0) {
        float val_loss = 0.0f;
        for (int i = 0; i < eval_batches; i++) {
```

```
            evalloader_next_batch(&val_loader, inputs, targets);
            val_loss += forward_only(model, inputs, targets);
        }
        val_loss /= eval_batches;
        printf("step %d: val loss %.4f\n", step, val_loss);
    }
}
```

This is simplified, but it shows the idea: the validation loop is nested inside the training loop, running occasionally instead of every step.

Why It Matters

Validation is the reality check of training. Without it, you could train forever and celebrate low training losses, only to discover that your model produces nonsense on new text. By tracking validation loss, you can:

- Detect overfitting early.
- Adjust hyperparameters (like learning rate or batch size).
- Know when training has plateaued and it's time to stop.

In professional setups, validation curves are often plotted live, but in *llm.c*, the minimalist approach is to just print numbers to the console.

Try It Yourself

1. Watch val loss: Run training and note how validation loss compares to training loss. Do they both decrease together?
2. Overfitting demo: Train on a very tiny dataset (like 10 KB of text). Notice how training loss plummets but validation loss stalls or rises.
3. Change eval interval: Reduce `eval_interval` so validation runs every step. How much slower does training feel?
4. Change eval batches: Set `eval_batches` to 1 vs 100. What difference does this make in the stability of the reported validation loss?
5. Validation as stopping rule: Stop training when validation loss stops improving for many intervals. How does this affect final performance?

The Takeaway

The EvalLoader is a twin of the DataLoader, but for validation. It feeds the model data it has never seen during training, and the resulting validation loss tells you whether your model is learning useful patterns or just memorizing. It's the simplest safeguard against wasted compute, and it's an essential part of every training loop-even in the stripped-down world of *llm.c*.

16. Sequence Length and Memory Budgeting

When training GPT-2 in *llm.c*, one of the most important decisions you make is choosing the sequence length (often called T). This value determines how many tokens the model processes in a single forward pass. It might sound like just another parameter, but sequence length has a huge impact on what the model can learn, how much memory it uses, and how fast training runs.

What Sequence Length Means

Sequence length is simply the number of tokens per training example. If `T = 1024`, the model reads 1,024 tokens in a row (like words or subwords) and tries to predict the next token at each position.

Think of it like this: if you give the model a paragraph of text, sequence length is how much of that paragraph it sees at once. Shorter lengths give the model less context, while longer lengths allow it to capture bigger patterns, like whole paragraphs or even multiple pages.

Where It Appears in the Code

In the logs, you'll often see lines like:

```
max_seq_len: 1024
```

This number is defined in the GPT-2 configuration and passed into the DataLoader. The DataLoader slices chunks of exactly `T` tokens from `train.bin` and `val.bin`. The model itself has fixed positional embeddings of size `T`, so it cannot process sequences longer than this maximum.

Memory Costs of Longer Sequences

Transformers are powerful but expensive. The attention mechanism compares every token to every other token in the sequence. This means memory and compute scale with the square of sequence length:

Sequence Length (T)	Relative Attention Cost
256	1×
512	4×
1024	16×
2048	64×

So doubling `T` doesn't just double the cost-it multiplies it by four. That's why training at long context lengths requires a lot of GPU memory.

Trade-offs

- Shorter sequences: Faster, less memory, but limited context. Good for quick experiments or tiny datasets like Tiny Shakespeare.
- Longer sequences: More memory, slower, but the model can understand larger spans of text. Required for large-scale GPT-2 training.

You can think of sequence length as a dial: turning it up increases the model's ability to "remember," but it also makes training much heavier.

Practical Choices in *llm.c*

- Tiny Shakespeare example: often trained with `T = 64` or `128` for speed.
- GPT-2 small (124M): typically uses `T = 1024`, the same as the original paper.
- If your GPU has limited memory, you might need to shrink `T` and/or batch size `B`.

Why It Matters

Choosing sequence length is about balancing learning power against hardware limits. A too-small sequence length can prevent the model from capturing long-term dependencies. A too-large one can make training impossible on your hardware. Every run of *llm.c* is a negotiation between what you'd like the model to see and what your system can handle.

Try It Yourself

1. Short vs long: Train Tiny Shakespeare with `T = 64` and then `T = 256`. Compare both the speed and the coherence of generated text.
2. Memory test: Increase `T` step by step until you hit an out-of-memory (OOM) error. Note the maximum your GPU can handle.
3. Batch trade-off: Try reducing batch size `B` while increasing `T`. Can you keep GPU memory stable while giving the model more context?
4. Validation impact: Run with different `T` values and watch how validation loss behaves. Does longer context always help?
5. Inspect embeddings: Print out the shape of the positional embeddings. Notice how they are always tied to `T`.

The Takeaway

Sequence length (`T`) controls how much context the model sees. It directly determines the size of the positional embeddings, the structure of batches, and the memory required for attention. In *llm.c*, adjusting `T` is one of the fastest ways to explore the trade-offs between speed, memory, and model capability.

17. Reproducibility and Seeding Across Runs

When training machine learning models, it's common to notice that two runs-using the same code and the same dataset-don't produce exactly the same results. This happens because many parts of training involve randomness. In *llm.c*, reproducibility is controlled by random seeds. A seed is a starting point for a random number generator. If you always start from the same seed, the sequence of "random" numbers will be identical, and so will the training run.

Where Randomness Appears

Even in a small project like *llm.c*, randomness shows up in several places:

Component	Random Role
Weight initialization	The model's parameters (like attention matrices) are set randomly at the start.
Optimizer states	Some optimizers use random noise (though AdamW is mostly deterministic).
Sampling outputs	When generating text, randomness decides which token to pick if probabilities are close.

Component	Random Role
Parallelism	On GPU, threads may execute in slightly different orders, sometimes introducing small nondeterminism.

Without a fixed seed, every training run can drift apart, even if all settings look the same.

How *llm.c* Handles Seeds

The repository provides a small random utilities module: `llmc/rand.h`. Inside you'll find functions such as:

```
void manual_seed(uint64_t seed);
float normal_(float mean, float std);
```

- `manual_seed` sets the seed for the internal random number generator, ensuring reproducibility.
- `normal_` is used for initializing weights with Gaussian noise, similar to PyTorch's `torch.nn.init.normal_`.

When you call `manual_seed(1337);`, the model weights will be initialized the same way every time.

Why Seeds Don't Guarantee Perfect Reproducibility

Even with a fixed seed, you may still see small differences:

- GPU kernels sometimes use parallel algorithms that are not bitwise deterministic.
- Floating-point math can produce slightly different rounding on different hardware.
- Multi-GPU runs (via NCCL/MPI) may introduce nondeterministic reduce operations.

These differences are usually tiny-validation loss might vary by 0.001-but they exist. For most educational purposes, *llm.c* seeds are enough to make experiments repeatable.

Typical Defaults

In many examples, you'll see:

```
manual_seed(1337);
```

This "magic number" 1337 is just a convention. You can change it to any integer. Using the same seed across runs guarantees the same starting weights, which helps when comparing hyperparameters.

Why It Matters

Reproducibility is crucial in machine learning because it lets you:

- Debug effectively: If a bug appears, you want it to appear consistently.
- Compare settings: You can test learning rates or batch sizes fairly by keeping everything else the same.
- Share results: Other people can run your exact setup and see the same outcomes.

Without seeds, it becomes hard to tell whether a difference came from your hyperparameter change or just random luck.

Try It Yourself

1. Run twice with same seed: Train GPT-2 with `manual_seed(1337)` set. Do you get identical training loss curves?
2. Change the seed: Try `manual_seed(42)` and compare the loss curve. How similar are they? Do they converge to about the same final validation loss?
3. Remove seeding: Comment out the seed line and run again. Notice how runs diverge.
4. Sampling experiment: With a fixed seed, generate text multiple times. Then change the seed and generate again. See how outputs change.
5. Multi-GPU test: If you have more than one GPU, run the same seed across devices. Do results stay exactly the same or only approximately?

The Takeaway

Reproducibility in *llm.c* comes from setting seeds for random number generators. While floating-point quirks mean you can't always get perfect bit-for-bit matches, seeds let you control the biggest source of randomness: weight initialization and sampling. With seeding, you can debug, compare, and share results confidently.

18. Error Surfaces from Bad Data (Bounds, Asserts)

When training a model in *llm.c*, everything depends on the quality and correctness of the data you feed in. If the dataset or batches contain mistakes, the training process can go off track quickly-sometimes by crashing outright, other times by producing strange loss values like `NaN`. To guard against this, the code uses bounds checks and asserts that catch problems early.

What Can Go Wrong with Data

There are several common data issues:

Problem	What Happens
Token ID out of range	The model expects IDs between 0 and `vocab_size-1`. A wrong ID can cause array indexing errors.
Empty or short dataset	The DataLoader may run out of tokens before filling a batch.
Mismatched tokenizer	If you build a dataset with a different tokenizer, IDs may not correspond to the GPT-2 tokenizer in `gpt2_tokenizer.bin`. This produces nonsense outputs.
Corrupt `.bin` files	If files are incomplete or written incorrectly, the DataLoader might read garbage values.

These errors show up as segfaults, invalid memory access, or exploding losses during training.

How *llm.c* Defends Against Bad Data

The repository makes heavy use of asserts-simple checks that stop the program immediately if something unexpected happens. For example, in `llmc/utils.h`, functions like `freadCheck` and `mallocCheck` ensure that file reads and memory allocations succeed. If not, they print an error message and abort instead of silently failing.

Inside the DataLoader, token IDs are often validated to make sure they fall inside the expected vocabulary range. If you try to access an invalid index in the embedding table, the program will crash quickly, which is better than continuing with corrupted values.

Example: Vocab Range Check

During training, every input token is used to look up a row in the embedding matrix. If a token ID is too large, you'd access memory outside the matrix. This is why checking `0 <= id < vocab_size` is essential. In C, asserts provide this safety net.

```
assert(id >= 0 && id < vocab_size);
```

This kind of check may look simple, but it saves hours of debugging mysterious crashes.

Error Surfaces in Loss

Even if your program doesn't crash, bad data can create "error surfaces" in the loss function:

- NaNs: Appear when invalid values propagate through softmax, layernorm, or division operations.
- Flat loss: If the dataset is empty or repetitive, the model never improves.
- Mismatch behavior: Training loss decreases but validation loss stays high if training and validation sets use inconsistent tokenization.

These are signs that something is wrong with the dataset or preprocessing.

Why It Matters

C is a low-level language with very little safety by default. One out-of-range index can corrupt memory and cause unpredictable bugs. By aggressively checking assumptions (file sizes, vocab bounds, token IDs), *llm.c* turns hard-to-find errors into immediate, clear failures. For learners, this makes it much easier to understand what went wrong.

Try It Yourself

1. Corrupt a dataset: Open `train.bin` and delete a few bytes. Run training and see what error appears. Notice how quickly asserts catch it.
2. Force a bad ID: Modify the DataLoader to add `+100000` to a token. Does the model crash with an assertion?
3. Skip asserts: Temporarily disable checks and rerun. Compare how much harder it is to figure out what went wrong.
4. Validation mismatch: Tokenize a file with a different tokenizer and save it as `val.bin`. Watch how the validation loss behaves compared to training loss.
5. Print debug info: Add logging to display the first 20 tokens of each batch. Can you spot bad data before it crashes?

The Takeaway

Bad data can silently sabotage training, but *llm.c* uses asserts and bounds checks to make errors loud and clear. This design choice helps learners focus on the real logic of transformers instead of chasing hidden bugs caused by corrupted or mismatched datasets. In machine learning, good data hygiene and strict validation are as important as the model itself.

19. Tokenization Edge Cases (UNKs, EOS, BOS)

Tokenization looks simple at first: take text, split it into tokens, and assign each token an ID. But in practice, there are always tricky situations. *llm.c* inherits the quirks of the GPT-2 tokenizer, which is byte-level BPE (Byte Pair Encoding). This design mostly avoids "unknown" tokens, but it still has details you need to understand when preparing datasets or interpreting outputs.

No True "UNK" in GPT-2

Some tokenizers, like those used in earlier NLP systems, include a special `UNK` (unknown) token for words that aren't in the vocabulary. GPT-2 avoids this problem by working at the byte level:

- Every possible byte (0–255) is in the base vocabulary.
- If the tokenizer doesn't know how to split a character or word, it just falls back to raw bytes.

That means you will never see an `UNK` token in *llm.c*. Any input text is always representable. This is one of the main reasons GPT-2's tokenizer is so robust.

Special Tokens: EOS and BOS

Even though GPT-2 doesn't use `UNK`, it does use other special tokens:

Token	ID	Purpose
EOS (End of Sequence)	50256	Marks the end of a text segment. Used during training and sampling.
BOS (Beginning of Sequence)	Not explicit in GPT-2	GPT-2 doesn't use a fixed BOS token. Instead, the model assumes generation starts at position 0.

In *llm.c*, you'll often see `EOS` at the end of training sequences or when sampling text. If you generate text and see strange endings, it's usually because the model predicted `EOS`.

Whitespace Quirks

The tokenizer also handles whitespace in a slightly unusual way. For example, the word "hello" and the word " hello" (with a leading space) map to different tokens. This is why generated text sometimes starts with a space-it's part of the token definition.

Example:

- `"hello"` → token ID 31373
- `" hello"` → token ID 15496

This is normal behavior for GPT-2. It helps the model capture spacing and punctuation consistently.

Unicode and Rare Characters

Because it's byte-level, GPT-2 can encode emojis, accented characters, or even binary junk data. But the BPE merges are optimized for English, so rare characters often get split into multiple byte tokens. That means sequences with lots of rare symbols (like Chinese or emojis) will use more tokens than plain English text.

Why It Matters

Edge cases in tokenization affect both dataset preparation and model outputs. If you see weird spacing or early `EOS` tokens, it's not a bug-it's just how the tokenizer works. Understanding these quirks helps you debug outputs and prepare datasets without surprises.

Try It Yourself

1. EOS inspection: Open `val.bin` with a hex viewer and look for token ID `50256`. These mark the ends of text segments.
2. Whitespace check: Use the tokenizer to encode `"hello"` and `" hello"`. Compare the token IDs.
3. Emoji test: Encode a string with emojis (e.g., `" "`) and see how many tokens it becomes.
4. Rare character dataset: Create a small `.txt` file with accented characters and tokenize it. How many bytes does each character consume?
5. Sampling experiment: Generate text until you see the EOS token appear. Notice how the model "knows" to stop.

The Takeaway

Tokenization in GPT-2 is robust, but it has quirks. There are no unknown tokens thanks to byte-level encoding, but whitespace and special tokens like `EOS` play important roles. By experimenting with these edge cases, you'll develop an intuition for how raw text is mapped into the numbers that drive training and generation in *llm.c*.

20. Data Hygiene and Logging

When training with *llm.c*, having clean data is just as important as having the right model code. If the dataset contains errors, duplicates, or formatting issues, the model may waste capacity memorizing noise instead of learning useful patterns. This is where data hygiene comes in-making sure your training and validation sets are prepared properly. Alongside this, logging ensures you can monitor what's happening during training and catch problems early.

What Data Hygiene Means

Data hygiene is about making sure your dataset is both valid and useful. For language models, this includes:

Check	Why It Matters
Correct tokenization	Must match the tokenizer (`gpt2_tokenizer.bin`), otherwise IDs won't line up.
No corrupt files	Binary `.bin` files must be complete; partial writes cause crashes.
Balanced splits	Training and validation sets should come from the same distribution.
Reasonable size	Too small → overfitting. Too large → slow or infeasible.
Deduplication	Repeated passages (e.g., web scrapes) make models memorize instead of generalize.

The scripts in `dev/data/` handle basic hygiene by tokenizing consistently and splitting into train/val sets. But if you bring your own dataset, you are responsible for cleaning it first.

Logging During Training

Once training starts, logging becomes your window into what's happening. *llm.c* uses a minimal logging system (`llmc/logger.h`) to print progress to the console. Typical logs include:

```
step 0: train loss 5.19, val loss 5.32
step 100: train loss 4.87, val loss 5.01
step 200: train loss 4.62, val loss 4.88
```

These numbers let you track:

- Training loss: Is the model fitting the data?
- Validation loss: Is it generalizing, or overfitting?
- Step timing: How long each batch takes, useful for profiling.

Even in such a small project, this logging loop gives you most of what you need to debug runs.

Why Hygiene and Logging Go Together

Bad data often reveals itself in the logs. For example:

- If validation loss is much higher than training loss, your validation set may be mismatched.
- If loss suddenly becomes `NaN`, your dataset might contain corrupt tokens.
- If loss plateaus at a high value, you may have too little data or poor preprocessing.

By keeping your data clean and watching logs closely, you can detect these issues early instead of wasting hours of compute.

Try It Yourself

1. Dirty dataset test: Take a `.txt` file, add random symbols or binary junk, and prepare a `.bin` dataset. What happens to training loss?
2. Duplicate passages: Copy the same paragraph 100 times into a training file. Does validation loss improve, or does the model just memorize?
3. Log frequency: Modify the code to log every step instead of every N steps. How noisy are the results?
4. Custom logger: Extend the logger to also print gradient norms or learning rate values. Does this help you understand training dynamics better?
5. Compare splits: Build two datasets with different train/val splits. Which one gives more stable validation losses?

The Takeaway

Data hygiene ensures the model learns from clean, consistent input, while logging ensures you can see whether learning is actually happening. Together, they form the foundation of reliable experiments in *llm.c*. If you clean your data carefully and pay attention to the logs, you'll catch most problems before they become serious.

Chapter 3. Model Definition and Weights

21. GPT-2 Config: Vocab, Layers, Heads, Channels

Every GPT-2 model, no matter how large or small, is defined by a handful of configuration numbers. These numbers decide how big the model is, how much memory it needs, and how powerful it can become. In *llm.c*, these settings are stored in a simple config struct and printed at the start of training. They describe the "blueprint" of the transformer.

The Core Parameters

Here are the most important values you'll see in the logs:

Parameter	Meaning	Example (GPT-2 Small)
`vocab_size`	Number of distinct tokens (from tokenizer).	50,257
`padded_vocab_size`	Vocab size rounded up to nearest multiple (for GPU efficiency).	50,304
`max_seq_len`	Longest sequence of tokens the model can handle.	1,024
`num_layers`	Number of transformer blocks stacked on top of each other.	12
`num_heads`	Number of attention heads per block.	12
`channels`	Width of hidden states (embedding dimension).	768
`num_parameters`	Total trainable weights in the model.	~124M

Together, these values define both the structure and the capacity of the model.

What They Control

- Vocabulary size connects the model to the tokenizer. Every input token ID must be less than `vocab_size`. The padded version makes GPU matrix multiplications easier.
- Max sequence length fixes the size of the positional embeddings. If you set this to 1024, the model can't read beyond 1024 tokens in one pass.
- Layers control model depth. Each layer contains an attention block and an MLP. More layers = more representational power.
- Heads divide attention into parallel "subspaces." With 12 heads, the model can track different types of relationships in the text at the same time.
- Channels set the dimensionality of embeddings and hidden vectors. Larger channels mean more expressive representations but also more computation.
- Parameters are the sum of it all. This number tells you how heavy the model is to train and how much memory it will consume.

Configs Across GPT-2 Sizes

The original GPT-2 models come in several sizes:

Model	Layers	Heads	Channels	Parameters
GPT-2 Small	12	12	768	124M
GPT-2 Medium	24	16	1024	355M

Model	Layers	Heads	Channels	Parameters
GPT-2 Large	36	20	1280	774M
GPT-2 XL	48	25	1600	1.6B

llm.c can scale between these by just changing a few numbers in the config struct.

Where Config Appears in the Code

In `train_gpt2.c` and `train_gpt2.cu`, you'll see something like:

```
GPT2Config config = {
    .vocab_size = 50257,
    .max_seq_len = 1024,
    .num_layers = 12,
    .num_heads = 12,
    .channels = 768,
};
```

Later, the model is initialized using this struct, and the log prints all the derived information (like `num_parameters`).

Why It Matters

The config is the contract between your dataset and your model.

- If `vocab_size` doesn't match your tokenizer, you'll get crashes.
- If `max_seq_len` is too small, you'll lose context.
- If `num_layers` or `channels` are too large for your GPU, you'll run out of memory.

By tweaking the config, you decide whether you want a tiny model for learning or a massive one closer to GPT-2 XL.

Try It Yourself

1. Print config: Run the trainer and note the printed values. Compare them with the GPT-2 sizes in the table.
2. Shrink the model: Change `num_layers = 4`, `num_heads = 4`, and `channels = 256`. Train on Tiny Shakespeare and see how fast it runs.

3. Increase sequence length: Try setting `max_seq_len = 2048`. Does your GPU still handle it, or do you get out-of-memory errors?
4. Parameter count check: Compute how many parameters your custom config has. Compare it to the reported `num_parameters`.
5. Tokenizer mismatch test: Intentionally set `vocab_size = 30000` and watch what error appears when loading the tokenizer.

The Takeaway

The GPT-2 config struct in *llm.c* is small but powerful. It defines everything about the model's architecture: vocabulary, sequence length, depth, width, and total parameters. By adjusting just a few integers, you can scale from a toy model that runs on CPU to a billion-parameter giant (if your hardware allows it). Understanding these numbers is the first step to understanding how transformer capacity is controlled.

22. Parameter Tensors and Memory Layout

Once the GPT-2 configuration is set, the next big step is to allocate the parameters of the model. These are the trainable numbers-weights and biases-that define how the model processes input tokens. In *llm.c*, parameters are stored in flat arrays of floats rather than in deeply nested objects like in PyTorch. This choice makes the code easier to read and keeps memory access predictable.

What Are Parameters?

Every part of the transformer has its own trainable weights:

- Embedding tables: one for tokens and one for positions.
- Attention layers: query, key, value, and output projections.
- MLP layers: two linear layers plus their biases.
- LayerNorms: scale (`gamma`) and shift (`beta`) values.
- Final projection: maps hidden states back to vocab size for logits.

Together, these add up to hundreds of millions of numbers, even for GPT-2 Small.

Flat Memory Design in *llm.c*

Instead of allocating each parameter separately, *llm.c* stores all parameters in one contiguous block of memory. Each layer is given a slice of this big array.

This has two benefits:

1. Simplicity: You only need one malloc (or cudaMalloc) for all parameters.
2. Performance: Contiguous memory access is faster on both CPU and GPU.

To keep track of where each layer's weights live inside the block, the code uses offsets.

Example in Code

In `train_gpt2.c`, parameters are packed into a single array:

```
float* params = (float*)mallocCheck(config.num_parameters * sizeof(float));
```

Later, helper functions compute pointers into this array for each sub-module. For example, the token embedding weights are just the first slice:

```
float* token_embedding_table = params;
```

Then the program moves forward, assigning chunks to positional embeddings, attention weights, and so on.

Shapes of the Tensors

Even though parameters are stored in 1D memory, they conceptually form 2D or 3D tensors. For example:

Parameter	Shape	Purpose
Token embeddings	`[vocab_size, channels]`	Maps token IDs to vectors.
Positional embeddings	`[max_seq_len, channels]`	Adds position info.
Attention weights (Q, K, V, O)	`[channels, channels]`	Project hidden states.
MLP layers	`[channels, 4×channels]` and `[4×channels, channels]`	Expand and contract hidden states.
LayerNorm scale/shift	`[channels]`	Normalize and rescale features.

When you look at the code, remember: these shapes are "virtual." They're just views into slices of the big 1D array.

Why This Layout Works Well

PyTorch or TensorFlow manage parameter tensors with lots of abstractions. *llm.c* strips this away: you see the raw memory, the exact number of parameters, and the order they're laid out in. This makes it clear how large the model really is and why it uses so much RAM or VRAM.

It also means you can easily save and load checkpoints by writing or reading the flat array directly to disk. No need for complicated serialization formats.

Why It Matters

Understanding parameter layout helps you:

- See how the model's size explodes as you increase layers, heads, or channels.
- Debug memory issues by checking how big each slice is.
- Realize how much of training is just linear algebra on big arrays of floats.

This perspective is powerful because it demystifies deep learning: at its core, GPT-2 is just multiplying slices of one giant float array again and again.

Try It Yourself

1. Print parameter count: Add a line in the code to print `config.num_parameters`. Compare it with the table for GPT-2 Small/Medium/Large.
2. Inspect a slice: Print the first 10 numbers of the embedding table. They'll look random (from initialization).
3. Change precision: Modify the code to allocate `half` (FP16) instead of `float`. How much memory do you save?
4. Checkpoint peek: Save a checkpoint, then open it in a hex viewer. It's just raw floats-proof that parameters are stored flat.
5. Parameter scaling: Double the number of layers and see how `num_parameters` changes. Can you predict the increase?

The Takeaway

In *llm.c*, parameters are not hidden inside classes or objects. They live in one flat block of memory, sliced up by convention into embeddings, attention matrices, MLP weights, and norms. This design makes the relationship between model architecture and memory crystal clear-and reminds you that even a billion-parameter transformer is "just" a giant array of numbers.

23. Embedding Tables: Token + Positional

Before a transformer can reason about text, it first needs to turn tokens into vectors. In *llm.c*, this job is handled by the embedding tables: one for tokens, one for positions. These tables are the very first layer of GPT-2, and they transform plain integer IDs into continuous values that the neural network can process.

Token Embedding Table

When you feed in a batch of token IDs, the model looks up their corresponding vectors in the token embedding table.

- Shape: `[vocab_size, channels]`
  - `vocab_size` `50,257` (for GPT-2)
  - `channels = hidden size` (768 for GPT-2 Small)
- Each row corresponds to one token in the vocabulary.
- Each row is a dense vector of size `channels`.

So if your input batch has size `(B, T)`, looking up embeddings gives you a tensor of shape `(B, T, channels)`.

In the code, this is implemented as an array slice from the flat parameter block:

```
float* token_embedding_table = params;  // first slice of parameters
```

At runtime, token IDs index directly into this table.

Positional Embedding Table

Transformers don't inherently know about word order. That's what positional embeddings are for.

- Shape: `[max_seq_len, channels]`
  - `max_seq_len = 1024` in GPT-2 Small
  - Same channel dimension as token embeddings
- Each position (0, 1, 2, ..., 1023) has its own vector.

During training, when the model sees token `i` at position `j`, it takes the token embedding vector and adds the positional embedding vector for `j`. This gives the model both word identity and word position.

In *llm.c*, positional embeddings immediately follow the token embeddings in the flat parameter array.

Adding Them Together

The embedding layer's forward pass is simple:

```
embedding_out[token, pos] = token_embedding[token] + positional_embedding[pos]
```

This results in a `(B, T, channels)` tensor that becomes the input to the first transformer block.

Why This Matters

Embeddings are the bridge between discrete tokens and continuous math. Without them, the model couldn't use linear algebra to learn patterns. By adding positional embeddings, GPT-2 knows the difference between:

- "dog bites man" → `dog` comes first, `man` comes last
- "man bites dog" → same tokens, but swapped positions change the meaning

This small step is essential: order and identity must both be captured before attention can begin.

Try It Yourself

1. Inspect shapes: Print the sizes of the token and positional embedding tables during initialization. Confirm they match `[vocab_size, channels]` and `[max_seq_len, channels]`.
2. Look at first rows: Print the first 5 vectors of the token embedding table. They should look like small random floats from initialization.
3. Change max_seq_len: Double `max_seq_len` in the config. How does this change the size of the positional table? Does training still work?
4. Overwrite embeddings: Try setting the token embedding table to all zeros. What happens to training loss?
5. Sampling experiment: After training a few steps, decode outputs without adding positional embeddings. Do the results become nonsensical or repetitive?

The Takeaway

The embedding tables are the foundation of GPT-2. Token embeddings give meaning to symbols, while positional embeddings give structure to sequences. In *llm.c*, they are just two slices of the flat parameter array, added together at the very start of the forward pass-but without them, the transformer would be blind to both words and order.

24. Attention Stack: QKV Projections and Geometry

After embeddings, the real magic of transformers begins: the attention mechanism. In GPT-2, every transformer block contains an attention stack. This is where the model learns how each token relates to others in the sequence-whether it's paying attention to the previous word, the beginning of a sentence, or even punctuation marks far away.

What Attention Does

Attention lets the model answer the question:

> "Given the current word, which other words in the context should I care about, and how much?"

Instead of treating words independently, the model uses attention to build connections across the sequence.

The Q, K, V Projections

Each attention block starts with three linear projections:

Name	Shape	Purpose
Q (Query)	`[channels, channels]`	Represents what each token is *asking* about.
K (Key)	`[channels, channels]`	Represents how each token can be *recognized*.
V (Value)	`[channels, channels]`	Represents the actual *information* to pass along.

Here's the flow:

1. Each input vector (from embeddings or previous block) is multiplied by these three matrices to produce Q, K, and V vectors.

2. Attention scores are computed by comparing Qs with Ks.
3. These scores are used to weight the Vs, mixing information from other tokens into the current one.

Geometry of Attention

- Q and K define a similarity score: how well does this token match another one?
- V carries the actual features (like meaning, grammar cues).
- The result is a weighted sum: tokens borrow information from others based on attention scores.

In equations:

```
scores = Q × K^T / sqrt(d_k)
weights = softmax(scores + mask)
output  = weights × V
```

The division by `sqrt(d_k)` normalizes scores so they don't blow up as dimensions grow.

Multi-Head Attention

GPT-2 doesn't use just one attention projection-it uses many in parallel, called heads. Each head learns to focus on different types of relationships:

- One head might track subject–verb agreement.
- Another might watch punctuation and quotes.
- Another might connect pronouns to their referents.

For GPT-2 Small:

- 12 heads per layer
- Each head works on a reduced dimension (`channels / num_heads`)
- Outputs are concatenated and projected back to `channels`

This setup is what gives transformers their flexibility.

Implementation in *llm.c*

In the parameter array, each transformer block has slices for Q, K, V, and output projection (O). During forward pass:

1. Multiply input by Q, K, V matrices.
2. Reshape into heads.
3. Compute attention scores (masked to prevent looking forward).
4. Apply softmax.
5. Multiply by V to get weighted values.
6. Concatenate heads and apply the O projection.

All of this is done with plain matrix multiplications and softmax calls-no magic beyond linear algebra.

Why It Matters

Attention is the beating heart of GPT-2. It's how the model captures dependencies across text, from short-term grammar to long-range coherence. Without QKV, embeddings would stay isolated, and the model could never build context-aware representations.

Try It Yourself

1. Print shapes: Log the shapes of Q, K, V matrices in one layer. Confirm they match `[channels, channels]`.
2. Visualize scores: After a forward pass, print the attention weights for one head. Do they concentrate on recent tokens or spread across the sequence?
3. Reduce heads: Change `num_heads` from 12 to 4. What happens to validation loss?
4. Break symmetry: Initialize all Q, K, V matrices with zeros. Does training loss decrease at all?
5. Mask experiment: Disable the causal mask (allow looking ahead). Does the model "cheat" by predicting future tokens perfectly?

The Takeaway

The attention stack is where tokens stop being isolated and start talking to each other. Q, K, and V projections turn context into weighted relationships, and multi-head attention lets the model juggle many types of dependencies at once. In *llm.c*, this is implemented with straightforward linear algebra, making the most powerful idea in modern NLP visible and accessible.

25. MLP Block: Linear Layers + Activation

After attention mixes information across tokens, GPT-2 applies a second transformation inside each block: the MLP (Multi-Layer Perceptron). This part doesn't look at other tokens-it processes each position independently. But it's just as important because it gives the model extra capacity to transform and refine the hidden features before passing them to the next layer.

What the MLP Looks Like

Every transformer block contains an MLP with two linear layers and a nonlinear activation in between:

```
hidden = Linear1(x)
hidden = GELU(hidden)
out    = Linear2(hidden)
```

This structure expands the feature dimension and then compresses it back down, which lets the network learn richer representations.

Shapes of the Layers

If the hidden size (channels) is `d_model`, the MLP works as follows:

Step	Shape	Purpose
Input	`[B, T, d_model]`	Output of attention for each token.
Linear1	`[d_model, 4 × d_model]`	Expands features 4× wider.
GELU	elementwise	Introduces nonlinearity.
Linear2	`[4 × d_model, d_model]`	Projects back to original size.
Output	`[B, T, d_model]`	Same shape as input, ready for residual add.

For GPT-2 Small (`d_model = 768`), Linear1 expands to 3072 channels, then Linear2 reduces back to 768.

Activation: GELU

The activation function in GPT-2 is GELU (Gaussian Error Linear Unit). It's smoother than ReLU, giving the model a more nuanced way of handling values around zero. In code, GELU looks like:

```
float gelu(float x) {
    return 0.5f * x * (1.0f + tanhf(0.79788456f * (x + 0.044715f * x * x * x)));
}
```

This formula may look complicated, but the idea is simple: it smoothly squashes negative values toward zero and keeps positive values flowing through.

Why Expand and Shrink?

The expansion to `4 × d_model` may seem wasteful, but it's deliberate:

- Expanding gives the model more capacity to represent patterns at each token.
- Shrinking keeps the overall parameter count manageable.
- Together, they act like a bottleneck layer that forces the model to transform information more effectively.

This "expand → activate → shrink" design is one of the main reasons transformers scale so well.

Implementation in *llm.c*

Just like attention, the MLP parameters live in the flat array of floats. Each block stores two weight matrices and two bias vectors. During forward pass:

1. Multiply input by `Linear1` weights, add bias.
2. Apply GELU elementwise.
3. Multiply by `Linear2` weights, add bias.
4. Pass result through residual connection.

Because each position is processed independently, the MLP is easy to parallelize across tokens.

Why It Matters

The MLP is the nonlinear refiner of transformer blocks. Attention spreads information, but MLPs transform it in-place, giving the model more expressive power. Without the MLP, the network would be mostly linear, limiting its ability to capture complex patterns in text.

Try It Yourself

1. Print shapes: Log the dimensions of Linear1 and Linear2 weights in one block. Do they match `[768, 3072]` and `[3072, 768]` for GPT-2 Small?
2. Swap activation: Replace GELU with ReLU in the code. Does training still work? How does validation loss compare?
3. Reduce expansion: Change expansion from 4× to 2× (`[768, 1536]`). What effect does this have on parameter count and performance?
4. Zero out MLP: Set MLP weights to zero. Does the model still learn anything, or does performance collapse?
5. Compare speed: Measure training step time with and without the MLP enabled. How much slower is it?

The Takeaway

The MLP block in GPT-2 is a simple two-layer network with GELU activation, applied independently to each token. It expands, activates, and compresses features, giving the model nonlinear power to reshape hidden states. In *llm.c*, it's implemented with basic matrix multiplications and a smooth GELU function, proving that even small building blocks can have a big impact on the model's ability to learn language.

26. LayerNorm: Theory and Implementation (`doc/layernorm`)

Deep neural networks often suffer from unstable training if activations drift too high or too low. To stabilize this, GPT-2 uses Layer Normalization (LayerNorm) inside every transformer block. In *llm.c*, LayerNorm is implemented directly in C, and there's even a detailed explanation in the repo's `doc/layernorm` file to help learners understand how it works.

The Idea of Normalization

When you pass vectors through many layers, their values can become unbalanced-some features dominate while others shrink. Normalization fixes this by:

1. Centering: subtracting the mean of the vector.
2. Scaling: dividing by the standard deviation.

This makes every feature vector have mean 0 and variance 1, improving stability.

Why "Layer" Norm?

There are different kinds of normalization (BatchNorm, InstanceNorm, etc.). LayerNorm is special because:

- It normalizes across the features of a single token (the "layer"), not across the batch.
- This makes it independent of batch size, which is important for NLP where batch sizes can vary.

So if a hidden vector has 768 channels, LayerNorm computes the mean and variance over those 768 numbers for each token.

Trainable Parameters

LayerNorm isn't just normalization-it also has two trainable vectors:

- (gamma): scales each feature after normalization.
- (beta): shifts each feature after normalization.

These allow the network to "undo" normalization when necessary, giving it flexibility.

Formula

For each input vector `x` of size `d`:

```
mean = (1/d) * Σ x_i
var  = (1/d) * Σ (x_i - mean)^2
x_norm = (x - mean) / sqrt(var + eps)
y =   * x_norm +
```

Where `eps` is a tiny constant (like `1e-5`) to avoid dividing by zero.

Implementation in *llm.c*

In the code, LayerNorm is implemented as a simple function that loops over features, computes mean and variance, and applies the formula above. It's not hidden inside a framework-it's right there in C, so you can step through it line by line.

For example, the forward pass looks like this (simplified):

```c
void layernorm_forward(float* out, float* inp, float* weight, float* bias, int N) {
    float mean = 0.0f, var = 0.0f;
    for (int i = 0; i < N; i++) mean += inp[i];
    mean /= N;
    for (int i = 0; i < N; i++) var += (inp[i] - mean) * (inp[i] - mean);
    var /= N;
    float inv_std = 1.0f / sqrtf(var + 1e-5f);
    for (int i = 0; i < N; i++) {
        out[i] = (inp[i] - mean) * inv_std * weight[i] + bias[i];
    }
}
```

This is the kind of clear, low-level implementation that makes *llm.c* educational.

Where It Fits in GPT-2

Each transformer block contains two LayerNorms:

- One before attention.
- One before the MLP.

GPT-2 uses Pre-LN architecture: inputs are normalized before each sublayer. This makes training more stable and gradients flow better.

Why It Matters

LayerNorm may look like a small detail, but without it, GPT-2 would fail to train reliably. It smooths out the flow of activations so attention and MLP layers can do their job. In practice, this is one of the critical "glue" components that makes deep transformers trainable at scale.

Try It Yourself

1. Print statistics: After applying LayerNorm, print the mean and variance of the output. Do they stay close to 0 and 1?
2. Remove and : Force gamma to 1 and beta to 0. Does the model still train? Compare losses.
3. Disable normalization: Comment out LayerNorm and train. How unstable does training become?
4. Compare positions: Try switching to Post-LN (apply normalization after attention/MLP). Does this change convergence speed?
5. Vary epsilon: Change `1e-5` to `1e-2` or `1e-8`. How sensitive is training?

The Takeaway

LayerNorm is the quiet stabilizer of GPT-2. It makes sure each token's features stay balanced, while and keep flexibility. In *llm.c*, it's implemented directly with clear C code, letting you see exactly how normalization is calculated. It's a small but indispensable piece of the transformer puzzle.

27. Residual Connections: Keeping the Signal Flowing

Transformers like GPT-2 don't just stack layers on top of each other blindly. They use residual connections-a trick that allows the input of a layer to be added back to its output. This simple addition helps signals flow through the network without vanishing or exploding, and it makes training deep models possible.

The Basic Idea

Imagine you have a function `F(x)` representing some transformation (like attention or an MLP). Instead of just computing:

```
y = F(x)
```

the transformer does:

```
y = F(x) + x
```

This means the layer learns only the *difference* it needs to add to the input, instead of replacing it entirely.

Why This Helps

Residuals solve two big problems in deep networks:

1. Gradient flow: During backpropagation, gradients can get smaller and smaller as they pass through many layers. Adding the input back ensures gradients always have a path straight through.
2. Information preservation: Even if `F(x)` distorts the signal, the original `x` is still there. This prevents the model from "forgetting" important information.
3. Faster training: The network doesn't have to re-learn identity mappings-it can just pass them through the skip connection.

Implementation in *llm.c*

Residuals in *llm.c* are implemented as a straightforward elementwise addition:

```
void residual_forward(float* out, float* inp1, float* inp2, int N) {
    for (int i = 0; i < N; i++) {
        out[i] = inp1[i] + inp2[i];
    }
}
```

Here:

- `inp1` is the output of the layer (like attention).
- `inp2` is the original input.
- `out` is the combined result.

This is done for every token position and feature channel.

Where Residuals Are Used

In GPT-2, every transformer block has two residuals:

1. Attention residual: Adds the input of the attention layer to its output.
2. MLP residual: Adds the input of the MLP to its output.

So the data flowing through the network always carries both the new transformation and the original signal.

Why It Matters

Without residual connections, stacking 12–48 transformer blocks would be nearly impossible to train. Gradients would vanish, and the model would either stop learning or take forever to converge. Residuals let deep transformers scale smoothly.

They also add an intuitive interpretation: each block is like a "refinement step" rather than a full rewrite of the representation.

Try It Yourself

1. Remove residuals: Comment out the addition in the code. Does training collapse?
2. Scale residuals: Multiply the input by 0.5 before adding. Does this slow convergence?
3. Check loss curves: Compare training with and without residuals for the first 500 steps.
4. Inspect outputs: Print the norms of `inp1`, `inp2`, and `out`. Are the scales balanced?
5. Deeper models: Increase the number of layers from 12 to 24. Does the importance of residuals become more obvious?

The Takeaway

Residual connections are the "lifeline" of deep transformers. By simply adding inputs back into outputs, they make it possible to train very deep networks without losing gradients or information. In *llm.c*, the implementation is as simple as looping over arrays and adding them-but the effect is profound: residuals are what let GPT-2 go deep and still work.

28. Attention Masking: Enforcing Causality

One of the defining traits of GPT-2 is that it's a causal language model. That means it predicts the *next* token given all the tokens before it, but never cheats by looking ahead. To enforce this, GPT-2 applies an attention mask inside every attention layer.

Why a Mask Is Needed

Without a mask, attention is free to connect any token to any other, including future ones. For example:

- Input: "The cat sat on the"
- Target: "mat"

If the model could peek at "mat" while computing attention, the task would be trivial-it could just copy the next word. That would break the training objective.

The mask forces the model to only use tokens at or before the current position when making predictions.

How the Mask Works

When computing attention scores (`Q × K^T / sqrt(d_k)`), the result is a matrix of size `[T, T]` where each row corresponds to one token attending to all others.

The mask modifies this matrix:

- Allowed positions (past and present): keep scores as is.
- Disallowed positions (future): set scores to `-inf`.

After applying softmax, those `-inf` entries become zero probability, effectively blocking attention to the future.

Implementation in *llm.c*

The causal mask is applied during the attention forward pass. The code uses a loop to zero out invalid positions:

```
for (int t = 0; t < T; t++) {
    for (int u = t + 1; u < T; u++) {
        scores[t][u] = -1e9; // block future positions
    }
}
```

Here `T` is the sequence length. This ensures that token `t` can only attend to itself and earlier tokens.

Visualizing the Mask

Think of the mask as a triangular matrix:

	0	1	2	3
0				
1				
2				
3				

Each row shows which past tokens a given position can look at. Future positions remain blank.

Why It Matters

The mask is what makes GPT-2 a predictive model instead of a bidirectional encoder like BERT. Without it, the model could "cheat" and the training objective would no longer match how it's used at inference time (generating text step by step).

This small detail-just filling part of a matrix with `-inf`-is critical to making autoregressive text generation possible.

Try It Yourself

1. Disable the mask: Comment out the masking code. Watch validation loss drop unrealistically, then notice that text generation produces garbage.
2. Reverse the mask: Block the past and allow the future. Does the model still train? What does it predict?
3. Partial mask: Only allow attention to the previous 5 tokens (a sliding window). How does this affect learning long-range structure?
4. Print scores: Before and after masking, log a row of attention scores. Notice how future positions become huge negatives.
5. Visualize: Write a small script to plot the attention mask as a matrix. It should look strictly lower-triangular.

The Takeaway

Attention masking is a simple but essential trick. By filling future positions with `-inf` before softmax, GPT-2 ensures that each token can only attend to its past. In *llm.c*, this is implemented with just a couple of loops-but it's what turns a generic transformer into a true causal language model.

29. Output Head: From Hidden States to Vocabulary

After tokens pass through embeddings, attention, MLPs, LayerNorm, and residuals, we end up with hidden states for every position in the sequence. But GPT-2's final job is not to output vectors-it must predict the next token from the vocabulary. This is handled by the output head, the last stage of the model.

What the Output Head Does

The output head maps hidden states of shape `(B, T, channels)` into logits of shape `(B, T, vocab_size)`. Each logit represents the model's "raw score" for how likely a particular token is at the next step.

The pipeline looks like this:

`hidden states → Linear projection → Logits → Softmax → Probabilities`

- Logits: real numbers, one per token in the vocabulary.
- Softmax: converts logits into probabilities that sum to 1.
- Predicted token: the token with the highest probability (or sampled from the distribution).

Tied Weights with Embeddings

In GPT-2, the token embedding table and the output head share weights. This means the same matrix is used both for:

- Mapping tokens to vectors at the start (embedding lookup).
- Mapping vectors back to tokens at the end (output head).

Mathematically, this improves efficiency and helps align input and output representations.

In *llm.c*, this is done by simply pointing both embedding and output head to the same parameter slice.

```
// token embedding table
float* token_embedding_table = params;
// output head reuses the same memory
float* output_head = token_embedding_table;
```

When the model projects hidden states back to vocab space, it does a matrix multiply with this shared matrix.

Shapes in Action

For GPT-2 Small:

- Hidden states: `[B, T, 768]`
- Output projection (embedding transpose): `[768, 50257]`
- Logits: `[B, T, 50257]`

That's more than 50k scores per position, one for each token in the vocabulary.

Why Weight Tying Helps

1. Memory efficiency: You don't need a separate giant matrix for the output head.
2. Better learning: The same vectors that represent tokens going in also represent them going out, which reinforces consistency.
3. Simpler code: Just reuse the same parameter slice.

This trick is why GPT-2 can scale vocab sizes without blowing up parameter counts too much.

Why It Matters

The output head is where everything comes together. For each position, the model collapses its hidden representation into a distribution over possible next tokens. This is how GPT-2 generates text one step at a time. Without this step, you'd only have abstract hidden states-useful internally, but not something you can read.

Try It Yourself

1. Print logits: After a forward pass, print the logits for the last token. Do they look like random floats at initialization?
2. Check probability sum: Apply softmax to logits and verify the probabilities sum to 1.
3. Untie weights: Make the output head its own matrix instead of reusing embeddings. Does training still work? How does the parameter count change?
4. Top-k sampling: Modify sampling to keep only the top 5 logits before softmax. What kind of text does this produce?
5. Greedy vs random: Compare greedy decoding (argmax) vs random sampling from probabilities. Which one gives more interesting outputs?

The Takeaway

The output head is the final bridge between hidden vectors and actual words. By reusing the token embedding matrix, GPT-2 projects hidden states back into vocabulary space and produces logits for every possible token. In *llm.c*, this step is just another matrix multiplication-but it's the one that turns internal math into real text predictions.

30. Loss Function: Cross-Entropy over Vocabulary

Training GPT-2 means teaching it to predict the next token in a sequence. To measure how well it's doing, we need a loss function that compares the model's predicted probabilities with the true token IDs. In *llm.c*, this is done with the cross-entropy loss-a standard choice for classification tasks.

From Logits to Probabilities

After the output head, we have logits of shape `(B, T, vocab_size)`. These are raw scores. To turn them into probabilities:

```
probs = softmax(logits)
```

Softmax ensures two things:

- All values are between 0 and 1.
- They sum to 1 across the vocabulary.

So for each position, you get a probability distribution over all possible next tokens.

Cross-Entropy Definition

Cross-entropy compares the predicted distribution `p` with the true distribution `q`. For language modeling:

- `q` is a one-hot vector (all zeros, except 1 at the true token index).
- `p` is the probability vector from softmax.

The formula for one token:

```
loss = -log(p[true_token])
```

For a batch, you average across all tokens in all sequences.

Implementation in *llm.c*

In C, this boils down to:

```
float loss = 0.0f;
for (int b = 0; b < B; b++) {
    for (int t = 0; t < T; t++) {
        int target = targets[b*T + t];
        float logit_max = -1e9;
        for (int v = 0; v < vocab_size; v++) {
            if (logits[b*T*vocab_size + t*vocab_size + v] > logit_max) {
                logit_max = logits[b*T*vocab_size + t*vocab_size + v];
            }
        }
        // compute softmax denominator
        float sum = 0.0f;
        for (int v = 0; v < vocab_size; v++) {
            sum += expf(logits[b*T*vocab_size + t*vocab_size + v] - logit_max);
        }
        float logprob = logits[b*T*vocab_size + t*vocab_size + target] - logit_max - logf(sum
        loss += -logprob;
    }
}
loss /= (B * T);
```

This snippet shows how *llm.c* explicitly computes softmax and cross-entropy in loops. No black boxes-just raw math.

Intuition

- If the model assigns high probability to the correct token → loss is small.
- If the model assigns low probability to the correct token → loss is large.
- Minimizing loss means pushing probability mass toward the right answers.

Why Cross-Entropy Works for Language

Language modeling is essentially a huge multi-class classification problem: at each step, which word comes next? Cross-entropy is perfect here because it directly penalizes wrong predictions proportional to how confident the model was.

Why It Matters

The loss function is the only signal the model gets about how well it's doing. Everything else-parameter updates, weight tuning, learning dynamics-flows from this single number. A

well-implemented cross-entropy ensures training is stable and meaningful.

Try It Yourself

1. Check values: Print the loss after the first few steps. It should be close to `log(vocab_size)` (10.8 for 50k vocab) before training.
2. Overfit tiny batch: Train on just one sequence. Does the loss go near 0 after enough steps?
3. Change target: Replace the true token with a random one. Does the loss increase immediately?
4. Compare vocab sizes: Train with a smaller vocabulary (e.g., 100 tokens). Does initial loss drop to `log(100)  4.6`?
5. Inspect probabilities: For one token, print the top 5 predicted probabilities. Does the true token climb to the top as training progresses?

The Takeaway

The cross-entropy loss is the compass guiding GPT-2 during training. It turns raw logits into probabilities and measures how well the model predicts the correct next token. In *llm.c*, it's implemented with explicit loops and math, letting you see exactly how probabilities and losses are computed. Without this step, the model would have no way to learn from its mistakes.

Chapter 4. CPU Inference (Forward Only)

31. Forward Pass Walkthrough

When we talk about the *forward pass* in GPT-2, we mean the process of turning an input sentence (like "The cat sat on the") into predictions for the next word. In simple terms, it's how the model "thinks" before giving an answer. In `train_gpt2.c`, this happens inside the function `gpt2_forward`. Let's walk through it slowly, step by step, so you can see how numbers flow through the model and transform along the way.

1. From Words to Numbers

Computers don't understand words like *cat* or *sat*. They only understand numbers. Before the forward pass starts, text is already tokenized into IDs (integers). For example:

```
"The cat sat" → [464, 3290, 616]
```

Each number is a token ID. The model doesn't yet know what "464" means in plain English-it just knows it's a number that points into a table.

2. Embedding: Giving Words Meaning

The first real step in the forward pass is embedding lookup. Imagine we have a huge dictionary, but instead of definitions in English, each word ID points to a long vector of numbers (say, 768 numbers for GPT-2 small).

- Word embeddings (`wte`): Each token ID becomes a vector that captures the meaning of the word.
- Position embeddings (`wpe`): Each token also gets a vector for its position: first word, second word, third word, etc.

The model adds these two vectors together. This way, it knows not just what the word is, but also where it is in the sentence.

For example:

Token	Word	Word Embedding (shortened)	Position Embedding (shortened)	Combined Vector
464	"The"	[0.2, -0.5, 0.1, …]	[0.0, 0.1, -0.3, …]	[0.2, -0.4, -0.2, …]
3290	"cat"	[0.9, -0.2, 0.4, …]	[0.1, -0.1, -0.2, …]	[1.0, -0.3, 0.2, …]

Now every token is a vector with both meaning and position built in.

3. Transformer Layers: The Thinking Steps

GPT-2 has multiple identical layers stacked on top of each other. Each layer has two big parts: attention and MLP (feed-forward network).

Attention (looking around):

- Each word asks: "Which other words should I pay attention to right now?"
- For "sat," attention might focus heavily on "cat," because those words are related.
- The code computes *queries*, *keys*, and *values* for every word, then does dot-products, softmax, and weighted sums to mix information.

MLP (processing deeply):

- After attention, each token passes through a mini neural network (two matrix multiplications with a nonlinear GELU function in between).

- This helps each word refine its understanding, even if it doesn't directly attend to another word.

Both blocks have residual connections: the input is added back to the output, like keeping the original notes while adding new insights. This prevents information loss.

4. Normalization: Keeping Numbers Stable

At many points, the model normalizes vectors so they don't explode in size or shrink too small. This is called LayerNorm. It ensures training is stable, like making sure your cooking pot doesn't boil over or dry out.

5. The Final Prediction Layer

After all layers, the model produces a final vector for each position. Then:

- It multiplies those vectors by the embedding table again (but transposed).
- This gives logits: raw scores for each word in the vocabulary (about 50k options).

Example: for the last token "on the," the logits might be:

Word	Logit	Probability (after softmax)
"mat"	7.2	0.85
"dog"	5.1	0.10
"car"	3.0	0.05

The highest probability is "mat."

6. Softmax: Turning Scores into Probabilities

The logits are big numbers, but they don't mean much until we apply softmax. Softmax makes them into probabilities that sum to 1. This way, we can interpret them as chances: "There's an 85% chance the next word is *mat*."

7. Cross-Entropy Loss: Measuring Mistakes

If we're training, we also give the model the correct next word. The model checks how much probability it gave to that word. If it gave it high probability, the loss is low. If it gave it low probability, the loss is high.

- Correct: "mat" (probability 0.85 → loss 0.16, small).
- Wrong: "car" (probability 0.05 → loss 3.0, large).

This loss is averaged across all tokens, and it's the signal that tells the backward pass how to update the model.

8. Why It Matters

The forward pass is the part of GPT-2 that generates predictions. Without it, the model can't "think" or make sense of input. It's like the brain processing sensory input before deciding what to do. In `train_gpt2.c`, the forward pass is written with plain C loops, which makes the math crystal clear instead of hidden inside deep learning libraries.

9. Try It Yourself

1. Print embeddings: Modify the code to print the vector for the first token. See how it's just numbers, but those numbers are the "meaning" of the word.
2. Inspect probabilities: After the forward pass, print the softmax probabilities for one position. They should sum to 1.0.
3. Change sequence length: Increase `T` from 64 to 128. Notice how validation slows down, because attention compares all tokens with all others (T^2 scaling).
4. Baseline loss: Before training, measure the loss. It should be around `log(vocab_size)` (10.8 for GPT-2 small). That's the loss of random guessing.
5. Mask experiment: Temporarily remove the causal mask in attention. The model will "cheat" by looking ahead, and loss will drop unrealistically.

The Takeaway

The forward pass is like the thought process of GPT-2. Input words become vectors, vectors mix through attention and MLPs, everything gets normalized, and finally the model produces probabilities for the next word. It's a carefully choreographed dance of math operations, all coded in plain C loops in `train_gpt2.c`. Once you understand this flow, you can follow exactly how GPT-2 turns raw tokens into intelligent predictions.

32. Token and Positional Embedding Lookup

Before GPT-2 can do anything intelligent with text, it needs to turn raw numbers (token IDs) into vectors that capture meaning and context. This is the role of embeddings. In `train_gpt2.c`, this step is handled by the function `encoder_forward`. Let's take a closer look at how it works and why it matters.

Tokens Are Just Numbers

Suppose you type:

```
"The cat sat on the mat."
```

After tokenization, this sentence might look like:

```
[464, 3290, 616, 319, 262, 1142, 13]
```

These are just IDs. The model doesn't inherently know that `3290` means "cat." It only knows it needs to use these numbers to fetch vectors from a table.

The Embedding Tables

The model has two important tables stored in memory:

1. Word Token Embeddings (`wte`)
    - Size: `(V, C)` where `V` is vocab size (~50,000 for GPT-2 small) and `C` is channels (768).
    - Each row corresponds to a token ID.
    - Example: row 3290 might be `[0.12, -0.45, 0.88, …]`.
2. Positional Embeddings (`wpe`)
    - Size: `(maxT, C)` where `maxT` is the maximum sequence length (e.g. 1024).
    - Each row corresponds to a position index: 0 for the first token, 1 for the second, etc.
    - Example: position 2 might be `[0.07, 0.31, -0.22, …]`.

Both tables are filled with trainable values. At the start, they're random. As training progresses, the optimizer updates them so they encode useful patterns.

Adding Them Together

For each token at position `t`:

- Look up its word vector from `wte`.
- Look up its position vector from `wpe`.
- Add them elementwise.

This gives a final vector of size `C` that represents what the token is and where it is.

Example with simplified numbers:

Token ID	Word	Word Embedding	Position	Combined
464	The	[0.1, -0.2, 0.3]	[0.2, 0.0, -0.1]	[0.3, -0.2, 0.2]
3290	cat	[0.4, 0.5, -0.3]	[0.0, 0.1, 0.2]	[0.4, 0.6, -0.1]

Now the vector doesn't just mean "cat," it means "cat at position 1."

Why Position Matters

Without positions, the model would treat:

- "The cat sat"
- "Sat cat the"

as identical, because they use the same tokens. But word order is essential in language. By adding positional embeddings, GPT-2 knows the difference between "dog bites man" and "man bites dog."

Inside the Code

The embedding lookup is written explicitly with loops in C:

```
for (int b = 0; b < B; b++) {
    for (int t = 0; t < T; t++) {
        float* out_bt = out + b * T * C + t * C;
        int ix = inp[b * T + t];
        float* wte_ix = wte + ix * C;
        float* wpe_t = wpe + t * C;
        for (int i = 0; i < C; i++) {
            out_bt[i] = wte_ix[i] + wpe_t[i];
```

```
        }
    }
}
```

What's happening here:

- Loop over batches (`b`) and sequence positions (`t`).
- Find the token ID `ix`.
- Fetch its embedding `wte_ix`.
- Fetch its position embedding `wpe_t`.
- Add them element by element.

The result, `out_bt`, is the vector for this token at this position.

Analogy

Think of it like name tags at a conference:

- The word embedding is your name: "Alice."
- The position embedding is your table number: "Table 7."
- Together, they tell the conference staff who you are and where you are seated.

Without the table number, they might know who you are but not where to find you. Without your name, they just know there's someone at Table 7 but not who. Both are needed for proper context.

Why It Matters

Embeddings are the foundation of the whole model. If this step is wrong, everything else collapses. They transform meaningless IDs into rich vectors that carry semantic and positional information. This is the entry point where language starts becoming something a neural network can reason about.

Try It Yourself

1. Print a token embedding: Modify the code to print out `wte_ix` for a specific token ID like "cat." You'll see a vector of floats, the learned representation.
2. Print a position embedding: Do the same for `wpe_t` at position 0, 1, 2… Notice how positions have unique but consistent patterns.
3. Check the sum: Verify that `out_bt[i] = wte_ix[i] + wpe_t[i]`. This is literally how word and position are fused.

4. Shuffle words: Try feeding "cat sat" vs. "sat cat." The embeddings will differ because the position vectors change, even though the words are the same.
5. Observe growth during training: After some training steps, dump the embeddings again. You'll notice they stop being random and start showing structure.

The Takeaway

The embedding lookup is the very first step of the forward pass. It takes raw numbers and makes them meaningful by combining token identity and position. This prepares the input for the deeper transformer layers. Even though the C code looks simple-a few nested loops-it's doing the crucial work of giving words a mathematical shape the model can understand.

33. Attention: Matmuls, Masking, and Softmax on CPU

The attention mechanism is the heart of GPT-2. It's where each word in the input sequence decides which other words to look at when forming its representation. In `train_gpt2.c`, this happens inside the `attention_forward` function, which implements multi-head self-attention using plain C loops and matrix multiplications. Let's break it down carefully, step by step, so even an absolute beginner can follow the flow.

The Big Idea of Attention

Imagine you're reading:

> "The cat sat on the mat."

When the model is trying to understand the word *sat*, it doesn't just look at *sat* by itself. It wants to consider other words like *cat* (the subject) and *mat* (likely the object). Attention gives each token a way to "consult" earlier tokens and decide how important they are.

This is done mathematically by projecting each token into three roles: Query (Q), Key (K), and Value (V).

- Query (Q): "What am I looking for?"
- Key (K): "What do I offer?"
- Value (V): "What information do I carry?"

Step 1: Creating Q, K, and V

For every input vector of size `C` (e.g., 768), the code performs three separate linear projections (matrix multiplications). These produce Q, K, and V vectors of smaller size, divided among attention heads.

In the code:

```
matmul_forward(acts.q, acts.ln1, params.wq, params.bq, B, T, C, C);
matmul_forward(acts.k, acts.ln1, params.wk, params.bk, B, T, C, C);
matmul_forward(acts.v, acts.ln1, params.wv, params.bv, B, T, C, C);
```

Here:

- `acts.ln1` is the normalized input from the previous step.
- `params.wq`, `params.wk`, `params.wv` are the weight matrices.
- The output shapes are `(B, T, C)`.

So each token now has three new representations: Q, K, and V.

Step 2: Computing Attention Scores

For each token at position `t`, we want to know how much it should pay attention to every earlier token (including itself). This is done with a dot product between its Query and all Keys.

Mathematically:

```
score[t][u] = (Q[t]  K[u]) / sqrt(dk)
```

- `t` = current token.
- `u` = another token at or before `t`.
- `sqrt(dk)` is a scaling factor (dk = size of each head) to keep values stable.

In the code, these dot products are done explicitly in loops.

Step 3: Applying the Causal Mask

GPT-2 is an autoregressive model, meaning it only predicts the future from the past, not the other way around. To enforce this, the attention matrix is masked:

- Token at position `t` can only look at positions `≤ t`.
- Anything beyond `t` is set to a very negative value (`-1e9`), which becomes effectively zero after softmax.

This ensures, for example, that when predicting the 3rd word, the model doesn't cheat by looking at the 4th.

Step 4: Turning Scores into Probabilities

The scores are raw numbers that can be large and unstable. To convert them into meaningful weights, the code applies softmax:

```
attention_weights[t][u] = exp(score[t][u]) / Σ exp(score[t][v])
```

This makes all weights positive and ensures they sum to 1. Now each token has a probability distribution over earlier tokens.

Example for the word *sat*:

Attended Token	Raw Score	After Softmax
The	1.2	0.10
cat	3.4	0.80
sat (itself)	0.7	0.10
on	masked	0.00

Clearly, *sat* focuses most strongly on *cat*.

Step 5: Weighted Sum of Values

Once the attention weights are computed, the model uses them to take a weighted sum of the Value vectors:

```
output[t] = Σ attention_weights[t][u] * V[u]
```

This produces a new representation for each token that blends in information from others.

For *sat*, its new vector will be mostly influenced by *cat*, but also a little by *The* and itself.

Step 6: Multi-Head Attention

In practice, attention is split into multiple heads (12 for GPT-2 small). Each head works on smaller chunks of the vector (C/heads).

- Head 1 might focus on subject–verb relationships.
- Head 2 might track distances (like "how far back was this token?").
- Head 3 might specialize in punctuation.

After all heads compute their outputs, the results are concatenated and projected back into size `C` with another matrix multiply.

Step 7: Residual Connection

Finally, the output of the attention block is added back to the original input (residual connection). This keeps the original signal flowing, even if the attention introduces distortions.

```
residual_forward(acts.residual2, acts.ln1, acts.att, B*T, C);
```

This ensures information isn't lost and gradients flow smoothly during training.

Why It Matters

Attention is the mechanism that lets GPT-2 capture relationships between words. Without it, the model would treat each token independently, losing context. By explicitly computing "who should I look at?" for every token, GPT-2 learns patterns like subject–verb agreement, long-distance dependencies, and even stylistic nuances.

Try It Yourself

1. Inspect the attention mask: Print out the scores before and after masking. Notice how future tokens are set to huge negative values.
2. Visualize weights: Run attention on a short sentence and plot the weights. You'll see which words attend to which.
3. Change sequence length: Try increasing `T` and observe how computation grows quadratically (`T`^{2}). Attention is expensive!
4. Experiment with heads: Force the model to use only 1 head instead of 12. See how this limits the diversity of patterns it can capture.
5. Check sum of weights: For one token, verify that all attention weights add up to 1.0 after softmax.

The Takeaway

Attention is what makes transformers powerful. It allows each word to dynamically decide which other words matter for understanding its role in a sentence. In `train_gpt2.c`, this process is spelled out with explicit loops and matrix multiplications, so you can follow every step of the math. Understanding this section gives you the key to why GPT-2-and all modern LLMs-work so well.

34. MLP: GEMMs and Activation Functions

After the attention block lets tokens "talk to each other," GPT-2 applies a second kind of transformation called the MLP block (multi-layer perceptron). Unlike attention, which mixes information between tokens, the MLP processes each token independently, enriching its internal representation. Even though it looks simpler than attention, the MLP is essential for capturing complex relationships in language.

What the MLP Does

Every token's vector (size `C`, e.g., 768 for GPT-2 small) goes through:

1. Linear expansion: project from size `C` to size `4C` (3072 in GPT-2 small).
2. Nonlinear activation: apply the GELU function, which adds flexibility.
3. Linear projection back: reduce size from `4C` back to `C`.
4. Residual connection: add the input vector back to the output, keeping the original signal intact.

This allows the model to not only share information between tokens (via attention) but also refine how each token represents itself.

Step 1: Expanding with a Matrix Multiply

The first step is to expand each token's vector from 768 to 3072 dimensions. This is done with a general matrix multiply (GEMM):

```
matmul_forward(acts.mlp_in, acts.ln2, params.wfc, params.bfc, B, T, C, 4*C);
```

- `acts.ln2`: the normalized input from the previous residual.
- `params.wfc`: the weight matrix of size `(C, 4C)`.
- `params.bfc`: bias vector of size `(4C)`.
- `acts.mlp_in`: the result, shape `(B, T, 4C)`.

Think of it like stretching a rubber band-suddenly, the token has much more room to express richer features.

Step 2: GELU Activation

After expansion, each number passes through GELU (Gaussian Error Linear Unit).

The formula:

```
GELU(x) = 0.5 * x * (1 + tanh(√(2/ ) * (x + 0.044715 * x^3)))
```

This looks complicated, but the key idea is:

- For small negative numbers, output 0 (ignore weak signals).
- For large positive numbers, output x (pass strong signals).
- For numbers in between, it smoothly blends.

Unlike ReLU, which just chops off negatives, GELU lets small signals through in a probabilistic way. This makes it better for language, where even small hints matter.

Analogy: Imagine you're grading homework. If an answer is completely wrong, you give 0 points (ReLU style). If it's perfect, you give full credit. But if it's partially right, you give partial credit. GELU behaves like that-soft, nuanced grading.

Step 3: Projecting Back Down

Once the token vector has been expanded and passed through GELU, it's projected back to the original size `C`:

```
matmul_forward(acts.mlp_out, acts.mlp_in_gelu, params.wproj, params.bproj, B, T, 4*C, C);
```

- `params.wproj`: the projection weights, size `(4C, C)`.
- `params.bproj`: bias, size `(C)`.
- `acts.mlp_out`: result, shape `(B, T, C)`.

Now each token has gone through a non-linear "thinking step," mixing and reshaping features.

Step 4: Residual Connection

Just like with attention, the MLP output is added back to the input:

```
residual_forward(acts.residual3, acts.residual2, acts.mlp_out, B*T, C);
```

This means the token keeps its old representation while adding the new refinements. If the MLP makes a mistake early in training, the residual ensures the token doesn't lose all its meaning.

Inside the Code: Simplicity

Even though MLPs in deep learning libraries like PyTorch are one-liners (`nn.Linear` + `nn.GELU` + `nn.Linear`), here in C you see every step spelled out:

- First GEMM expands to 4C.
- Loop applies GELU element by element.
- Second GEMM projects back to C.
- Residual adds input and output.

It's like watching a magician reveal the trick instead of just seeing the final illusion.

Why the Expansion Matters

You might ask: why expand to 4C and then shrink back? Why not just keep the size the same?

The expansion allows the model to capture more complicated combinations of features. By spreading information out, applying a nonlinear transformation, and then compressing it again, the model can discover patterns that wouldn't fit in the smaller space.

Think of it like brainstorming on a huge whiteboard. You spread out all your ideas (4C), reorganize them, and then condense the best ones into a neat summary (C).

Example Walkthrough

Let's say we're processing the token "cat" in the sentence *The cat sat*.

1. Input vector (size 768): `[0.12, -0.08, 0.33, …]`
2. After first matrix multiply: expanded to `[1.2, -0.9, 0.5, …]` (size 3072).
3. After GELU: `[1.1, -0.0, 0.4, …]` (smooth nonlinearity).
4. After projection: back to `[0.15, -0.02, 0.27, …]` (size 768).
5. Add back original input: `[0.27, -0.10, 0.60, …]`.

Now "cat" has been enriched with new internal features that help the model predict what comes next.

Why It Matters

The MLP is the part of GPT-2 that lets each token refine itself. Attention gives context from neighbors, but the MLP deepens the representation of the token itself. Without it, the model would lack the ability to detect fine-grained patterns.

Try It Yourself

1. Print intermediate sizes: Add debug prints to see how token vectors grow to 4C and shrink back to C.
2. Swap activation: Replace GELU with ReLU in the code and train. Compare losses-you'll notice GPT-2 prefers GELU.
3. Disable residual: Temporarily remove the residual add. Watch how the model struggles to learn, because it can't preserve information.
4. Visualize values: Track how many values are near 0 before and after GELU. You'll see GELU softly zeroes out weak signals.
5. Smaller expansion: Try changing 4C to 2C in the code. You'll save memory but lose accuracy, since the MLP has less expressive power.

The Takeaway

The MLP block is a token's personal deep thinker. It stretches the representation wide, filters it through GELU, compresses it again, and then adds it back to the original. While attention handles the conversation between words, the MLP ensures each word processes and refines its own role. Together, they create the layered reasoning ability that makes GPT-2 so powerful.

35. LayerNorm on CPU (Step-by-Step)

One of the most important but often overlooked ingredients in GPT-2 is Layer Normalization, or LayerNorm for short. While attention and MLPs are the big stars, LayerNorm is like the stage crew keeping everything running smoothly behind the scenes. It ensures the numbers flowing through the network stay stable and balanced, preventing explosions or collapses that could make training impossible. In `train_gpt2.c`, LayerNorm is implemented with explicit loops so you can see every calculation. Let's walk through it carefully.

Why Do We Need Normalization?

Imagine a classroom where every student talks at different volumes. Some whisper, some shout. If you try to listen to all of them at once, the loud voices drown out the quiet ones.

Neural networks face a similar problem. The outputs of layers can have wildly different scales. If one dimension of a vector is much larger than the others, it dominates. Training becomes unstable, and gradients may vanish or explode.

LayerNorm fixes this by ensuring that, for each token at each layer, the vector has:

- Mean = 0 (centered around zero)
- Variance = 1 (consistent spread of values)

After that, trainable parameters scale and shift the result so the model can still learn flexible transformations.

The Math Behind LayerNorm

For a given token vector `x` of size `C` (e.g., 768):

1. Compute mean:

   ```
    = (1/C) * Σ x[i]
   ```

2. Compute variance:

   ```
   ² = (1/C) * Σ (x[i] -  )²
   ```

3. Normalize:

   ```
   x_norm[i] = (x[i] -  ) / sqrt( ² +  )
   ```

   where is a tiny constant (like 1e-5) to avoid division by zero.

4. Scale and shift with trainable weights `g` (gamma) and `b` (beta):

   ```
   y[i] = g[i] * x_norm[i] + b[i]
   ```

So the final output has controlled statistics, but still enough flexibility for the model to adjust.

The Code in `train_gpt2.c`

Here's a simplified version from the repository:

```
void layernorm_forward(float* out, float* inp, float* weight, float* bias, int B, int T, int
    for (int b = 0; b < B; b++) {
        for (int t = 0; t < T; t++) {
            float* x = inp + b*T*C + t*C;
            float* o = out + b*T*C + t*C;

            // mean
            float mean = 0.0f;
            for (int i = 0; i < C; i++) mean += x[i];
            mean /= C;

            // variance
            float var = 0.0f;
            for (int i = 0; i < C; i++) {
                float diff = x[i] - mean;
                var += diff * diff;
            }
            var /= C;

            // normalize, scale, shift
            for (int i = 0; i < C; i++) {
                float norm = (x[i] - mean) / sqrtf(var + 1e-5f);
                o[i] = norm * weight[i] + bias[i];
            }
        }
    }
}
```

Notice the structure:

- Outer loops over batch `B` and sequence length `T`.
- Inner loops compute mean, variance, and then apply normalization per token vector of length `C`.
- `weight` and `bias` are the learnable gamma and beta.

This is exactly what LayerNorm means: normalize *each layer's inputs per token.*

Example Walkthrough

Suppose we have a single token vector (C=4) = `[2.0, -1.0, 3.0, 0.0]`.

1. Mean: `(2 - 1 + 3 + 0)/4 = 1.0`

2. Variance: `((2-1)² + (-1-1)² + (3-1)² + (0-1)²)/4 = (1 + 4 + 4 + 1)/4 = 2.5`
3. Normalize: subtract mean and divide by sqrt(2.5):

   ```
   [ (2-1)/1.58, (-1-1)/1.58, (3-1)/1.58, (0-1)/1.58 ]
   = [0.63, -1.26, 1.26, -0.63]
   ```

4. Scale and shift (say weight=[1,1,1,1], bias=[0,0,0,0]):

   ```
   [0.63, -1.26, 1.26, -0.63]
   ```

Now the vector has mean 0, variance 1, and is ready for the next layer.

Analogy

Think of LayerNorm like a baking recipe. If one ingredient is way too strong (like adding five times too much salt), the whole dish is ruined. LayerNorm tastes the mixture, balances all the flavors, and then lets you adjust the seasoning with learnable gamma (scale) and beta (shift).

Why It Matters

Without LayerNorm, the model would quickly become unstable:

- Some tokens would dominate, while others fade.
- Gradients could explode, making loss jump wildly.
- Training would be inconsistent between batches.

With LayerNorm, each layer works with clean, normalized inputs. This allows deeper stacks of attention and MLP blocks to learn reliably.

Try It Yourself

1. Print statistics: Add debug code to check the mean and variance before and after LayerNorm. Before: mean 0, variance 1. After: mean 0, variance 1.
2. Remove LayerNorm: Comment out LayerNorm in the code. Watch training collapse-loss will not decrease properly.
3. Change epsilon: Try making `= 1e-1` or `= 1e-12`. See how too large or too small values can break stability.
4. Observe gamma and beta: Initialize gamma=1, beta=0. During training, watch how these parameters drift, fine-tuning normalization.
5. Experiment with batch norm: Replace LayerNorm with BatchNorm (not typical for transformers). You'll see it doesn't work well, because transformers process variable-length sequences where per-batch statistics vary too much.

The Takeaway

LayerNorm is the quiet but critical stabilizer in GPT-2. It ensures every token vector is balanced, centered, and scaled before moving into attention or MLP. In `train_gpt2.c`, you see exactly how it works: compute mean, compute variance, normalize, then scale and shift. Even though it's just a few lines of C code, it's one of the main reasons deep transformers can stack dozens of layers without breaking.

36. Residual Adds and Signal Flow

Once embeddings, attention, and MLP blocks are computed, there's still one piece left to keep the whole network stable and effective: residual connections. In `train_gpt2.c`, these appear in functions like `residual_forward`, where outputs of a layer are added back to their inputs. This simple-looking step is one of the key reasons GPT-2 and other deep transformer models can stack many layers without collapsing.

The Core Idea

A residual connection says:

```
output = input + transformation(input)
```

Instead of replacing the old representation with the new one, the model adds them together. That way, the original signal always survives, even if the new transformation is noisy or imperfect.

Think of it like taking lecture notes. Each time the teacher explains more, you don't throw away your old notes. You add new details next to them. That way, you preserve everything learned so far, while layering new insights on top.

Why Residuals Are Crucial

1. Preventing information loss: If you only applied transformations, some features might vanish forever. Adding the input back ensures no information is lost.
2. Helping gradients flow: During backpropagation, gradients must travel backward through many layers. Without shortcuts, they can vanish or explode. Residuals create direct paths for gradients, making learning stable.
3. Improving training speed: With residuals, deeper networks converge faster because the model can "skip" bad transformations while still using the identity mapping.

The Code in `train_gpt2.c`

Here's the implementation of residual addition:

```
void residual_forward(float* out, float* inp1, float* inp2, int N) {
    for (int i = 0; i < N; i++) {
        out[i] = inp1[i] + inp2[i];
    }
}
```

It's deceptively simple:

- `inp1` is the original input.
- `inp2` is the new transformation (from attention or MLP).
- `out` stores the sum.
- `N` is the total number of floats (`B * T * C`).

Even though it's just a single line inside a loop, this is what makes stacking 12+ transformer blocks possible.

Example Walkthrough

Suppose we're processing the token "cat."

- Input vector (simplified, size=3): `[0.5, -0.3, 0.7]`
- After attention block: `[0.2, 0.1, -0.4]`

Residual addition:

```
[0.5 + 0.2, -0.3 + 0.1, 0.7 - 0.4] = [0.7, -0.2, 0.3]
```

Now the representation of "cat" contains both the original signal and the contextual information from attention.

Later, after the MLP:

- Input again: `[0.7, -0.2, 0.3]`
- MLP output: `[0.1, -0.5, 0.4]`
- Residual: `[0.8, -0.7, 0.7]`

Step by step, the vector grows richer without losing its foundation.

Analogy

Residual connections are like building layers in Photoshop. Each new layer adds adjustments, but you always keep the original photo underneath. If a new adjustment is bad, you can still see the original. This makes the final composition stronger and safer to experiment with.

Residuals Across the Model

In GPT-2's forward pass, residuals appear in two main places inside each transformer block:

1. After attention:

   ```
   x = x + Attention(x)
   ```

2. After MLP:

   ```
   x = x + MLP(x)
   ```

Together with LayerNorm before each block, these form the backbone of the transformer architecture:

```
x → LayerNorm → Attention → Residual Add → LayerNorm → MLP → Residual Add
```

Why It Matters

Without residual connections, GPT-2 would struggle to train past a few layers. Deeper stacks would lose track of the original signal, gradients would vanish, and performance would stall. Residuals are the glue that holds the whole architecture together, enabling models with billions of parameters to train effectively.

Try It Yourself

1. Remove residuals: Temporarily comment out the `residual_forward` calls. Training will quickly fail-the loss won't decrease properly.
2. Print before/after: Inspect a token's vector before and after residual add. Notice how the numbers change smoothly rather than being overwritten.
3. Experiment with scaling: Try replacing `out[i] = inp1[i] + inp2[i];` with `out[i] = inp1[i] + 0.1 * inp2[i];`. This reduces the impact of the new transformation-sometimes used in advanced architectures.
4. Compare to skip-less RNNs: Research how older recurrent networks without residuals had trouble scaling deep. You'll see why residuals are a game changer.
5. Chain of signals: Track how a single token's vector evolves across all 12 layers. You'll notice it keeps its core identity while absorbing new context.

The Takeaway

Residual connections may look like a simple addition, but they're the key to deep learning's success in transformers. They preserve information, stabilize training, and allow GPT-2 to stack many layers without falling apart. In `train_gpt2.c`, this idea is laid bare: a few lines of C code implementing one of the most powerful tricks in modern neural networks.

37. Cross-Entropy Loss on CPU

After embeddings, attention, MLPs, LayerNorm, and residuals have done their job, the model produces logits-raw scores for every word in the vocabulary at every position in the sequence. But logits alone don't tell us if the model is "good" or "bad" at predicting the right word. To measure performance and guide learning, GPT-2 uses the cross-entropy loss function. In `train_gpt2.c`, this is implemented in the function `crossentropy_forward`.

What Are Logits?

At the final stage of the forward pass, each token position has a vector of length `V` (vocabulary size, ~50k). For example, the model might produce these logits for a tiny vocabulary of 3 words:

```
logits = [5.2, 1.1, -2.7]
```

Logits are just numbers-bigger means "more likely," smaller means "less likely"-but they aren't probabilities yet.

Step 1: Softmax – Turning Scores Into Probabilities

To compare predictions with the true target, we first convert logits into probabilities. The tool for this is the softmax function:

```
p[i] = exp(logits[i]) / Σ exp(logits[j])
```

Softmax has two important effects:

1. It makes all values positive.
2. It normalizes them so they sum to 1, forming a probability distribution.

Example:

- Logits: `[5.2, 1.1, -2.7]`
- Subtract max (5.2) for stability → `[0.0, -4.1, -7.9]`
- Exponentiate → `[1.0, 0.017, 0.0004]`
- Normalize → `[0.98, 0.017, 0.0004]`

Now the model is saying:

- Word 0: 98% chance
- Word 1: 1.7% chance
- Word 2: 0.04% chance

Step 2: Cross-Entropy – Measuring Mistakes

Cross-entropy compares the predicted probability for the correct word against the ideal case (probability = 1).

Formula:

```
loss = -log(probability_of_correct_word)
```

- If the model assigns high probability to the correct word, loss is small.
- If the model assigns low probability, loss is large.

Example:

- Correct word = Word 0, probability = 0.98 → loss = -log(0.98) 0.02 (excellent).
- Correct word = Word 1, probability = 0.017 → loss = -log(0.017) 4.1 (bad).

Step 3: Averaging Over the Batch

In practice, we don't train on just one word, but a batch of sequences. The code loops over every token in every batch, collects their losses, and averages them.

From `train_gpt2.c`:

```c
float loss = 0.0f;
for (int i = 0; i < B*T; i++) {
    int target = targets[i];
    float logit_max = -1e9;
    for (int j = 0; j < Vp; j++) {
        if (logits[i*Vp + j] > logit_max) logit_max = logits[i*Vp + j];
    }
    float sum = 0.0f;
```

```
    for (int j = 0; j < Vp; j++) {
        sum += expf(logits[i*Vp + j] - logit_max);
    }
    float log_sum = logf(sum);
    float correct_logit = logits[i*Vp + target];
    loss += (log_sum + logit_max - correct_logit);
}
loss /= (B*T);
```

What's happening here:

1. For each token, find the max logit (`logit_max`) to improve numerical stability.
2. Compute softmax denominator (`sum`).
3. Calculate log probability of the correct token.
4. Accumulate losses across all tokens.
5. Divide by total tokens (`B*T`) to get the average.

Numerical Stability Tricks

Without subtracting `logit_max`, `exp(logits)` can overflow. For example, `exp(1000)` is infinite. By subtracting the max, the largest logit becomes 0, so its exponential is 1, and all others are 1. This keeps numbers manageable while preserving the probability ratios.

Example With a Sentence

Sentence: *The cat sat on the mat.*

Suppose the model predicts probabilities for the last token:

- "mat": 0.85
- "dog": 0.10
- "car": 0.05

Correct word = "mat."

Loss = `-log(0.85)` `0.16`.

If instead the model guessed "dog" with 0.10, loss = `-log(0.10)` `2.3`. Higher penalty for being wrong.

Analogy

Cross-entropy is like grading multiple-choice exams. If the student picks the right answer confidently (high probability), they lose almost no points. If they're hesitant or wrong, they lose more points. Over many questions (tokens), you calculate their average score-the training loss.

Why It Matters

Cross-entropy loss is the guiding signal for the entire training process. It tells the optimizer:

- "Increase probability for the right words."
- "Decrease probability for the wrong words."

Without it, GPT-2 would have no way of knowing whether its predictions are improving.

Try It Yourself

1. Check baseline loss: Before training, print the loss. It should be close to `log(vocab_size)` (~10.8 for GPT-2 small), which corresponds to random guessing.
2. Inspect softmax sums: For one token, sum all probabilities. It should equal ~1.0.
3. Force the wrong answer: Temporarily change the target to an incorrect word. Watch how the loss shoots up.
4. Observe loss during training: Print loss every step. It should steadily decrease as the model learns.
5. Compare with accuracy: Track how often the model's top prediction matches the target. Loss and accuracy will move together, but loss is smoother and more informative.

The Takeaway

Cross-entropy loss turns raw model scores into a clear training signal. It penalizes wrong predictions, rewards confident correct ones, and ensures the optimizer knows exactly how to adjust weights. In `train_gpt2.c`, you see this implemented explicitly, without any library shortcuts-just loops, exponentials, and logs. Understanding this section is key to understanding how GPT-2 learns from its mistakes.

38. Putting It All Together: The `gpt2_forward` Function

Up to this point, we've explored the forward pass piece by piece - embeddings, attention, feed-forward layers, layer normalization, residual connections, and finally the loss. But a model doesn't live as disconnected pieces; they all come together in a single function that drives inference: `gpt2_forward`. This function is where the code actually executes the story we've been telling. Let's walk through it carefully so you can see how every building block plugs into the whole picture.

The role of `gpt2_forward`

Think of `gpt2_forward` as the director of the play. The actors (embeddings, attention, MLP, layernorm, etc.) already know their roles. The director calls them on stage in the right order and makes sure they hand the script off smoothly to the next actor. In our case:

1. Tokens come in as integers (word IDs).
2. They're turned into embeddings (token + position).
3. Each transformer block processes the sequence through attention, MLP, layernorm, and residuals.
4. The final hidden states are mapped back into vocabulary space.
5. If labels are provided, a loss is computed.

Code skeleton

Here's a simplified excerpt of the real function from `train_gpt2.c` (slightly shortened for readability):

```
void gpt2_forward(GPT2 *model, int *tokens, int *labels, int B, int T) {
    // Step 1: Embedding lookup
    embedding_forward(model->token_embedding, tokens, B, T);
    embedding_forward(model->position_embedding, positions, B, T);

    // Step 2: Transformer blocks
    for (int l = 0; l < model->config->n_layer; l++) {
        attention_forward(&model->blocks[l].attn, ...);
        mlp_forward(&model->blocks[l].mlp, ...);
        layernorm_forward(&model->blocks[l].ln, ...);
        residual_forward(...);
    }

    // Step 3: Final normalization + logits
```

```
    layernorm_forward(model->final_ln, ...);
    matmul_forward(model->lm_head, ...); // project to vocab

    // Step 4: Loss (optional)
    if (labels != NULL) {
        crossentropy_forward(...);
    }
}
```

Don't worry if this looks intimidating - we'll decode each part in plain language.

Step 1: Embedding lookup

Before the model can reason about words, it has to map token IDs into continuous vectors. That's where embedding tables come in:

- `token_embedding` converts each integer token ID into a dense vector of size `C` (the channel dimension).
- `position_embedding` does the same for positions (0, 1, 2, …, T-1).
- These two are added together, giving each token both a meaning (word identity) and a place in the sentence.

Step 2: Transformer blocks

Each block is like a mini-pipeline that processes the sequence and passes it forward. Inside the loop:

1. Attention: compares tokens with each other, weighted by learned Q/K/V projections.
2. MLP: expands each token vector, applies a nonlinear GELU activation, then projects back down.
3. LayerNorm: normalizes values for stable training and inference.
4. Residual: adds the input of the block back to its output to keep information flowing.

This loop runs `n_layer` times - for GPT-2 124M, that's 12 blocks.

Step 3: Final normalization and logits

After the last block, the sequence of token representations goes through a final layer normalization. Then, a large matrix multiplication (`lm_head`) projects each token's hidden state into the size of the vocabulary (50,000 for GPT-2). The result is a tensor of shape `(B, T, vocab_size)` containing the raw prediction scores for each next token.

Step 4: Optional loss computation

If you pass `labels` (the correct next tokens) into `gpt2_forward`, the function calls `crossentropy_forward`. This compares the predicted scores with the true tokens and outputs a single number: the loss. The loss tells you "how wrong" the model was, which is critical during training. But if you're only doing inference, you don't need this step.

How the pieces connect

Here's a table that maps our earlier sections to the parts of `gpt2_forward`:

Code Step	Concept	Section Covered Earlier
Embeddings	token + positional vectors	32
Attention	QKV projections, masking, softmax	33
MLP	feed-forward expansion and compression	34
LayerNorm	normalization for stability	35
Residual	skip connections for signal flow	36
CrossEntropy	comparing predictions with labels	37

So `gpt2_forward` is really just a neat orchestration of everything you've already learned.

Why it matters

Understanding `gpt2_forward` gives you the complete mental picture of inference. It shows how embeddings, attention, MLP, normalization, and residuals work together in code to turn a batch of tokens into predictions. Without this integration step, the model would just be a collection of disconnected parts.

Try it yourself

1. Print shapes: Add `printf` statements inside `gpt2_forward` to print tensor shapes after embeddings, after each block, and after logits. This helps you see the data flow.
2. Use a single block: Change the loop to run only 1 transformer block instead of all 12. Watch how the outputs degrade - the model loses depth of reasoning.
3. Disable position embeddings: Comment out the line that adds `position_embedding`. Try running inference. You'll notice the model becomes worse at handling word order.
4. Loss vs no loss: Call `gpt2_forward` with and without labels. Compare the difference - with labels you get a scalar loss, without labels you just get logits.
5. Smaller vocab: Try using a toy tokenizer with a small vocabulary and rerun the projection step. You'll see the logits shrink to `(B, T, tiny_vocab_size)`.

The takeaway

`gpt2_forward` is where GPT-2 inference really happens. It ties together every concept - embeddings, attention, feed-forward layers, normalization, residuals, and the final projection into vocabulary space. Once you understand this function, you don't just know the pieces of GPT-2, you know how they actually work together to produce predictions. It's the "main stage" of inference, and mastering it means you can confidently say you understand how a transformer runs forward on CPU.

39. OpenMP Pragmas for Parallel Loops

CPU training in `train_gpt2.c` is intentionally "plain C," but it still squeezes out a lot of speed by adding a few OpenMP pragmas (`#pragma omp …`) in the hottest loops. OpenMP lets the compiler split a loop's iterations across multiple CPU cores-no threads to create by hand, no locks to manage. If you compile without OpenMP support, these pragmas are simply ignored and the code still runs (just slower).

Below we'll (1) show exactly where OpenMP is used, (2) explain why those loops are good candidates, and (3) offer practical tips to get solid speedups on your machine.

OpenMP in this file: where it appears and why

Location / Function	Pragma used	What's parallelized	Why it's a great fit
`matmul_forward`	`#pragma omp parallel for collapse(2)`	Outer loops over `b` (batch) and `t` (time)	Each `(b,t)` row computes an independent output vector; no write conflicts. Large, regular work = easy scaling.
`matmul_forward` (tiled)	`#pragma omp parallel for`	Collapsed `B*T` loop in tiles of `LOOP_UNROLL`	Heaviest compute in the model; tiling + per-thread tiles keep caches warm.
`matmul_backward` (part 1)	`#pragma omp parallel for collapse(2)`	Backprop into `inp` over `(b,t)`	Each `(b,t)` reads weights and `dout`, writes a private slice of `dinp` → no overlap.
`matmul_backward` (part 2)	`#pragma omp parallel for`	Backprop into `weight/bias` over `o` (output channel)	Each thread owns one output channel's gradient row, avoiding atomics.

Location / Function	Pragma used	What's parallelized	Why it's a great fit
`softmax_forward`	`#pragma omp parallel for collapse(2)`	Over `(b,t)` positions	Each softmax is independent; perfect "embarrassingly parallel" loop.
`attention_forward`	`#pragma omp parallel for collapse(3)`	Over `(b, t, h)` = batch, time, head	Per `(b,t,h)` head's work is independent; big 3-D grid parallelizes extremely well.

A few key patterns to notice:

- Collapse clauses (`collapse(2)` / `collapse(3)`) fuse nested loops into one big iteration space so the scheduler can distribute more, smaller chunks-great for load-balancing when `B`, `T`, or `NH` are modest.
- Parallelizing along independent dimensions avoids race conditions. For example, in `matmul_backward` the pass that writes `dinp[b,t,:]` is parallelized over `(b,t)` so no two threads update the same memory.
- Own-your-row strategy: when accumulating `dweight`, the loop goes over `o` (output channels) so each thread writes its own gradient row `dweight[o,:]`. No atomics needed.

Quick refresher: what OpenMP is doing

A typical pattern looks like:

```
#pragma omp parallel for collapse(2)
for (int b = 0; b < B; b++) {
    for (int t = 0; t < T; t++) {
        // compute outputs for (b, t) independently
    }
}
```

When compiled with OpenMP, the compiler creates a team of threads and divides the iteration space (`B*T` in this example) among them. Each thread executes its assigned iterations; when the loop finishes, threads sync at an implicit barrier.

Because each `(b,t)` (or `(b,t,h)`) writes to a disjoint slice of the output arrays, there's no need for locks or atomics. This is why these loops scale cleanly across cores.

Enabling OpenMP, safely

- The source guards the OpenMP header with:

```
#ifdef OMP
#include <omp.h>
#endif
```

  so you can define `OMP` in your build and add your compiler switch. Example (GCC/Clang):

  `-D OMP -fopenmp`

- If you forget `-fopenmp` (or your platform's equivalent), the pragmas are ignored and the program runs single-threaded.
- You can control threads at runtime:

  `export OMP_NUM_THREADS=8`

  A good rule of thumb is to start with the number of physical cores on your CPU.

Why these loops benefit the most

1. Matrix multiplies dominate runtime. `matmul_forward`/`matmul_backward` consume the bulk of CPU time. Parallelizing them yields the largest end-to-end speedups.
2. Softmax is independent per position. Each `(b,t)` softmax computes a max, then exponentials and a sum-no cross-talk between positions.
3. Attention splits across batch/time/head. The triple loop over `(b,t,h)` has lots of work per iteration ($Q \cdot K$, softmax, weighted sum), making thread overhead negligible compared to useful compute.
4. Minimal synchronization and no atomics. By choosing iteration spaces that own exclusive output slices, we avoid costly synchronization.

Practical tips for better scaling

- Set `OMP_NUM_THREADS` to your CPU. Too many threads can hurt (oversubscription). Start with physical cores, then experiment.
- Pin threads (optional, advanced). Some OpenMP runtimes support `OMP_PROC_BIND=close` to improve cache locality.
- Mind memory bandwidth. On wide CPUs, GEMMs may become bandwidth-bound. Bigger `B`/`T` improves arithmetic intensity; tiny batches underutilize cores.

- Warm caches with tiling. The "tiled" `matmul_forward` keeps small accumulators in registers and reuses loaded weights across `LOOP_UNROLL` inner iterations.
- Avoid hidden sharing. If you add new parallel loops, ensure each thread writes to unique memory regions. If you must accumulate to the same place, restructure (like "own-your-row") or use per-thread scratch buffers then reduce.

Micro-walkthrough: why `collapse` helps

Consider `softmax_forward`:

```
#pragma omp parallel for collapse(2)
for (int b = 0; b < B; b++) {
    for (int t = 0; t < T; t++) {
        // 1) find max over V
        // 2) exp/log-sum
        // 3) normalize first V entries; zero out padded [V..Vp)
    }
}
```

If `B=4`, `T=64`, that's 256 independent softmaxes. With `collapse(2)`, OpenMP sees a single loop of 256 iterations to distribute evenly; without `collapse`, it might chunk by `b` first (only 4 big chunks), which can load-imbalance.

Common pitfalls (and how this code avoids them)

- Race conditions: Two threads writing the same `out[i]`. *Avoided by design:* each parallel loop writes distinct slices (e.g., per `(b,t)` or per `o`).
- False sharing: Threads write adjacent memory locations on the same cache line. It's minimized by the large, contiguous slices per thread (entire rows/tiles), but if you extend the code with fine-grained parallelism, keep this in mind.
- Tiny loops: Overhead can exceed work. The file parallelizes only large, hot loops (GEMMs, attention, softmax), not small scalar ops.

Try it yourself

1. Change thread count: Run with `OMP_NUM_THREADS=1,2,4,8,…` and log step time. Plot speedup vs. threads.
2. Toggle a pragma: Comment out `#pragma omp` in `matmul_forward` only. Measure the slowdown; you'll see where most time goes.

3. Experiment with `collapse`: Remove `collapse(2)` in `softmax_forward`. On small `B`, you'll likely see worse scaling.
4. Per-layer profiling: Print elapsed time around `matmul_forward`, `attention_forward`, and `softmax_forward` to see which benefits most on your CPU.
5. Schedule policy (advanced): Try `#pragma omp parallel for schedule(static)` vs. `dynamic` on a heavy loop to see if it changes load balance (defaults are usually fine here).

The takeaway

A handful of well-placed OpenMP pragmas deliver big wins on CPU by parallelizing the most expensive loops (GEMMs, attention, softmax) across cores-without complicating the code. The design ensures each thread works on independent slices, so there's no locking, no atomics, and very little overhead. If you compile with OpenMP enabled, you get fast, multi-core training; if not, you still have a clean, readable reference implementation.

40. CPU Memory Footprint and Performance

When you train GPT-2 on your CPU using `train_gpt2.c`, two big questions usually pop up almost immediately: *how much memory is this going to take?* and *how fast will it run?* Let's walk through both of these in a beginner-friendly way, so you understand not just what happens in the code, but why it behaves the way it does.

Memory: where does it all go?

Imagine training GPT-2 is like cooking a big meal in a small kitchen. You need space for ingredients, bowls for mixing, and counter space for preparing. Memory on your CPU is that kitchen. GPT-2 needs several "bowls" to hold different parts of the computation:

1. Parameters (the weights of the model). These are the "fixed recipe" - the actual numbers the network learns. They come from the checkpoint file you load at the start. For GPT-2 124M, this is about 124 million floating-point numbers. Each one takes 4 bytes, so just the weights are around 500 MB.

2. Optimizer state (AdamW). Training doesn't just adjust weights blindly; it keeps track of two extra moving averages for each weight, called m and v. That means for every single parameter, you store three numbers: the weight, m, and v. So memory for optimizer state is often double the size of the weights themselves. For GPT-2 124M, that's about 1 GB more.

3. Gradients. Every time we run a backward pass, we store how much each weight should change. That's another buffer roughly the same size as the weights - another 500 MB.

4. Activations (intermediate results). This is the sneaky one. Every forward pass produces temporary tensors like embeddings, attention maps, and feed-forward outputs. Their size depends on batch size (B) and sequence length (T). If B=4 and T=64, activations are a few hundred MB. If B=32 and T=1024, they can balloon to many gigabytes.

Here's a rough mental budget for GPT-2 124M with a small setup (B=4, T=64):

- Parameters: ~500 MB
- Optimizer state: ~1 GB
- Gradients: ~500 MB
- Activations: ~200–300 MB Total: ~2–2.5 GB

Even for the "tiny" GPT-2, you already need a couple gigabytes of RAM to train. On a laptop, this can quickly push you to the limit.

Performance: where does time go?

Now let's talk speed. When you run `train_gpt2.c` on CPU, you'll see lines like:

```
step 1: train loss 5.191576 (took 1927.230000 ms)
```

That "took X ms" tells you how long one step took. Why is it slow? Three main reasons:

1. Matrix multiplications (matmuls). These are the heart of neural networks. Every attention head and every MLP layer does them. On CPU, most of your step time is spent here. That's why the code uses OpenMP pragmas (`#pragma omp`) to parallelize loops across cores.
2. Attention softmax. Attention compares every token in a sequence with every other token. If your sequence length is 1024, that's over a million comparisons per head per layer. On CPU, this quadratic growth is painful.
3. Memory bandwidth. CPUs can only move numbers from RAM to cores so fast. Even if you had infinite FLOPs, you'd still be slowed down by how quickly you can fetch and store these huge tensors.

A simple experiment

You can see these effects yourself:

- Change batch size (B). Run with B=1, then with B=8. Notice how memory usage and step time scale up.

- Change sequence length (T). Try T=16, then T=256. You'll see attention costs grow dramatically.
- Change threads. Set `OMP_NUM_THREADS=1` versus `OMP_NUM_THREADS=8`. With more threads, you'll often see speedups, but only up to the number of physical cores your CPU has.

Why this matters

For beginners, CPU runs are perfect for learning:

- You can debug with small batches and short sequences.
- You can step into functions with a debugger and watch tensors being created.
- You don't need a GPU just to understand how training works.

But when it comes to *serious* training - larger GPT-2 models or even long sequences - CPU is simply too slow. What takes seconds on GPU may take minutes on CPU. That's why in practice, people use CPUs for learning and testing, and GPUs for large-scale training.

The takeaway

Training GPT-2 on CPU is like practicing piano on a small keyboard. It's slower, limited, and you can't play the biggest pieces, but it's great for learning the fundamentals. Memory usage comes from weights, optimizer state, gradients, and activations, and performance is dominated by matmuls and attention. Once you understand where the resources go, you can adjust batch size, sequence length, and threads to find the sweet spot for your machine.

Chapter 5. Training Loop (CPU Path)

41. Backward Pass Walkthrough

Up until now, we've spent all our time looking at the forward pass. That's the part of the model that takes tokens, pushes them through embeddings, attention, feed-forward layers, and finally produces logits or a loss. For inference, forward pass alone is enough. But if you want to train a model, forward is only half the story.

Training means adjusting the weights of the model so that its predictions become better over time. To do this, we need a way to figure out how wrong each weight was and in what direction it should move to reduce the loss. That's the job of the backward pass.

The backward pass is also called backpropagation. It's the algorithm that moves information in reverse through the network: from the loss, back through the final logits, through every

transformer block, down to the embeddings. Along the way, it calculates gradients - small numbers that tell us how much each weight contributed to the error.

The big idea: chain rule in action

At the heart of backpropagation is something very familiar from calculus: the chain rule. If the output of the network depends on many functions stacked together (embedding → attention → MLP → … → loss), then the derivative of the loss with respect to an early parameter is a product of partial derivatives through the entire chain.

Instead of writing long formulas, the code in `train_gpt2.c` simply calls each layer's backward function in reverse order. The gradient flows backward, step by step, and each layer computes its own contribution using local rules.

Think of it like a relay race, but run backwards: the loss hands a "blame baton" to the output head, which hands it back to the last transformer block, and so on, until it reaches the very first embedding table.

Walking through `gpt2_backward`

Here's a simplified sketch of how the backward function looks in the code (names shortened for readability):

```
void gpt2_backward(GPT2 *model, int *tokens, int *labels, int B, int T) {
    // Step 1: loss gradient
    crossentropy_backward(...);

    // Step 2: final projection (lm_head)
    matmul_backward(model->lm_head, ...);

    // Step 3: transformer blocks in reverse
    for (int l = model->config->n_layer - 1; l >= 0; l--) {
        residual_backward(...);
        layernorm_backward(&model->blocks[l].ln, ...);
        mlp_backward(&model->blocks[l].mlp, ...);
        attention_backward(&model->blocks[l].attn, ...);
    }

    // Step 4: embeddings
    embedding_backward(model->token_embedding, tokens, ...);
    embedding_backward(model->position_embedding, positions, ...);
}
```

Let's unpack this line by line.

Step 1: Starting from the loss

The journey begins with the loss function. In training, the most common loss is cross-entropy. Its backward function compares the predicted probabilities with the true labels and produces a gradient for the logits.

- If the model predicted "cat" with high confidence and the true label was "dog," the gradient will push the logits away from "cat" and toward "dog."
- This gradient is the starting signal that propagates backward through the entire network.

Step 2: Back through the output head

After the loss, the next stop is the final linear projection (`lm_head`). This is just a big matrix multiply that turns hidden states into vocabulary logits. Its backward function computes two things:

1. The gradient with respect to the weights of `lm_head`.
2. The gradient with respect to the hidden states that fed into it.

This hidden-state gradient is then passed back to the last transformer block.

Step 3: Transformer blocks in reverse

Here comes the heavy lifting. Each block has multiple components, and their backward functions are called in the exact opposite order of the forward pass.

1. Residual backward: the skip connection splits the gradient into two paths - one flowing back into the transformed output, one flowing back into the original input.
2. LayerNorm backward: computes gradients with respect to its scale (`gamma`) and shift (`beta`), and also passes gradients back to the normalized input.
3. MLP backward: applies the chain rule to the two linear layers and the GELU activation. The code reuses temporary values from the forward pass (like activations) to make this efficient.
4. Attention backward: this is the trickiest. It computes gradients for Q, K, and V projections, as well as for the softmaxed attention weights. It has to apply the causal mask again to ensure no illegal gradient flows.

This loop continues until all transformer blocks have been processed.

Step 4: Back to embeddings

Finally, the gradient reaches the embedding tables. This is where the model first looked up vectors for tokens and positions. Now it calculates how much each embedding contributed to the error. These gradients are added into the embedding matrices, telling the optimizer how to update them.

Why this matters

The backward pass is what makes learning possible. Without it, the model would forever output the same predictions, never improving. By flowing "blame" backwards, each parameter learns how to nudge itself so that the next forward pass is a little bit better.

Even though the code looks like a lot of function calls, the principle is simple: start from the loss, step backward through each layer, apply the chain rule locally, and collect gradients.

Try it yourself

1. Print gradient norms: Add a `printf` to see the average gradient magnitude at each layer. Notice how they change - sometimes exploding, sometimes vanishing.
2. Freeze a layer: Comment out `mlp_backward` for one block and see how the model fails to update properly.
3. Inspect embeddings: After training a few steps, dump a few rows of the token embedding matrix. You'll see the numbers changing because of gradient updates.
4. Tiny dataset experiment: Train on a very small dataset (like a 10-word corpus) and watch how the backward pass quickly pushes embeddings to memorize it.
5. Check symmetry: Compare the order of calls in `gpt2_forward` with `gpt2_backward`. They're exact opposites - forward builds, backward unbuilds.

The takeaway

Backpropagation is the learning engine of neural networks. In `llm.c`, the backward pass is written out explicitly, showing how gradients flow from the loss, through the output head, back through every transformer block, and finally into embeddings. Once you understand this flow, you can see how training stitches forward and backward together to slowly shape a random model into a working language model.

42. Skeleton of Training Loop

The backward pass gave us gradients, but gradients by themselves don't train a model. Training requires a loop: a cycle that repeatedly runs forward, backward, and update steps over and over until the model improves. This cycle is called the training loop, and it is the heartbeat of every deep learning program. In `train_gpt2.c`, the loop is written explicitly in C, which means you can see every piece instead of it being hidden away in a framework.

The basic rhythm

Every training step follows the same rhythm:

1. Get a batch of data (input tokens and their labels).
2. Run the forward pass to compute predictions and loss.
3. Run the backward pass to compute gradients.
4. Update weights using an optimizer like AdamW.
5. Log progress and, occasionally, validate.

This rhythm repeats thousands or millions of times. With each repetition, the weights shift slightly, nudging the model toward lower loss and better predictions.

How the loop looks in code

Here's a simplified sketch from `train_gpt2.c` (with some details omitted for clarity):

```
for (int step = 0; step < max_steps; step++) {
    // 1. Load batch of tokens and labels
    dataloader_next_batch(&train_loader, tokens, labels, B, T);

    // 2. Forward pass
    gpt2_forward(&model, tokens, labels, B, T);

    // 3. Zero gradients
    gpt2_zero_grad(&model);

    // 4. Backward pass
    gpt2_backward(&model, tokens, labels, B, T);

    // 5. Optimizer step (AdamW)
    adamw_update(&opt, &model, learning_rate);

    // 6. Logging and validation
```

```
    if (step % log_interval == 0) { print_loss(step, model.loss); }
    if (step % val_interval == 0) { run_validation(...); }
}
```

This loop captures the full training lifecycle: data, forward, backward, update, and monitoring.

Step 1: Batching data

The dataloader feeds the loop with small chunks of tokens. Instead of sending the whole dataset at once, it breaks it down into batches of size `B` (number of sequences per batch) and length `T` (number of tokens per sequence).

- Example: if `B=4` and `T=128`, each batch is 512 tokens long.
- Each sequence has a matching set of labels, which are simply the same tokens shifted one position ahead (so the model always predicts the *next* word).

This batching keeps memory use manageable and helps the model see many small samples instead of a few giant ones.

Step 2: Forward pass

The forward pass computes predictions for all tokens in the batch and calculates the loss. This is the "evaluation" step - how well did the model do on this batch? The result is stored in `model.loss`.

Step 3: Zeroing gradients

Before calculating new gradients, the old ones must be cleared out. If you skip this step, gradients from previous batches would accumulate and corrupt the update. In frameworks like PyTorch you'd call `optimizer.zero_grad()`. Here it's a plain C function:

```
gpt2_zero_grad(&model);
```

It walks through all parameters and resets their gradient buffers to zero.

Step 4: Backward pass

Now the backward function is called. It pushes gradients back through the network, computing how each weight influenced the error. At this point, every parameter has an associated gradient stored in memory.

Step 5: Optimizer update

With gradients ready, the optimizer (AdamW in this code) updates each parameter:

```
new_weight = old_weight - learning_rate * gradient (with AdamW tweaks)
```

This step is what actually changes the model. Without it, the model would never learn - the forward and backward passes would just repeat the same results forever.

Step 6: Logging and validation

Every few steps, the loop prints out useful numbers: current step, loss, time taken, and sometimes throughput (tokens per second). This feedback is important to check whether training is actually working.

Every few hundred or thousand steps, the loop also runs a validation pass on held-out data. This tells you whether the model is just memorizing training data or genuinely learning patterns that generalize.

Why the training loop matters

The training loop is deceptively simple, but it is the engine room of machine learning. Every improvement in model performance happens because this loop runs many times. By writing it explicitly in C, `llm.c` exposes details that high-level frameworks usually hide: zeroing gradients, passing pointers to arrays, calling backward and optimizer functions directly.

This makes it a perfect learning tool. You can see clearly:

- Where the data comes in,
- Where predictions are made,
- Where gradients are calculated,
- And where learning actually happens.

Try it yourself

1. Print the loss curve: Add a `printf` inside the loop and write the loss to a file. Plot it - you should see it decrease over time.
2. Change batch size: Set `B=1` vs. `B=8`. Notice how the loop becomes noisier with smaller batches but smoother with larger ones.
3. Skip backward: Comment out `gpt2_backward` and optimizer update. Run the loop. You'll see the loss never decreases - a clear demonstration that forward alone doesn't train.
4. Experiment with steps: Try `max_steps=10` vs. `max_steps=1000`. Short runs show no improvement; longer runs start to reduce the loss.
5. Slow it down: Insert a `sleep(1);` inside the loop. This makes the rhythm visible step by step, so you can literally watch the model "breathe" as it trains.

The takeaway

The skeleton of the training loop is the core cycle of learning. It feeds data into the model, computes predictions, finds errors, sends them backward, updates weights, and logs progress. Everything else - optimizers, schedulers, distributed training, mixed precision - is just an enhancement of this basic loop. If you understand how this loop works in `llm.c`, you understand the beating heart of deep learning training.

43. AdamW Implementation in C

Training a neural network is about adjusting millions of parameters so that the model gradually becomes better at predicting text. The function `gpt2_update` in `train_gpt2.c` is responsible for this adjustment. It implements the AdamW optimizer, one of the most widely used algorithms in deep learning. Let's walk through both the theory and the actual implementation.

From Gradient Descent to AdamW

The most basic optimizer is gradient descent:

```
new_param = old_param - learning_rate * gradient
```

This approach works, but it has weaknesses. The step size (learning rate) must be tuned carefully: too small and training is slow, too large and training diverges. Moreover, all parameters use the same step size, even though some may need gentler updates.

AdamW improves this by keeping track of moving averages of gradients and scaling updates adaptively. It also introduces weight decay, which prevents parameters from growing too large and helps regularize the model.

How AdamW Works

AdamW combines several techniques into a single update rule. First, it uses momentum: instead of relying only on the current gradient, it averages recent gradients. This smooths noisy updates. Second, it maintains a running estimate of the squared gradient, which scales down steps in directions where gradients are consistently large. These are sometimes called the first and second moments.

Since both running averages start at zero, the algorithm applies bias correction during the first few steps. Without this, the early updates would be too small. Finally, AdamW applies weight decay directly in the update, shrinking parameter values slightly each step.

Putting it together, each parameter update looks like this:

```
m_t =  1 * m_(t-1) + (1 -  1) * g_t
v_t =  2 * v_(t-1) + (1 -  2) * g_t²
m̂_t = m_t / (1 -  1^t)
v̂_t = v_t / (1 -  2^t)

new_param = old_param - lr * ( m̂_t / (sqrt(v̂_t) +  ) +   * old_param )
```

Here `m` is momentum, `v` is variance, `lr` is learning rate, is a small constant for stability, and is the weight decay factor.

The Implementation in `train_gpt2.c`

```
void gpt2_update(GPT2 *model, float learning_rate, float beta1, float beta2,
                 float eps, float weight_decay, int t) {
    if (model->m_memory == NULL) {
        model->m_memory = (float*)calloc(model->num_parameters, sizeof(float));
        model->v_memory = (float*)calloc(model->num_parameters, sizeof(float));
    }

    for (size_t i = 0; i < model->num_parameters; i++) {
        float param = model->params_memory[i];
        float grad = model->grads_memory[i];
```

```
        float m = beta1 * model->m_memory[i] + (1.0f - beta1) * grad;
        float v = beta2 * model->v_memory[i] + (1.0f - beta2) * grad * grad;

        float m_hat = m / (1.0f - powf(beta1, t));
        float v_hat = v / (1.0f - powf(beta2, t));

        model->m_memory[i] = m;
        model->v_memory[i] = v;
        model->params_memory[i] -= learning_rate *
            (m_hat / (sqrtf(v_hat) + eps) + weight_decay * param);
    }
}
```

The first `if` block allocates memory for the moving averages `m` and `v` the first time the optimizer runs. Then, for each parameter, the code computes the new averages, applies bias correction, and finally updates the parameter with the AdamW formula.

Example Walkthrough

Suppose we have a parameter `w = 0.5` with gradient `g = 0.2` on the first training step. Using $_1 = 0.9$ and $_2 = 0.999$:

- Momentum:

  ```
  m = 0.9 * 0 + 0.1 * 0.2 = 0.02
  ```

- Variance:

  ```
  v = 0.999 * 0 + 0.001 * 0.04 = 0.00004
  ```

- Bias correction:

  ```
  m̂ = 0.02 / (1 - 0.9) = 0.2
  v̂ = 0.00004 / (1 - 0.999) = 0.04
  ```

- Final update (lr $= 0.001$, weight_decay $= 0.01$):

  ```
  update = 0.001 * (0.2 / sqrt(0.04) + 0.01 * 0.5)
         = 0.001 * (1.0 + 0.005)
         = 0.001005
  ```

So the parameter becomes `w = 0.498995`.

Intuition

Think of a ball rolling down a slope. The gradient is the slope itself. Momentum makes the ball keep rolling even if the slope flattens briefly. The variance term makes the ball slow down on rocky ground where the slope changes rapidly. Bias correction ensures the ball doesn't move too timidly at the start. Weight decay adds friction so the ball doesn't roll out of control.

Why It Matters

Optimizers are the difference between a model that trains smoothly and one that diverges or gets stuck. AdamW became popular because it combines stability with efficiency. It automatically adapts to each parameter's scale, reduces the need for manual learning rate tuning, and includes weight decay in a principled way. For GPT-style models with hundreds of millions of parameters, these qualities make training feasible.

Try It Yourself

1. Change the learning rate from `0.001` to `0.01` in the code and see how quickly the model diverges.
2. Set `weight_decay = 0` and compare validation loss after a few epochs. The model might overfit more quickly.
3. Print out the first 10 values of `m_memory` and `v_memory` during training to watch how they evolve over steps.
4. Replace AdamW with plain SGD (just `param -= lr * grad`) and compare training speed and stability.
5. Experiment with $_1 = 0$ (no momentum) or $_2 = 0$ (no variance smoothing) and see how noisy updates become.

The Takeaway

AdamW provides a balance of speed, stability, and generalization. In practice, it allows models like GPT-2 to train much more reliably than with vanilla gradient descent. The C implementation in `llm.c` demonstrates that beneath the math, it's just a simple loop applying a few arithmetic operations for each parameter.

44. Gradient Accumulation and Micro-Batching

Modern language models are enormous, and so are the batches of text we would like to feed them during training. But real hardware has limits: a single GPU or CPU may not have enough memory to process a large batch in one go. To solve this, training code often uses

gradient accumulation and micro-batching. Both ideas allow us to simulate training with larger batches without requiring more memory than our hardware can provide.

What Problem Are We Solving?

When you process a batch of data, you run forward and backward passes to calculate gradients. If your batch size is very large, you get smoother gradients (less noisy), which often helps the model converge better. But large batches may not fit in memory.

Imagine trying to train with a batch of 1024 sequences on a GPU that can only handle 128 sequences at once. Without tricks, you would be forced to use the smaller batch size and give up the benefits of larger batches. Gradient accumulation fixes this problem by letting you split the big batch into smaller micro-batches, process them one at a time, and accumulate the results as if you had processed the big batch all at once.

How It Works in Practice

Let's say we want an effective batch size of 1024, but our hardware only supports 128. We split the big batch into 8 micro-batches of 128 each:

1. Run forward + backward on micro-batch 1, store the gradients.
2. Run forward + backward on micro-batch 2, add its gradients to the stored ones.
3. Repeat until all 8 micro-batches are processed.
4. Once gradients for all 8 are accumulated, perform the optimizer update.

The important part is step 4: we only update the parameters once per effective batch, not after each micro-batch. This preserves the effect of training with a large batch.

Pseudocode Example

Here's how this might look in simplified pseudocode:

```
int accumulation_steps = 8;
for (int step = 0; step < total_steps; step++) {
    zero_grad(&model);
    for (int i = 0; i < accumulation_steps; i++) {
        dataloader_next_batch(&train_loader, tokens, labels, B, T);
        forward(&model, tokens, labels, B, T);
        backward(&model, tokens, labels, B, T);
        // do NOT call optimizer update yet
    }
```

```
    adamw_update(&model, lr, beta1, beta2, eps, weight_decay, step+1);
}
```

Notice how the optimizer only runs once per outer loop iteration, even though gradients were accumulated across multiple micro-batches.

Why Gradient Accumulation Helps

- Memory efficiency: You can train with larger effective batch sizes without needing more hardware.
- Training stability: Larger batches reduce the variance of gradients, making training less noisy.
- Flexibility: You can scale effective batch size up or down depending on your needs without changing hardware.

Micro-Batching vs. Accumulation

Micro-batching refers to the act of splitting a batch into smaller parts. Gradient accumulation is what you do after micro-batching: sum up the gradients across those parts. Together, they allow you to simulate training with any batch size you want, within memory constraints.

Why It Matters

The quality of training often depends on batch size. If you can't fit a large batch directly, gradient accumulation ensures you still reap the benefits. It's one of those "engineering hacks" that makes training state-of-the-art models possible on limited resources.

Try It Yourself

1. Run training with batch size = 16 and no accumulation. Watch how noisy the loss curve looks.
2. Now set micro-batch size = 4 and accumulation_steps = 4. This simulates batch size = 16, but in smaller chunks. Compare the loss curve.
3. Increase accumulation_steps to simulate batch size = 32. Observe if training becomes smoother.
4. Experiment with turning accumulation off and on while keeping the same effective batch size. Notice how optimizer updates per epoch differ.
5. Print out how many times the optimizer is called. With accumulation, it should be fewer than the number of micro-batches.

The Takeaway

Gradient accumulation and micro-batching are techniques that let you train with large effective batch sizes while staying within the limits of your hardware. They preserve the benefits of large batches-stability and smoother gradients-without demanding extra memory. In `llm.c`, the simplicity of the training loop means you can clearly see where accumulation fits: gradients are summed across micro-batches, and only then does the optimizer step in. This is a small adjustment in code but a huge enabler in practice.

45. Logging and Progress Reporting

Every training loop needs a way to show what's happening under the hood. Without logs, you wouldn't know if the model is improving, if the code is running efficiently, or if something has silently gone wrong. In `train_gpt2.c`, logging is intentionally minimal but highly informative: each training step prints the step number, the current training loss, and how long that step took to run.

The Real Code for Logging

Here's the relevant snippet from `train_gpt2.c`:

```
// do a training step
clock_gettime(CLOCK_MONOTONIC, &start);
dataloader_next_batch(&train_loader);
gpt2_forward(&model, train_loader.inputs, train_loader.targets, B, T);
gpt2_zero_grad(&model);
gpt2_backward(&model);
gpt2_update(&model, 1e-4f, 0.9f, 0.999f, 1e-8f, 0.0f, step+1);
clock_gettime(CLOCK_MONOTONIC, &end);

double time_elapsed_s = (end.tv_sec - start.tv_sec) +
                        (end.tv_nsec - start.tv_nsec) / 1e9;
printf("step %d: train loss %f (took %f ms)\n",
       step, model.mean_loss, time_elapsed_s * 1000);
```

This small block accomplishes two things:

1. It measures how long the training step took using `clock_gettime`.
2. It reports the step number, the loss, and the elapsed time in milliseconds.

The output looks like this when training:

```
step 0: train loss 4.677779 (took 1987.546 ms)
step 1: train loss 5.191576 (took 1927.230 ms)
step 2: train loss 4.438685 (took 1902.987 ms)
```

Understanding What's Reported

- Step number (`step`) Tells you where you are in training. Since deep learning often runs for thousands of steps, this acts like a progress bar.
- Training loss (`model.mean_loss`) Shows how well the model is fitting the training batch. A lower value generally means better predictions. Watching this number decrease over time is the main signal that learning is happening.
- Step duration (`time_elapsed_s * 1000`) Measures performance. If one step takes 2000 ms, then 5000 steps would take about 3 hours. Monitoring this helps you estimate total training time and spot performance regressions (e.g., if a new change suddenly doubles the step time).

Why It Matters

Logs are your window into the training process. If the loss goes down smoothly, training is healthy. If it suddenly spikes or stays flat, something is wrong-maybe the learning rate is too high, or the model has run out of capacity. Timing information also matters: you need to know whether the code is running efficiently or wasting cycles.

Try It Yourself

1. Change the learning rate from `1e-4` to `1e-2` and watch how the loss behaves. If it jumps or becomes unstable, you'll see it directly in the logs.
2. Add validation logging by running the model on a held-out dataset every 100 steps and printing `val_loss`. Compare it to `train_loss`.
3. Record the log output to a file with:

   ```
   ./train_gpt2 > log.txt
   ```

   Then plot `train_loss` over steps in Python or Excel to visualize the curve.
4. Add throughput reporting: divide the batch size times sequence length (`B*T`) by the step time to print tokens per second. This gives a clearer sense of efficiency.
5. Try disabling `clock_gettime` and only print loss. Notice how much harder it becomes to judge performance without timing information.

The Takeaway

Even the simplest logs can tell you a lot. With just a single line-step, loss, and duration-you know how fast training is, whether it's converging, and how long it will take. In larger frameworks, this kind of information is often hidden behind dashboards and monitoring tools, but the core idea is the same: training is only useful if you can see and interpret its progress.

46. Validation Runs in the Training Loop

When you train a model, it is not enough to look only at how well it does on the training data. The real test is whether the model has learned patterns that apply to new, unseen data. This is where validation comes in. Validation is like a quiz the model takes from time to time during training. It does not count toward learning-it is just a check to see how much the model has really understood.

In `train_gpt2.c`, validation is built right into the training loop. Every so often, instead of updating weights, the program pauses and runs the model on a set of tokens it has never trained on. It then prints out the average validation loss. This number tells you if the model is actually generalizing, not just memorizing.

How the validation code looks

Here is the actual block of code that handles validation:

```
if (step % 10 == 0) {
    float val_loss = 0.0f;
    dataloader_reset(&val_loader);
    for (int i = 0; i < val_num_batches; i++) {
        dataloader_next_batch(&val_loader);
        gpt2_forward(&model, val_loader.inputs, val_loader.targets, B, T);
        val_loss += model.mean_loss;
    }
    val_loss /= val_num_batches;
    printf("val loss %f\n", val_loss);
}
```

At first glance, this might look like just a few lines of C code. But behind it are several important ideas about how machine learning models are tested while they learn. Let's go through this step by step.

Step-by-step explanation

The first line checks whether it is time to run validation:

```
if (step % 10 == 0) {
```

This means that validation happens every 10 steps. The `%` operator is "modulo," which returns the remainder of a division. If the step number is divisible by 10 (like 0, 10, 20, 30), then the block runs. By spacing it out this way, validation does not slow training too much but still gives you regular updates.

Next, the code sets up a place to store the running total of the validation loss:

```
float val_loss = 0.0f;
```

Then it resets the validation dataloader:

```
dataloader_reset(&val_loader);
```

This makes sure the validation dataset starts from the beginning each time. That way, the results are consistent-you're always checking the model on the same set of text, rather than starting from a random place.

Now comes the loop over validation batches:

```
for (int i = 0; i < val_num_batches; i++) {
    dataloader_next_batch(&val_loader);
    gpt2_forward(&model, val_loader.inputs, val_loader.targets, B, T);
    val_loss += model.mean_loss;
}
```

Here's what's happening inside:

- `dataloader_next_batch` fetches the next chunk of tokens and labels from the validation set.
- `gpt2_forward` runs the model forward on those tokens, predicting the next word for each one, and computes the loss against the true labels.
- The loss from that batch is added to `val_loss`.

Notice that there is no call to `gpt2_zero_grad`, no `gpt2_backward`, and no `gpt2_update`. That is because validation does not train the model. It only measures performance.

Finally, the program averages the loss across the number of batches:

```
val_loss /= val_num_batches;
```

And prints the result:

```
printf("val loss %f\n", val_loss);
```

This gives you a single number that summarizes how well the model is performing on unseen data at this point in training.

How to read validation loss

Imagine you are training and see logs like this:

```
step 0: train loss 4.677779 (took 1987.546 ms)
val loss 4.901234
step 1: train loss 5.191576 (took 1927.230 ms)
step 2: train loss 4.438685 (took 1902.987 ms)
...
step 10: train loss 3.912342 (took 1890.321 ms)
val loss 4.100321
```

The training loss is printed every step, while the validation loss appears every 10 steps. If both numbers are going down, that is a sign the model is genuinely learning. If training loss drops but validation loss stays the same or starts going up, the model is probably memorizing the training set-this is called overfitting.

Why validation is important

Without validation, you could be tricked into thinking the model is improving just because the training loss is going down. But that might only mean it has memorized the training data. Validation checks prevent this by showing you whether the model can handle data it has not seen before. It is like a student practicing with old exam papers (training) versus being tested with new problems (validation).

Small details that matter

The code averages validation loss over `val_num_batches`, which is set earlier to 5. That means it only checks 5 batches, not the entire validation dataset. This is a shortcut-it makes validation much faster, at the cost of some accuracy in the measurement. But for training feedback, this is usually enough.

The batch size `B` and sequence length `T` for validation are the same as training. This keeps the loss comparable between training and validation.

Try it yourself

You can experiment with the validation process to understand it better. Here are some ideas:

1. Change the frequency from every 10 steps to every 5 or even every step. You'll see more validation updates, but training will slow down.
2. Increase `val_num_batches` to 20. The validation loss will become less noisy, but each check will take longer.
3. Comment out the validation block and train again. Notice how you lose a sense of whether the model is really generalizing.
4. Save validation loss values to a file and plot them. Compare the curve against the training loss curve. You'll see how they move together or diverge.
5. Try using a very small validation dataset. Watch how the loss jumps around more compared to a larger, more stable dataset.

The takeaway

Validation runs are short forward-only tests that give you confidence the model is learning patterns that apply to new text. They are easy to implement-a few lines of code in `train_gpt2.c`-but they are one of the most important tools for monitoring training. By checking validation loss regularly, you make sure your model is not just memorizing but actually becoming better at language modeling.

47. Checkpointing Parameters and Optimizer State

Training a model can take hours, days, or even weeks. If you stop the program halfway-whether by accident (a crash, a power cut) or on purpose (pausing to save compute)-you don't want to start over from scratch. Checkpointing solves this problem by saving the model's parameters and optimizer state to disk so you can resume training later.

What a checkpoint contains

A checkpoint is like a "save game" for machine learning. At a minimum, it needs:

1. Model parameters – the actual weights of the neural network, stored as floating-point numbers in memory. These define what the model has learned so far.
2. Optimizer state – for AdamW, this includes the running averages of gradients (`m_memory`) and squared gradients (`v_memory`). Without these, the optimizer would lose its "memory" of past updates, which could destabilize resumed training.
3. Step counter – the number of steps completed so far. This matters for bias correction in AdamW and for scheduling the learning rate.

Together, these three things capture the full training state.

Saving a checkpoint

Although `train_gpt2.c` is kept minimal and does not include full checkpointing code, the idea is straightforward. You allocate a file, write all parameters, optimizer buffers, and metadata, then close the file. In pseudocode, it looks like this:

```
FILE *f = fopen("checkpoint.bin", "wb");
fwrite(model.params_memory, sizeof(float), model.num_parameters, f);
fwrite(model.m_memory, sizeof(float), model.num_parameters, f);
fwrite(model.v_memory, sizeof(float), model.num_parameters, f);
fwrite(&step, sizeof(int), 1, f);
fclose(f);
```

This is a binary dump of the model and optimizer. Later, you can load the file back with `fread` calls into the same memory locations.

Loading a checkpoint

Loading is the reverse:

```
FILE *f = fopen("checkpoint.bin", "rb");
fread(model.params_memory, sizeof(float), model.num_parameters, f);
fread(model.m_memory, sizeof(float), model.num_parameters, f);
fread(model.v_memory, sizeof(float), model.num_parameters, f);
fread(&step, sizeof(int), 1, f);
fclose(f);
```

Once loaded, training can continue exactly where it left off.

Why optimizer state matters

It might seem enough to save only the model's parameters. But AdamW depends on moving averages of past gradients. If you throw those away and restart with only the parameters, the optimizer will behave differently. Learning may suddenly become unstable, or the effective learning rate may feel wrong. That's why saving both the parameters and optimizer state gives the most faithful restart.

Why checkpointing is essential

Training is rarely smooth. Servers reboot, experiments are interrupted, bugs are found. Without checkpoints, any interruption means wasted compute and lost progress. With checkpoints, you can pause and resume at will. They also let you archive important moments in training-for example, saving the model when validation loss is lowest, not just at the end.

Try it yourself

1. Write a small function that saves the model's parameters after every 100 steps. Then kill the program midway and reload from the saved file. Confirm that resumed training picks up where it left off.
2. Try saving only parameters but not optimizer state. Resume training and compare loss curves. You'll see that the run diverges from the original.
3. Save checkpoints at multiple steps and later reload them to compare model generations (does the model produce more fluent text after 10 steps, 100 steps, 1000 steps?).
4. Intentionally corrupt part of a checkpoint file (flip a few bytes) and try reloading. This helps you understand why consistency checks or checksums are often added in real systems.
5. Store checkpoints in a versioned way (e.g., `checkpoint_step100.bin`, `checkpoint_step200.bin`) so you can roll back if a later training phase degrades performance.

The takeaway

Checkpointing is what makes long-running training practical. By saving parameters, optimizer state, and the step counter, you preserve not just what the model knows but how it is learning. In real projects, checkpoints are the bridge between experiments and production: they let you stop, resume, compare, and deploy models without ever starting from scratch. Even though `llm.c` does not fully implement it, the concept is simple and invaluable.

48. Reproducibility and Small Divergences

When training deep learning models, two runs that look identical on the surface can still behave differently. One run might converge quickly, another might take longer, and sometimes losses diverge even though you used the same dataset and code. This happens because of the way randomness and numerical precision interact during training. Reproducibility is about controlling these factors so that results are consistent and meaningful.

Sources of randomness

There are several places where randomness sneaks into training:

- Data order: if batches are shuffled differently, the model sees tokens in a new sequence. Early steps can influence the trajectory of training.
- Weight initialization: initial parameters are usually set randomly. Different seeds lead to slightly different starting points.
- Dropout and sampling: while `train_gpt2.c` is minimal and doesn't include dropout layers, many neural networks do. Dropout randomly disables activations during training.
- Floating-point arithmetic: on CPUs and GPUs, the order of summations or parallel reductions can cause tiny rounding differences. Over many steps, these small changes accumulate.

How llm.c handles reproducibility

The repository includes functions like `manual_seed` and `random_f32` in `llmc/rand.h`. These are simple random number generators that can be seeded with a fixed value. For example:

```
manual_seed(1337);
```

If you call this before training, the random number generator starts from the same state every run. That means weight initialization and sampling will be reproducible.

The dataloaders also have reproducibility options. When you initialize a `DataLoader`, you can decide whether it shuffles batches or not. Keeping this consistent ensures the model sees the same data order each run.

Why small divergences happen anyway

Even with fixed seeds, you might notice that two runs are not perfectly identical. On CPUs, differences often come from OpenMP parallel loops-threads may sum numbers in a different order, producing slightly different results. On GPUs, parallelism and library implementations (like cuBLAS or cuDNN) can do the same.

These differences are usually very small, but deep learning systems are chaotic: tiny changes in the early steps can grow into visible differences later. This doesn't mean the code is wrong-it just means floating-point math has limits.

Why reproducibility matters

Reproducibility isn't just about peace of mind. It has real uses:

- Debugging: if a bug appears, you want to reproduce the exact same run to diagnose it.
- Comparisons: when testing new optimizers, schedulers, or architectures, you want fair comparisons on identical conditions.
- Science: reproducible results are essential for research papers and benchmarks.

At the same time, absolute bit-for-bit reproducibility is often unrealistic in large parallel systems. Instead, the goal is practical reproducibility: ensuring that runs are *similar enough* to reach the same conclusions.

Example experiment

Suppose you seed training with `manual_seed(1337)` and use the same dataset. You might get a loss curve like this:

```
Run A: step 1000 → val loss 3.42
Run B: step 1000 → val loss 3.43
```

The numbers are not identical, but they are close. The important part is that the model's learning trajectory is stable and results are comparable.

If you remove the seed and allow full randomness, you might get:

```
Run A: step 1000 → val loss 3.42
Run B: step 1000 → val loss 3.89
```

Both are valid, but harder to compare.

Try it yourself

1. Run training twice without setting a seed. Compare how training loss and validation loss differ at step 500.
2. Set a fixed seed with `manual_seed(42)` before building the model. Run training twice and compare again. You should see closer numbers.
3. Enable OpenMP with multiple threads and then run with a single thread. Notice how results differ slightly due to floating-point summation order.
4. Save two checkpoints from runs with different seeds. Use the model to generate text and compare outputs. You'll see different wording, but both grammatically plausible.
5. Increase the dataset size and check if differences between runs shrink. With more data, randomness matters less.

The takeaway

Reproducibility in training is about controlling randomness where possible and accepting small divergences where not. In `llm.c`, reproducibility is made clear through simple seeding functions and deterministic dataloader options. Perfect bit-level reproducibility isn't the point-the goal is to ensure results are stable, comparable, and scientifically sound, even if tiny numerical differences creep in.

49. Command-Line Flags and Defaults

When you run a training program, you often want to change certain settings without editing the source code. For example, you might want to try a different batch size, adjust the learning rate, or train for more steps. Command-line flags make this possible. In `train_gpt2.c`, defaults are set inside the program, but it can also be compiled to accept arguments, giving you flexibility while keeping the code minimal.

Why flags exist

Deep learning experiments are highly sensitive to hyperparameters-values like learning rate, batch size, sequence length, or number of steps. If every change required modifying source code, recompiling, and rerunning, experimentation would be slow and error-prone. Flags allow you to configure these parameters quickly at runtime.

In many large frameworks (like PyTorch or TensorFlow), command-line arguments are parsed with helper libraries. In `llm.c`, the philosophy is simplicity: flags are either defined in code as constants, or you can extend `main` with standard C argument parsing to override defaults.

Defaults in `train_gpt2.c`

Looking at the code, here are the main defaults hardcoded in the `main` function:

- Batch size (`B`):

  ```
  int B = 4; // number of sequences per batch
  ```

- Sequence length (`T`):

  ```
  int T = 64; // tokens per sequence
  ```

- Validation batches:

  ```
  int val_num_batches = 5;
  ```

- Training steps:

  ```
  for (int step = 0; step <= 40; step++) {
  ```

  By default, only 40 steps are run in this example.

- Optimizer hyperparameters (inside `gpt2_update` call):

  ```
  gpt2_update(&model, 1e-4f, 0.9f, 0.999f, 1e-8f, 0.0f, step+1);
  ```

  Here the learning rate is `1e-4`, beta values for AdamW are `0.9` and `0.999`, epsilon is `1e-8`, and weight decay is `0.0`.

These defaults are chosen to make the reference training loop run quickly and predictably, especially on small datasets like Tiny Shakespeare or Tiny Stories.

How to add flags

If you want flexibility, you can extend `main` with argument parsing:

```
int main(int argc, char argv) {
    int B = 4;
    int T = 64;
    int max_steps = 40;
    if (argc > 1) B = atoi(argv[1]);
    if (argc > 2) T = atoi(argv[2]);
    if (argc > 3) max_steps = atoi(argv[3]);
    ...
}
```

Now you can run:

```
./train_gpt2 8 128 100
```

This sets batch size to 8, sequence length to 128, and steps to 100, without changing source code.

Why it matters

Command-line flags make experimentation far more efficient. You can try multiple configurations in one day without recompiling or editing the file repeatedly. This is especially useful when running jobs on clusters where you want scripts that launch many experiments automatically with different parameters.

Defaults are equally important: they give you a safe, predictable starting point. Beginners can run the code without thinking about flags, while advanced users can override values as needed.

Try it yourself

1. Keep the default batch size of 4 and sequence length of 64. Run training and note the time per step.
2. Change batch size to 8 by editing the code. Observe how training speed changes and how memory usage increases.
3. Modify the loop to train for 200 steps instead of 40. Watch how loss decreases further.
4. Add argument parsing to accept learning rate as a flag. Experiment with `1e-3` vs. `1e-5` and see how quickly training diverges or stalls.
5. Create a shell script that runs training multiple times with different values for `B` and `T`. Compare results.

The takeaway

Command-line flags and defaults balance simplicity with flexibility. Defaults make the code runnable out of the box, while flags let you scale experiments without constantly editing the source. In `train_gpt2.c`, this design keeps the training loop minimal but still adaptable, encouraging both clarity and experimentation.

50. Example CPU Training Logs and Outputs

One of the best ways to understand what a training loop is doing is by reading its logs. Logs are the program's way of telling you how training is progressing: what the loss is, how fast it's running, and whether validation checks are improving. In `train_gpt2.c`, logging is deliberately minimal so you can easily see the essentials without being overwhelmed.

What the logs look like

Here's a snippet of output from running the CPU training loop on Tiny Shakespeare:

```
train dataset num_batches: 1192
val dataset num_batches: 128
[GPT-2]
max_seq_len: 1024
vocab_size: 50257
padded_vocab_size: 50304
num_layers: 12
num_heads: 12
channels: 768
num_parameters: 124475904
num_activations: 73347840
val loss 5.325529
step 0: train loss 4.677779 (took 1987.546000 ms)
step 1: train loss 5.191576 (took 1927.230000 ms)
step 2: train loss 4.438685 (took 1902.987000 ms)
...
```

Each part of this output has meaning:

- Dataset sizes: how many training and validation batches are available.
- Model config: confirmation that the GPT-2 model was loaded correctly (sequence length, vocab size, number of layers, etc.).
- Validation loss: an average measure of how well the model is doing on unseen data.
- Training step logs: for each step, you see the training loss and how long the step took in milliseconds.

Understanding loss values

Loss is the number that tells us how far the model's predictions are from the correct answers. Lower is better.

- A loss around 5.3 means the model is essentially guessing.
- As training progresses, you want to see this number slowly decrease.
- If the number gets stuck, or goes up, it can indicate problems with the learning rate, dataset, or implementation.

Think of it like a report card: at the beginning, the model is failing every test, but as it practices (trains), the grades (loss values) improve.

Speed measurements

The "took … ms" part shows how long each step took. On CPU, this is usually slow, sometimes a couple of seconds per step. On GPU, the same step might only take tens of milliseconds.

Timing logs are useful because they help you:

- Estimate how long full training will take.
- Compare performance between machines.
- Spot problems if training suddenly slows down.

Occasional validation checks

Every few steps, the code switches to validation data and prints a `val loss`. This is crucial: training loss always goes down if the model memorizes the training set, but validation loss tells you if it is *actually learning patterns* that generalize.

If training loss goes down but validation loss stays high, that's a sign of overfitting.

Generated samples

At certain steps, the code also prints generated text like this:

```
generating:

 The King had not
 that the Duke of Northumberland and the Duke of
...
```

Even though the text might look strange at first, it's a powerful sign that the model is learning. At the beginning, output is pure gibberish, but as training continues, you start to see recognizable words and patterns.

Why it matters

Logs are your window into the training process. Without them, training would be a black box-you'd wait hours and have no idea if it was working. By watching loss curves, step times, and sample outputs, you can make informed adjustments and gain confidence that the model is on the right track.

Try it yourself

1. Run the training loop as-is and save the console output. Mark how loss changes between step 0 and step 40.
2. Increase the number of steps to 200 and compare how the losses evolve.
3. Change the batch size from 4 to 8 and note both the training speed and the loss behavior.
4. Edit the code to print validation loss every step instead of every 10 steps. Does the trend look smoother?
5. Save the generated samples at steps 20 and 40. Compare how the quality changes.

The takeaway

Training logs are like a diary of the model's progress. They show you how quickly the model is learning, how well it generalizes, and how fast the computation runs. By reading and interpreting logs carefully, you can guide experiments, detect problems early, and appreciate the progress that's happening inside the model.

Chapter 6. Testing and Profiling

51. Debug State Structs and Their Role

When building and training a model as complex as GPT-2, you need ways to peek inside and check whether the values being passed around make sense. This is where debug state structs come in. In *llm.c*, the code is written in plain C, without the rich debugging utilities of frameworks like PyTorch. That means the developers had to create their own mechanism to store, inspect, and compare intermediate values.

A struct in C is just a container that groups related variables together. For debugging, you can think of a struct as a little notebook where the program writes down numbers as it computes them. These numbers might include:

- The raw embeddings for tokens.
- The attention scores before and after softmax.
- The outputs of each MLP block.

- The predicted probabilities for the next token.

By saving these into a structured format, the program can later compare them to the outputs of a trusted reference implementation (usually PyTorch).

How it works in practice

Inside *llm.c*, there are places where arrays of floats—like hidden states or logits—are copied into a debug state struct. Once stored, these values can be printed, dumped to a file, or checked against "golden" results from PyTorch.

Imagine you're testing a tiny batch of input tokens. The forward pass runs as usual, but at specific checkpoints (say, right after attention or after the final linear projection), the program writes those arrays into a struct. Later, when running a PyTorch model with the same inputs and weights, the two outputs can be compared element by element.

This is essential for catching subtle errors:

- A misplaced transpose in matrix multiplication.
- Forgetting to apply a mask in attention.
- A floating-point precision mismatch in softmax.

Without the struct, you'd only know the model loss looks "off." With the struct, you know *exactly* which step went wrong.

Why it matters

Debug structs bridge the gap between C and Python ecosystems. PyTorch has a decade of battle-tested layers, so it's the gold standard for correctness. By saving intermediate activations in C and comparing them against PyTorch, developers ensure that every layer behaves identically. This builds confidence that the *llm.c* codebase isn't just "roughly correct," but precisely reproduces GPT-2's math.

For anyone modifying the code—for example, writing a new activation function or experimenting with quantization—the debug structs act like a safety net. You can quickly see if your change accidentally altered the outputs in a way that breaks parity with the original model.

Try it yourself

1. Trace a forward pass: Run the CPU version with a tiny batch and enable debug dumps. Look at the embeddings, attention scores, and final logits.
2. Cross-check with PyTorch: Run the same input through Hugging Face GPT-2. Print the same tensors. Compare a few entries by hand—do they match closely?
3. Introduce a bug: Change the scaling factor in attention (e.g., remove the `1/√d` term). Run again and see how quickly the mismatch shows up in the debug struct.
4. Extend the struct: Add a new field for an intermediate step you care about, like LayerNorm outputs. Print it during debugging to see how normalization changes the activations.

The takeaway

Debug state structs are the microscope of *llm.c*. They allow you to pause the flow of numbers, record them, and compare them against a known-good model. Without them, development would feel like working blindfolded. With them, you can track down errors precisely, ensure parity with PyTorch, and confidently extend the system knowing you have a reliable safety net.

52. `test_gpt2.c`: CPU vs PyTorch

Testing is one of the most important parts of the *llm.c* project. The file `test_gpt2.c` exists specifically to check whether the C implementation of GPT-2 produces the same outputs as PyTorch, which is the trusted reference. Without this file, you would only know if the final training loss looked reasonable. With it, you can verify that every part of the forward pass matches.

At its core, `test_gpt2.c` runs a very controlled experiment: it loads a GPT-2 model checkpoint (exported from PyTorch), prepares a small batch of input tokens, executes a forward pass in C, and compares the outputs against the corresponding tensors from PyTorch. If everything matches within a tight numerical tolerance, you know the C code is correct. If not, you have a clear signal that something is wrong.

How the test works

1. Load the checkpoint The test begins by reading a binary checkpoint file, such as `gpt2_124M.bin`, which contains the weights of the GPT-2 model. These weights were originally trained in PyTorch and then exported into a binary format that *llm.c* can understand.

2. Prepare the inputs The test uses a known sequence of token IDs—sometimes from a dataset like Tiny Shakespeare, sometimes just a few hand-picked tokens. This ensures the same inputs can be run through both PyTorch and C implementations.
3. Run the C forward pass The function `gpt2_forward` is executed on the CPU. All embeddings, attention layers, MLPs, and final logits are computed exactly as they would be during real inference or training.
4. Compare with PyTorch For each major tensor (e.g., hidden states, attention outputs, final logits), the values are compared against saved outputs from PyTorch. The comparison usually allows for very small differences, since floating-point math can vary slightly between libraries. A tolerance like `1e-5` is common.
5. Report mismatches If any element deviates beyond the allowed tolerance, the test reports the difference. Developers can then investigate where the divergence started, often by adding more debug dumps of intermediate states.

Why this test is crucial

C code is low-level and unforgiving. A single indexing mistake, a wrong stride, or a missing scaling factor in attention can make outputs diverge wildly. Since GPT-2 has millions of parameters, such errors are almost impossible to spot by hand. By tying the implementation back to PyTorch, `test_gpt2.c` provides a ground truth check.

This also ensures scientific reproducibility. If someone else downloads *llm.c* and runs `test_gpt2.c`, they should see the same level of agreement with PyTorch. That way, they can trust that training runs, losses, and model outputs are not artifacts of a broken implementation.

Example in action

Imagine you've just modified the attention code to optimize the matrix multiplication. You recompile and run `test_gpt2.c`. If you see an error like:

```
Mismatch at position [0, 12, 42]: C=0.1234, Torch=0.1235
```

you know the two match within tolerance—everything is fine. But if you see:

```
Mismatch at position [0, 12, 42]: C=0.3456, Torch=0.1235
```

that's a red flag. It means the optimization introduced a bug. Without the test, you might not notice until much later when training fails to converge.

Why it matters

`test_gpt2.c` is the guarantee that the C implementation is not just "close enough," but *faithful*. It ensures that improvements, optimizations, or even rewrites don't silently corrupt the model's behavior. It's a direct bridge between the experimental world of C internals and the well-established baseline of PyTorch.

Try it yourself

1. Run `make test_gpt2` in the repository. Observe whether the outputs match PyTorch.
2. Deliberately change one line of code in `gpt2_forward`—for example, remove the attention scaling factor. Run the test again and see how quickly it fails.
3. Add your own print statements to show which tensors are being compared. Watch how the numbers line up almost exactly.
4. Try running with different checkpoints (e.g., 124M vs 355M) to see if parity holds across scales.

The takeaway

`test_gpt2.c` is not just another file in the repository—it's the truth meter. It reassures you that the complicated layers of GPT-2 have been implemented correctly in C and remain consistent with PyTorch. This confidence is what allows further work—whether training, profiling, or extending the model—to proceed on a solid foundation.

53. Matching Outputs Within Tolerances

Once you have a test like `test_gpt2.c` set up, the next challenge is figuring out how close the outputs need to be for the test to pass. Computers don't always produce bit-for-bit identical results when doing floating-point math. The order of operations, the precision of instructions, and even the type of hardware (CPU vs GPU) can cause tiny differences.

If you demanded exact matches, most tests would fail even when the implementation is correct. That's why *llm.c* uses tolerance-based comparison. Instead of asking "are these numbers exactly equal?", the code asks "are these numbers *close enough*?"

Absolute vs relative tolerance

There are two common ways to define "close enough":

- Absolute tolerance: check that the difference between two numbers is smaller than a threshold. For example,

`|0.123456 - 0.123455| = 0.000001 < 1e-5`

This works well for values near zero.

- Relative tolerance: check that the difference is small relative to the size of the numbers. For example,

  `|1000.0 - 1000.1| / 1000.0 = 0.0001`

  Even though the absolute difference is 0.1, that's tiny compared to the scale of 1000.

In practice, the code often combines both. It passes if the difference is smaller than either the absolute tolerance or the relative tolerance.

Why tolerances are necessary

Imagine you run the forward pass on CPU in C and in PyTorch. PyTorch might use fused kernels or higher-precision accumulations. If you compare the final logits, you may see values like:

- PyTorch: `-3.4521234`
- C: `-3.4521255`

The difference is just `0.0000021`. For practical purposes, they're the same. Without tolerance, this tiny difference would fail the test. With tolerance, you can safely say both implementations agree.

Example from debugging

Suppose you compare the probabilities after softmax. You might get:

- PyTorch: `0.3333333, 0.3333333, 0.3333333`
- C: `0.3333334, 0.3333333, 0.3333333`

Here, the first value differs in the last decimal place. The tolerance rule says that's fine, since the absolute error is smaller than `1e-7`.

But if you saw something like:

- PyTorch: `0.3333333, 0.3333333, 0.3333333`
- C: `0.5000000, 0.2500000, 0.2500000`

the mismatch is huge—no tolerance rule would allow it. That's a clear bug.

Why it matters

Matching within tolerance isn't just a technical detail; it's about trust. It lets you say with confidence that the implementation is mathematically faithful to the reference. You don't waste time chasing harmless decimal noise, but you also don't miss real mistakes.

This approach is also what makes cross-platform development possible. The same *llm.c* code can be run on Linux, macOS, or even inside different compilers, and as long as results fall within tolerance, you know the model behavior is preserved.

Try it yourself

1. Run `test_gpt2.c` and look at the output logs. Notice how many decimal places match.
2. Change the tolerance threshold in the code from `1e-5` to something stricter like `1e-8`. See if the test starts failing due to harmless floating-point noise.
3. Add a deliberate bug—for example, skip dividing by `√d` in the attention code—and rerun. The mismatches will be far larger than the tolerance, proving the bug is real.
4. Compare CPU results with PyTorch, then recompile with different compiler flags (like `-O0` vs `-O3`) and check if results still fall within tolerance.

The takeaway

Tolerance-based testing is what allows *llm.c* to be both rigorous and realistic. It ensures that differences are only flagged when they matter, while ignoring the harmless quirks of floating-point math. This makes the test suite a reliable tool for catching true errors without overwhelming you with false alarms.

54. Profiling with `profile_gpt2.c`

After verifying that the outputs match PyTorch, the next big question is: how fast is the code running? Correctness is essential, but performance is what makes a minimal C implementation like *llm.c* worthwhile. That's where `profile_gpt2.c` comes in. It is a small program that runs controlled forward passes through GPT-2 and measures the time they take, helping you understand where the bottlenecks are.

What profiling means

Profiling is the act of measuring the performance of a program, not just its correctness. Instead of asking "does this number match PyTorch?", profiling asks:

- How many milliseconds does one forward pass take?

- Which part of the model consumes the most time?
- Does using OpenMP threads actually speed things up?
- How does batch size affect runtime?

By answering these, you can make informed decisions about optimization.

How `profile_gpt2.c` works

The profiling program is structured like a simplified inference loop.

1. Model setup It loads a GPT-2 checkpoint (e.g., `gpt2_124M.bin`) into memory and allocates space for activations.
2. Dummy input Instead of using real text, it creates random token IDs. That way, the cost measured comes purely from the computation, not from data loading or tokenization.
3. Timing with clock functions Before and after each forward pass, it records timestamps with `clock_gettime(CLOCK_MONOTONIC, &start)` and `&end`. The difference gives the runtime in seconds, which is usually converted into milliseconds.
4. Looping for stability A single run can be noisy due to background processes on your computer. To smooth things out, `profile_gpt2.c` runs the forward pass multiple times and averages the results.
5. Reporting results Finally, it prints the average time per forward pass. Sometimes it also estimates FLOPs (floating-point operations per second), giving you a rough idea of efficiency compared to the theoretical peak of your CPU.

What you can learn from profiling

Running `profile_gpt2.c` on a CPU gives insights like:

- The attention blocks dominate runtime, because they involve large matrix multiplications.
- Increasing batch size makes the runtime longer, but not proportionally—sometimes bigger batches use hardware more efficiently.
- OpenMP can speed things up when there are multiple CPU cores available, but scaling may flatten out after a certain number of threads.

This helps decide where to spend effort. For example, if LayerNorm takes 2% of the runtime but attention takes 70%, you know optimization should focus on the attention code.

Why it matters

Profiling isn't just about numbers. It's about guiding your development choices. Without profiling, you might spend weeks hand-optimizing LayerNorm, only to discover it barely affects overall runtime. With profiling, you see immediately where the slowdowns are and can focus on the real bottlenecks.

It also provides baseline performance numbers. If you change something in the implementation, re-running `profile_gpt2.c` will tell you if it sped things up or slowed them down. This feedback loop is essential for optimization work.

Try it yourself

1. Compile and run `profile_gpt2.c` with a small model checkpoint. Note the reported runtime per forward pass.
2. Change the batch size `B` and sequence length `T`, then re-run. Watch how runtime scales with larger inputs.
3. Set `OMP_NUM_THREADS=1` to disable threading and compare it against `OMP_NUM_THREADS=4` or higher. How much faster is it with multiple cores?
4. Modify the code to time individual layers (embeddings, attention, MLP). This gives even more precise insight into which parts dominate computation.

The takeaway

`profile_gpt2.c` turns performance from a guess into hard data. It tells you exactly how long a forward pass takes, how much threading helps, and where the real bottlenecks are. With it, you can track progress as you optimize the code, ensuring that your changes make the model not only correct but also efficient.

55. Measuring FLOPs and CPU Performance

When talking about performance in deep learning, it's not enough to say "this run took 200 milliseconds." To really understand efficiency, we need a measure that's independent of hardware and input size. That's where FLOPs come in: floating-point operations. A FLOP is a single numerical calculation involving real numbers—like an addition, multiplication, or division.

By counting how many FLOPs a model requires and comparing it to how many the computer can perform per second, you can evaluate how close your implementation is to the theoretical maximum speed of your CPU.

What FLOPs mean in practice

Every layer of GPT-2—embeddings, attention, feed-forward—can be broken down into a sequence of multiplications and additions. For example:

- A matrix multiplication between two matrices of size `M × K` and `K × N` requires `2 × M × K × N` FLOPs.
- A softmax across a vector of size `d` requires about `3d` FLOPs (exponentials, sums, and divisions).
- An MLP block with hidden size `h` and intermediate size `4h` requires multiple matrix multiplications, adding up to billions of FLOPs per training step.

When you sum these across all layers, even a small GPT-2 model like 124M parameters involves several gigaFLOPs (billions of operations) for one forward pass.

How `profile_gpt2.c` estimates FLOPs

The profiling code doesn't literally count every multiplication. Instead, it uses formulas derived from matrix dimensions:

1. The configuration struct (`model.config`) gives the number of layers, heads, embedding size, and sequence length.
2. For each block (attention + MLP), the code applies standard FLOPs formulas for matrix multiplication and softmax.
3. These counts are added up to estimate the total FLOPs for a forward pass.
4. Dividing the total FLOPs by the measured runtime (in seconds) gives FLOPs/second, also known as throughput.

For example, if a forward pass takes 0.1 seconds and involves 20 billion FLOPs, then throughput is about 200 GFLOPs/s.

Why this is useful

1. Compare hardware: You can test the same model on a laptop CPU and a server CPU, then compare FLOPs/s to see how much faster the server is.
2. Compare implementations: If you modify attention to use a different algorithm, the FLOPs count won't change, but if runtime decreases, throughput increases—showing the optimization worked.
3. Know your limits: CPUs often achieve only a fraction of their theoretical peak FLOPs due to memory bottlenecks. Profiling shows how close you're getting in practice.

Example

Let's say:

- Forward pass FLOPs: 15 billion
- Runtime: 0.2 seconds
- Throughput: 75 GFLOPs/s

If your CPU's datasheet says the peak is 200 GFLOPs/s, then you're at about 37% efficiency. That gap might be due to memory latency, cache misses, or lack of vectorization.

Try it yourself

1. Run `profile_gpt2.c` and note the reported FLOPs and runtime.
2. Change the sequence length `T` and observe how FLOPs scale linearly with it.
3. Increase the number of layers in the model configuration—watch FLOPs rise accordingly.
4. Compare your measured FLOPs/s with the theoretical maximum listed for your CPU. How close are you?

The takeaway

FLOPs turn performance from "this feels fast" into hard numbers you can compare across runs, machines, and implementations. By knowing both the operation count and the achieved throughput, you gain a clear picture of how efficient the *llm.c* code really is and where further optimizations might pay off.

56. Capturing Memory Usage on CPU

While FLOPs tell us how much raw computation a model needs, performance isn't just about speed—it's also about memory usage. Modern language models are enormous, and on CPUs with limited RAM, memory can become the true bottleneck. That's why *llm.c* also emphasizes monitoring how much memory is used during inference and training.

What consumes memory in GPT-2

There are several key components that take up space:

1. Parameters (weights): GPT-2 124M has about 124 million parameters. Each is stored as a 32-bit float (4 bytes). That alone is roughly 500 MB.

2. Gradients: During training, gradients for each parameter are stored. That doubles the memory usage.
3. Optimizer states: AdamW requires two additional memory slots per parameter (`m` and `v`), which doubles it again. With parameters, gradients, and optimizer states combined, training can require 4× the parameter size in memory.
4. Activations: These are the intermediate outputs of each layer (attention scores, MLP results, normalized states). For backpropagation, activations from the forward pass must be kept until gradients are computed. Depending on batch size and sequence length, activations can rival parameter memory.

Measuring memory in practice

On CPU, memory usage can be inspected in several ways:

- Operating system tools: `top`, `htop`, or Activity Monitor show total memory used by the program.
- Manual accounting in code: *llm.c* knows how many parameters, gradients, and optimizer states exist. By multiplying their counts by 4 bytes, it can estimate usage precisely.
- Instrumentation during profiling: you can add checkpoints that print memory usage at different stages of the forward or backward pass.

For example:

- Parameters only: ~500 MB.
- Parameters + gradients: ~1 GB.
- Parameters + gradients + optimizer: ~2 GB.
- Adding activations: 2.5–3 GB, depending on batch size and sequence length.

Why memory matters on CPU

On a CPU, you don't just care about "can it fit in RAM?" You also care about cache efficiency. Modern CPUs have multiple levels of cache (L1, L2, L3), which are much faster than main RAM. If activations or weights don't fit well in cache, performance can suffer even if you technically have enough RAM.

Memory footprint also limits experiment flexibility. For example, increasing sequence length from 64 to 1024 multiplies activation storage by 16. A run that fits at `T=64` may crash or swap at `T=1024`.

Example scenario

Suppose you run GPT-2 124M with:

- Batch size `B=4`
- Sequence length `T=64`

This might use ~2.5 GB of memory for training. If you raise `T` to 512, suddenly activations balloon, and total usage may exceed 10 GB. On a laptop with 8 GB RAM, this simply won't work.

By monitoring memory carefully, you can avoid mysterious crashes and plan runs realistically.

Try it yourself

1. Run training with a small batch (`B=2`, `T=64`) and check memory usage with `htop`.
2. Increase `T` step by step (128, 256, 512) and record the growth. Watch how activations dominate beyond a certain length.
3. Calculate parameter memory manually: `num_params × 4 bytes`. Compare it to what the OS reports. The difference comes from activations and optimizer states.
4. Modify the code to print memory allocations explicitly when arrays are created. This gives an internal log of usage at each step.

The takeaway

Memory is the silent partner of FLOPs: you need both to train and run models efficiently. Profiling without tracking memory is incomplete—you might have a model that's fast but impossible to run on your machine. By capturing and understanding memory usage, you gain the ability to scale responsibly, balance batch size and sequence length, and keep your experiments stable.

57. Reproducing Known Loss Curves (CPU-only)

Once the model is correct and its performance is measured, the next important step is to check whether it learns in the way we expect. In deep learning, we usually monitor this with a loss curve—a graph that shows how the training loss decreases over time as the model sees more data.

For GPT-2 and other language models, the standard loss function is cross-entropy, which measures how well the predicted probability distribution over tokens matches the actual next token in the dataset. If the implementation is right, the loss should fall in a predictable way when trained on text like Tiny Shakespeare or Tiny Stories.

What a "known" loss curve looks like

The community has already run countless GPT-2 experiments in PyTorch, so we know roughly what the curve should look like:

- At the very beginning, the loss is high (around 5–6) because the model is basically guessing.
- After a few hundred steps, the loss starts dropping steadily.
- For Tiny Shakespeare, loss often goes down toward ~2.0 with a small GPT-2 model (124M parameters).
- The exact numbers can vary, but the shape—a downward trend with small fluctuations—is consistent.

If the C implementation produces a similar curve, that's a strong sign the forward pass, backward pass, and optimizer are all working correctly.

How reproduction is tested in practice

1. Train with a small dataset The test usually uses Tiny Shakespeare or Tiny Stories since they are small enough to run quickly on CPU.
2. Log the loss per step Each training step prints something like:

   ```
   step 0: train loss 4.87
   step 10: train loss 4.12
   step 20: train loss 3.75
   ...
   ```

3. Plot the curve Save the loss values and make a simple plot with step on the x-axis and loss on the y-axis.
4. Compare against PyTorch If you train the same model in PyTorch with the same hyperparameters, the loss curve should look almost identical. Small differences are normal due to random seeds or floating-point math.

Why this is important

Reproducing a known loss curve is more than just a sanity check. It tells you:

- The math is right: gradients, optimizer updates, and scheduler logic are functioning.
- The data pipeline is correct: tokens are being fed in properly and batches are consistent.
- Nothing is silently broken: without this, a bug might go unnoticed until much later in training.

This is especially important on CPU, because training is slower and you may only run a few hundred steps. If the curve starts to dip in the expected way, you know you're on the right track.

Example scenario

Suppose you train GPT-2 124M with `B=4` and `T=64` on Tiny Shakespeare. The loss starts around 4.9, and by step 200 it falls to around 3.2. If PyTorch shows a similar trajectory, then your implementation is validated.

But if your loss stays flat, say at 5.0 for hundreds of steps, that's a red flag. It could mean gradients are not flowing, the optimizer isn't updating weights, or your data loader is feeding the same batch repeatedly.

Try it yourself

1. Train for 200 steps on Tiny Shakespeare with the C code and save the printed losses.
2. Train the same setup in PyTorch. Plot both curves together and compare.
3. Intentionally break something—for example, comment out the optimizer update step—and observe how the loss no longer decreases.
4. Experiment with learning rates. Too high may cause the curve to bounce up and down, while too low will make it drop very slowly.

The takeaway

Reproducing known loss curves is the ultimate integration test. It proves that the entire pipeline—data, model, training loop, optimizer—works together in harmony. When your loss curve matches the reference, you can trust that your C implementation of GPT-2 is not only correct in theory but also effective in practice.

58. Debugging Numerical Stability (NaNs, Infs)

Even if the model produces correct outputs most of the time, there's a hidden danger in deep learning: numerical instability. This happens when floating-point numbers inside the computation blow up to infinity (`Inf`) or collapse into "not a number" (`NaN`). When this occurs, training usually grinds to a halt—loss becomes undefined, gradients explode, and parameters no longer update meaningfully.

Why NaNs and Infs happen

Neural networks involve many multiplications, exponentials, and divisions. On paper, all of these are fine. But computers store numbers with limited precision (32-bit floats in this case). When values get too large or too small, they can no longer be represented correctly.

Common sources include:

- Softmax overflow: computing `exp(x)` on large positive numbers leads to `Inf`.
- Division by very small numbers: for example, dividing by `sqrt(v_hat) + eps` in AdamW can produce instability if `eps` is too small.
- Exploding gradients: during backpropagation, errors compound across many layers, producing extremely large values.
- Improper initialization or learning rates: weights that are too large or step sizes that are too aggressive can push activations outside a stable range.

How to detect instability in *llm.c*

Because the code is written in C without automatic checks, NaNs and Infs can spread silently unless you look for them. Some useful strategies include:

1. Assertions: insert `assert(!isnan(value) && !isinf(value));` inside loops to catch bad values immediately.
2. Debug prints: log sample values from activations or gradients each step to see if they drift toward extremely large numbers.
3. Check the loss: if the loss suddenly becomes `nan` or `inf`, that's a strong signal something went wrong upstream.
4. Small runs: testing on tiny sequences and batches makes it easier to inspect values directly.

How to fix instability

Several practical techniques help keep numbers stable:

- Add an epsilon: in divisions or square roots, add a small constant (like `1e-8`) to prevent division by zero.
- Rescale before softmax: subtract the maximum value in the vector before computing exponentials. This keeps values in a safe range.
- Gradient clipping: cap gradients so they cannot exceed a certain norm. This stops runaway updates.
- Adjust learning rate: if training diverges, lowering the learning rate often restores stability.
- Check data: corrupted inputs or unexpected tokens can inject extreme values into the model.

Example scenario

Suppose you're training GPT-2 on Tiny Shakespeare. The first few steps look fine:

```
step 0: train loss 4.95
step 1: train loss 4.72
step 2: train loss nan
```

This sudden jump to `nan` suggests instability. Checking the gradients reveals extremely large values in the attention weights. The fix might be lowering the learning rate from `1e-4` to `5e-5` or enabling gradient clipping.

Try it yourself

1. Train with a very high learning rate (`1e-2`) and watch how quickly NaNs appear.
2. Add a debug check inside `gpt2_forward` that prints when any activation exceeds `1e6`. Run a few steps and observe if values explode.
3. Modify the softmax code to omit subtracting the max. Compare stability before and after.
4. Add a gradient clipping routine and measure whether it prevents loss from diverging.

The takeaway

Numerical stability is the difference between a model that trains smoothly and one that collapses after a few steps. By anticipating where NaNs and Infs can arise, adding checks, and applying stabilizing tricks, you make *llm.c* robust. This ensures that experiments are reliable and that debugging focuses on real algorithmic issues rather than avoidable numerical traps.

59. From Unit Test to Full Training Readiness

Unit tests are the first line of defense: they check whether small, isolated parts of the code—like embeddings, attention, or softmax—produce the correct outputs. But passing unit tests isn't the same as being ready for full training. The transition from "this layer works" to "the whole system learns correctly over thousands of steps" involves additional challenges.

The gap between unit tests and training

- Unit tests check correctness: for example, verifying that `gpt2_forward` produces the same logits as PyTorch on a single batch.
- Training readiness checks robustness: making sure the model can run repeatedly for thousands of steps without crashing, diverging, or leaking memory.

Think of it like testing a car. Unit tests are like checking the brakes, headlights, and steering individually. Training readiness is taking the car on a 500-mile road trip and making sure nothing overheats, rattles loose, or fails under stress.

What needs to be validated for training readiness

1. Loss curve behavior Run the training loop for several hundred steps. The training loss should steadily decrease, matching known curves from PyTorch. If it stagnates or spikes, something is wrong in gradients or the optimizer.
2. Validation runs Regularly measure validation loss during training. If it decreases at first and then stabilizes, that shows the model is generalizing. If it decreases too quickly and then shoots up, that suggests overfitting.
3. Memory stability Training uses more memory than inference because of gradients and optimizer states. A memory leak—forgetting to free arrays or reallocating without release—will cause the program to crash after many steps.
4. Optimizer state updates Check that AdamW accumulates `m` and `v` correctly over many iterations. If bias correction is missing, loss curves will diverge from expected baselines.
5. Reproducibility With the same random seed, two runs should produce nearly identical loss curves. Small differences are normal, but major deviations suggest nondeterministic bugs.

Why it matters

Without this step, you might believe your implementation is complete after unit tests, only to discover that training silently fails at step 500. Training readiness ensures the system is not only mathematically correct in small pieces but also practically usable for long-running experiments.

This is also where confidence in deploying the code comes from. Passing training readiness means others can clone the repository, run the scripts, and expect stable training without mysterious crashes.

Example scenario

You run `test_gpt2.c` and all outputs match PyTorch within tolerance—great. Then you launch training for 5,000 steps. After step 600, the loss becomes `nan`. Investigation reveals that `gpt2_update` wasn't applying bias correction properly, so the optimizer went unstable. That's a bug you'd never catch with a one-batch unit test, but training readiness exposes it.

Try it yourself

1. Run training for 1,000 steps on Tiny Shakespeare and log the loss every 10 steps. Check that it decreases smoothly.
2. Add validation runs every 100 steps. Watch for the classic gap between train loss (lower) and validation loss (slightly higher).
3. Use `htop` or similar tools to monitor memory usage during training. Confirm that it stays steady rather than creeping upward.
4. Run the same training twice with the same seed. Compare the two loss curves—are they nearly identical?

The takeaway

Unit tests prove the pieces are correct. Training readiness proves the whole system works under real conditions. Both are necessary. Together, they give you the confidence that *llm.c* isn't just a collection of working parts, but a functioning engine capable of training GPT-2 models end to end.

60. Limitations of CPU Testing

Testing GPT-2 on CPU is invaluable for verifying correctness, but it comes with clear limits. Understanding these limitations helps you interpret the results properly and know when it's time to move on to GPU-based experiments.

Speed constraints

The most obvious limitation is speed. CPUs are optimized for general-purpose tasks, not the massive parallelism that neural networks demand. A single forward and backward pass on GPT-2 124M can take seconds or even minutes on CPU, while a GPU might handle it in milliseconds. This makes:

- Full-scale training impractical: training GPT-2 124M to convergence could take weeks or months on CPU.

- Experiment cycles slower: testing new optimizations or debugging is slowed because each run takes longer.

For this reason, CPU testing is best suited for small-scale sanity checks, not full training runs.

Memory overhead

CPU memory is typically more abundant than GPU VRAM, but slower. The bottleneck often isn't "do we have enough RAM?" but "how quickly can we move data in and out of memory?" As sequence length `T` grows, the activations balloon, and cache efficiency drops. This makes even medium-sized runs sluggish.

Limited realism

Although CPU runs confirm that the math is correct, they don't always reflect the realities of GPU execution. For example:

- CUDA kernels have different numerical characteristics (fused operations, different rounding).
- GPU memory layouts can expose bugs that CPU arrays hide.
- Parallel execution may create timing or synchronization issues that never appear on CPU.

So while CPU parity with PyTorch is necessary, it isn't sufficient. You must repeat testing once CUDA code is introduced.

Loss of scale insights

A CPU test can prove correctness for a few batches, but it doesn't tell you how the code scales under heavy load. On GPU, you learn about kernel efficiency, memory throughput, and distributed training. CPU tests simply don't expose those concerns.

Why it matters

CPU testing is the foundation: it proves the algorithm is implemented correctly, step by step, without relying on specialized hardware. But if you stop there, you'll miss the bigger picture of performance and scalability. CPU results should be treated as a green light to proceed, not the final word on readiness.

Example scenario

Suppose you run 500 training steps on Tiny Shakespeare. The loss curve drops exactly as expected—success. But training on CPU is so slow that finishing an epoch takes several hours. This validates correctness, but makes it obvious that GPUs are required for meaningful experiments.

Try it yourself

1. Train GPT-2 124M on CPU for 100 steps and record the time per step. Extrapolate how long it would take to run 100k steps.
2. Increase sequence length from 64 to 512 and observe how memory access times affect throughput.
3. Compare your CPU loss curve with a GPU run from PyTorch. Notice they align in shape but differ dramatically in speed.
4. Use profiling tools (`perf`, `valgrind`, or `gprof`) to see which CPU functions dominate runtime.

The takeaway

CPU testing is the safe laboratory where you validate correctness, catch numerical errors, and reproduce known loss curves. But its limitations—slow speed, reduced realism, and lack of scaling insights—mean it's only a first step. Once CPU testing passes, the journey continues with GPU testing, profiling, and multi-device scaling.

Chapter 7. CUDA Training (`train_gpt2.cu`)

61. CUDA Architecture Overview (streams, kernels)

When the CPU version of *llm.c* runs, it executes instructions one after another on your processor cores. This is fine for small models or debugging, but deep learning workloads—especially Transformers like GPT-2—demand an enormous number of floating-point operations. To handle that, *llm.c* also includes CUDA versions of the training loop that shift computation to NVIDIA GPUs.

At a high level, CUDA is NVIDIA's programming model that lets developers write code to run directly on the GPU. Unlike CPUs, which might have a few cores optimized for general-purpose tasks, GPUs contain thousands of simpler cores designed to process large batches of data in parallel. CUDA provides the tools to organize work so that those cores can stay busy.

Kernels: Small Programs That Run on the GPU

In CUDA, a *kernel* is a function that runs on the GPU. When you launch a kernel, you don't call it once like a normal C function—you launch thousands of copies at the same time. Each copy handles a different piece of the data. For example, if you want to multiply two vectors of a million elements, you can launch a million GPU threads, each multiplying one pair of numbers.

In *llm.c*, kernels are used for operations that can be expressed in terms of lots of small, independent tasks. Examples include:

- Applying the GeLU activation function elementwise to a big tensor.
- Adding residual connections across every dimension.
- Normalizing values in LayerNorm.

For bigger, structured operations like matrix multiplications (GEMMs), the CUDA code often relies on specialized libraries such as cuBLAS or cuBLASLt, which are highly tuned for NVIDIA GPUs.

Streams: Overlapping Work

A GPU has the ability to handle multiple tasks at once. CUDA introduces the idea of *streams*, which are sequences of operations that run in order relative to each other, but can overlap with operations in other streams. This means:

- While one kernel is executing, another can start transferring data between CPU and GPU.
- Computation and communication can overlap, reducing idle time.

In the training loop of *llm.c*, streams let you schedule batches of work so that data preparation and model computation can proceed side by side. This is crucial for keeping the GPU saturated with useful work instead of waiting on the CPU.

The Memory Hierarchy

CUDA programming is also shaped by the GPU memory hierarchy:

- Registers: Fastest, private to each thread.
- Shared Memory: Small chunks of memory shared among threads in a block; much faster than global memory.
- Global Memory: Large, but slower. This is where tensors like weights, activations, and gradients usually live.

- Host Memory (CPU RAM): Separate from GPU memory; transferring between them can be slow and should be minimized.

For example, in the attention kernel, partial results might be stored in registers or shared memory while processing a block of the sequence, before writing the final result back to global memory.

How This Fits Into *llm.c*

In `train_gpt2.cu`, most of the heavy lifting is done by calls into cuBLAS/cuBLASLt and cuDNN for matrix multiplications and attention. But understanding the CUDA model—kernels, streams, and memory—helps explain:

- Why we batch operations the way we do.
- Why minimizing data transfers between CPU and GPU is so important.
- How GPU kernels map naturally to the kinds of tensor operations GPT-2 requires.

Why It Matters

Without CUDA, training GPT-2 would be painfully slow, even on a powerful CPU. CUDA gives access to thousands of cores working in parallel, but it also requires careful programming to avoid bottlenecks. Knowing about kernels, streams, and memory hierarchy is the foundation for understanding later sections where we dive into matrix multiplication, attention, and optimization strategies.

Try It Yourself

1. Write a simple CUDA kernel that adds two arrays elementwise. Compare its performance to a CPU loop.
2. Modify the kernel to use shared memory and see if it improves performance for larger arrays.
3. Create two CUDA streams: one for computing a kernel, and another for copying data. Measure whether the operations overlap in time.
4. Use `nvprof` or `nsys` to profile a CUDA program and observe how kernels and memory transfers appear on the timeline.
5. Think about how you would split a big matrix multiplication across thousands of threads—each thread computing one row, one column, or one element? What are the tradeoffs?

The Takeaway

CUDA is not just about writing code for GPUs—it's about rethinking computation as thousands of small tasks that can run side by side. Kernels handle the per-thread work, streams let you schedule and overlap operations, and the memory hierarchy dictates how to organize data for maximum speed. All of these ideas come together in *llm.c*'s CUDA implementation, making training feasible for models like GPT-2.

62. Matrix Multiplication via cuBLAS/cuBLASLt

Matrix multiplication—often called GEMM (General Matrix-Matrix Multiply)—is the beating heart of deep learning. In GPT-2, most of the computation comes from multiplying large matrices: projecting embeddings into query, key, and value vectors, applying attention weights, and processing the MLP feed-forward layers. On the CPU, we saw this done with nested loops and mild optimizations. On the GPU, however, we need far more efficient approaches. That's where cuBLAS and cuBLASLt come in.

Why Matrix Multiplication Is So Central

Almost every step of a Transformer involves multiplying two big matrices:

- Embedding lookup can be seen as a matrix multiply between one-hot token vectors and the embedding table.
- The attention mechanism computes dot products between queries and keys, followed by a weighted sum of values.
- The MLP applies two fully connected layers, each of which is essentially a GEMM.

If you profile GPT-2, you'll find that GEMM operations dominate the runtime. That's why NVIDIA's libraries devote enormous effort to making these multiplications as fast as possible.

cuBLAS: The Classic Workhorse

cuBLAS is NVIDIA's GPU-accelerated version of the BLAS (Basic Linear Algebra Subprograms) library. It provides highly optimized implementations of GEMM and related routines. Under the hood, cuBLAS:

- Splits large matrices into tiles that fit into the GPU's shared memory.
- Schedules thousands of threads to compute different tiles in parallel.
- Uses fused multiply-add (FMA) instructions for high throughput.
- Adapts to different GPU architectures to exploit Tensor Cores where available.

A typical call looks like this:

```
cublasHandle_t handle;
cublasCreate(&handle);

float alpha = 1.0f, beta = 0.0f;
cublasSgemm(handle,
    CUBLAS_OP_N, CUBLAS_OP_N,
    m, n, k,
    &alpha,
    A, m,
    B, k,
    &beta,
    C, m);
```

Here `A` and `B` are the input matrices, and `C` is the result. The function handles all the low-level scheduling.

cuBLASLt: The Flexible Successor

While cuBLAS is powerful, it's somewhat rigid. cuBLASLt (Lightweight cuBLAS) is a newer API that adds:

- Better support for mixed precision (e.g., FP16 or BF16 inputs with FP32 accumulation).
- More control over algorithm selection, so developers can tune for performance or memory usage.
- Features like epilogues, which let you fuse additional operations (e.g., bias addition, activation functions) directly into the GEMM, reducing memory transfers.

In practice, cuBLASLt often outperforms cuBLAS because it can exploit Tensor Cores more aggressively and fuse multiple steps into a single kernel call.

Precision and Tensor Cores

On modern NVIDIA GPUs (Volta, Turing, Ampere, Hopper), Tensor Cores accelerate matrix multiplications dramatically when using FP16, BF16, or TF32. These special hardware units can perform matrix-multiply-and-accumulate on small blocks of numbers in a single instruction.

For example:

- On CPUs, multiplying two 16×16 matrices is done with many scalar multiplications.
- On GPUs with Tensor Cores, the entire block can be computed in one fused operation.

In GPT-2 training, using FP16 with cuBLASLt enables much higher throughput, while keeping master weights in FP32 to preserve numerical stability.

Practical Example in *llm.c*

In `train_gpt2.cu`, most of the calls to perform linear layers—such as projecting input activations into query, key, and value matrices—are implemented with cuBLAS/cuBLASLt. For instance:

- Inputs `(B, T, C)` are multiplied by a weight matrix `(C, 3C)` to produce `(B, T, 3C)`.
- Later, the output of attention `(B, T, C)` is multiplied by another projection matrix `(C, C)`.

Instead of writing custom kernels for each case, the code defers to cuBLAS/cuBLASLt, ensuring maximum performance across GPU architectures.

Why It Matters

Matrix multiplications are so frequent and heavy that their performance directly determines how fast you can train GPT-2. By leaning on cuBLAS/cuBLASLt, *llm.c* avoids reinventing the wheel and gets near-peak GPU efficiency. This makes the code clean, maintainable, and scalable to larger models.

Try It Yourself

1. Write a small CUDA program that multiplies two matrices using naive kernels, and compare its performance to cuBLAS.
2. Experiment with FP32 versus FP16 inputs and observe the speedup when Tensor Cores are enabled.
3. Enable cuBLASLt's epilogues to fuse bias addition into GEMM, and measure memory savings.
4. Profile GPT-2 training with `nvprof` or `nsys` to see how much time is spent in GEMM calls.
5. Try scaling up matrix sizes to simulate bigger models and note how performance grows relative to CPU implementations.

The Takeaway

Matrix multiplication is the computational engine of GPT-2, and on GPUs it's powered by cuBLAS and cuBLASLt. These libraries harness the GPU's architecture—tiling, Tensor Cores, mixed precision—to squeeze out maximum efficiency. Understanding how they work gives insight into why the GPU version of *llm.c* runs so much faster than the CPU version, and sets the stage for attention kernels and other CUDA-accelerated components.

63. Attention Kernels: cuDNN FlashAttention

The attention mechanism is at the core of every Transformer. It allows the model to weigh different parts of the input sequence when producing an output. For GPT-2, this means that when generating the next token, the model doesn't just look at the last word—it considers the entire sequence of words before it, adjusting how much each past token contributes. But attention is expensive. Naively, it scales quadratically with sequence length: for a sequence of 1024 tokens, you need to compute a 1024×1024 attention matrix. That's more than a million entries, and each must be computed, normalized, and multiplied back into the value vectors.

On CPUs, we saw how this is implemented step by step: queries, keys, and values are projected with matrix multiplications, dot products between queries and keys are computed, softmax is applied, and the result is multiplied by values. On GPUs, we want to do the same thing, but much faster. That's where cuDNN's FlashAttention comes into play.

What Is FlashAttention?

FlashAttention is an algorithm that rethinks how attention is computed. Instead of materializing the full attention matrix in memory, it computes softmax and the weighted sum in a streaming fashion. This reduces memory usage and improves cache efficiency.

Normally, attention involves these steps:

1. Compute scores = Q × K (queries times keys transpose).
2. Apply softmax over scores to get attention weights.
3. Multiply weights × V (values) to get the output.

The problem: step 1 produces a huge score matrix of size `(sequence_length × sequence_length)`. Storing and processing this full matrix becomes the bottleneck.

FlashAttention avoids storing the full matrix by computing attention block by block. It processes tiles of queries and keys, applies the softmax incrementally, and accumulates results directly into the output. This drastically cuts memory bandwidth requirements, which is critical for GPUs.

cuDNN FlashAttention in Practice

In *llm.c*'s CUDA training code, when `USE_CUDNN` is enabled, the code can take advantage of cuDNN's implementation of FlashAttention. This means:

- The library handles the tiling and streaming automatically.
- It can leverage Tensor Cores for mixed-precision computation (FP16/BF16 inputs with FP32 accumulation).
- It reduces memory use, which allows training longer sequences without running out of GPU memory.

From a developer's point of view, enabling cuDNN FlashAttention usually involves passing specific descriptors and flags to cuDNN routines rather than writing custom kernels. Instead of manually managing loops and softmax stability tricks, you hand over the responsibility to cuDNN, which has a heavily optimized kernel.

Why This Is a Game-Changer

The quadratic cost of attention has long been a bottleneck in scaling Transformers. With FlashAttention, the bottleneck shifts. The computation is still $O(N^2)$, but because memory is handled so much more efficiently, the GPU spends less time waiting on memory loads and more time doing actual math. This means:

- Training can be faster even at the same sequence length.
- You can push to larger sequence lengths (e.g., 2K or 4K tokens) without running out of GPU memory.
- Energy efficiency improves because you avoid redundant reads/writes to global memory.

Example: Why Memory Access Matters

Let's imagine a toy example with 4 tokens. A naive implementation might build a 4×4 attention matrix, compute softmax, and multiply by values. That's fine for 4 tokens, but with 1024 tokens, you'd be juggling matrices of a million entries. Even if each entry is just 2 bytes (FP16), that's megabytes of temporary storage per step. On a real GPU, constantly moving that in and out of global memory slows everything down.

FlashAttention says: instead of storing the whole million entries, compute them in chunks, normalize them on the fly, and immediately use them to update the output. This way, only small temporary blocks live in memory, and global memory pressure drops dramatically.

How It Shows Up in GPT-2 Training

When GPT-2 processes a batch of sequences, each block of the model applies attention. In the CUDA version of *llm.c*, these attention calls can be routed through cuDNN FlashAttention. Practically, this means that the inner loop of training—the part that would otherwise grind on those giant attention matrices—becomes leaner and faster.

This matters even more as models grow. For GPT-2 124M (12 layers, 12 heads, 1024 sequence length), attention is already expensive. For GPT-2 1.5B or LLaMA-style models with longer contexts, FlashAttention can be the difference between feasible training and "out of memory" errors.

Why It Matters

Attention is the defining operation of Transformers, but it's also their Achilles heel. FlashAttention addresses the biggest inefficiency—memory bandwidth—without changing the model's outputs. By using cuDNN's optimized kernels, *llm.c* ensures it runs close to hardware peak performance while still producing correct results. For anyone learning about deep learning systems, this is a perfect example of how algorithmic innovations (streaming softmax) and hardware-level optimizations (Tensor Cores, tiling) combine to make state-of-the-art training practical.

Try It Yourself

1. Run GPT-2 training in *llm.c* with `USE_CUDNN=0` and then with `USE_CUDNN=1`. Compare training speed and GPU memory usage.
2. Write a naive CUDA kernel that builds the full attention matrix, then benchmark it against cuDNN FlashAttention.
3. Vary sequence lengths (128, 512, 1024, 2048) and see how performance diverges between naive and FlashAttention implementations.
4. Examine how mixed precision interacts with FlashAttention—try FP16 versus BF16.
5. Explore the FlashAttention paper and compare its algorithmic explanation with what you see in practice using *llm.c*.

The Takeaway

Attention is expensive, but it doesn't have to be crippling. FlashAttention shows that clever algorithm design plus hardware-aware implementation can shrink the memory bottleneck dramatically. By leaning on cuDNN's implementation, *llm.c* can train GPT-2 models more efficiently, and learners get a real-world view of how deep learning libraries squeeze performance out of GPUs.

64. Mixed Precision: FP16/BF16 with Master FP32 Weights

Training large models like GPT-2 involves multiplying and adding enormous amounts of numbers—billions of operations in every training step. GPUs can do this very quickly, but the type of numbers you use matters a lot. Traditionally, training is done in 32-bit floating point (FP32), which gives good precision but is heavy on memory and compute. Modern GPUs offer special hardware—Tensor Cores—that run much faster when using reduced precision, such as FP16 (half-precision floating point) or BF16 (bfloat16). This technique is called mixed precision training.

Why Mixed Precision Helps

Using FP16 or BF16 has two main benefits:

1. Speed: GPUs can perform more FP16/BF16 operations per clock cycle than FP32. For example, NVIDIA Tensor Cores are specifically designed to accelerate half-precision math, often delivering 2× or more throughput.
2. Memory: FP16/BF16 values take half the storage of FP32. That means you can fit larger batches or longer sequences into the same GPU memory, which is critical for scaling models.

But reduced precision comes with a tradeoff: it's easier for numbers to underflow (become zero) or overflow (become infinity), which can destabilize training.

Master Weights in FP32

The trick used in *llm.c* (and also in PyTorch and TensorFlow) is to keep a master copy of weights in FP32. Here's the process:

1. During the forward pass, weights are cast to FP16/BF16 so the GPU can run the math on Tensor Cores.
2. The gradients are computed in reduced precision as well.
3. When it's time to update parameters, the optimizer applies updates to the FP32 master copy.
4. The updated master weights are cast back to FP16/BF16 for the next forward pass.

This way, you get the speed and memory savings of mixed precision without fully losing the stability of FP32.

FP16 vs. BF16

Both FP16 and BF16 use 16 bits, but they split the bits differently:

Format	Exponent Bits	Mantissa Bits	Range	Precision
FP16	5	10	Smaller	Higher precision for small numbers
BF16	8	7	Wider	Rougher precision, better range

- FP16 has better precision near zero but a narrower range, so it's more prone to overflow.
- BF16 has the same exponent size as FP32, giving it a much wider range but less precision.

Modern NVIDIA GPUs (Ampere, Hopper) support both, but BF16 is often preferred for stability in very large models.

Example in Practice

Imagine training GPT-2 with a sequence length of 1024 and batch size of 32. With FP32, the activations might take ~12 GB of GPU memory. Switching to FP16 halves that to ~6 GB, leaving room for larger models or more sequences.

In *llm.c*, enabling mixed precision means the forward pass can look something like this:

1. Cast embeddings, weights, and activations to FP16/BF16.
2. Run matrix multiplications on Tensor Cores (very fast).
3. Compute gradients in reduced precision.
4. Convert gradients back to FP32 for stable updates.

This flow is invisible to the high-level code but handled internally in CUDA/cuBLAS/cuDNN calls.

Common Challenges

Mixed precision introduces new wrinkles:

- Loss scaling: small gradients may underflow to zero in FP16. The solution is to multiply the loss by a large factor during backpropagation, then divide gradients back later. This preserves information.
- Debugging: NaNs and Infs become more common when switching to FP16. Careful monitoring is required to catch these early.
- Performance tuning: Not all operations benefit equally from FP16. For example, reductions (like summing a large array) may lose too much precision unless done in FP32.

Why It Matters

Mixed precision is one of the key reasons modern Transformers can be trained efficiently on today's hardware. Without it, many models would require double the GPU memory and much more time to train. By combining FP16/BF16 for speed and memory efficiency with FP32 master weights for stability, *llm.c* mirrors the strategy used in production frameworks. This shows how even a minimalist codebase can teach the cutting-edge tricks that power real-world large-scale training.

Try It Yourself

1. Train GPT-2 in *llm.c* with FP32 only, then repeat with FP16. Compare memory usage (`nvidia-smi`) and runtime per step.
2. Experiment with FP16 vs. BF16 if your GPU supports both. Observe whether one is more stable.
3. Intentionally remove the FP32 master weights (update parameters in FP16 only) and see how quickly training diverges.
4. Plot validation loss curves with FP32, FP16, and BF16 runs to see if the model quality differs.
5. Try scaling the batch size up with FP16 and note how much bigger a model you can fit into the same GPU.

The Takeaway

Mixed precision combines the best of both worlds: the speed and memory efficiency of FP16/BF16 with the stability of FP32. This technique has become a standard in deep learning, and *llm.c* demonstrates it in a clear, accessible way. It's not just a neat optimization—it's what makes training large language models on modern GPUs feasible at all.

65. Loss Scaling in Mixed Precision Training

When training in mixed precision (FP16 or BF16), one of the biggest challenges is numerical underflow. Gradients can become so small that they round down to zero when represented in 16-bit format. If that happens too often, the optimizer stops receiving meaningful updates, and training can stagnate or collapse. To address this, frameworks introduce a technique called loss scaling.

The Core Idea

Loss scaling works by multiplying the loss value by a constant factor (called the scale factor) before starting backpropagation. Since gradients are proportional to the loss, this also multiplies all gradients by the same factor, making them larger and less likely to underflow when stored in FP16.

At the end of backpropagation, the gradients are divided by the same scale factor, restoring their correct values before the optimizer step.

Mathematically:

1. `scaled_loss = loss × scale`
2. Compute gradients of `scaled_loss` → produces `scaled_gradients`
3. `true_gradients = scaled_gradients ÷ scale`

The optimizer then uses `true_gradients` to update the weights.

Static vs. Dynamic Scaling

There are two common approaches:

- Static scaling: Use a fixed scale factor throughout training. For example, always multiply the loss by `1024`. This is simple but risky; if the scale is too high, gradients may overflow to infinity. If it's too low, underflow still happens.
- Dynamic scaling: Adjust the scale factor on the fly. If overflows (NaNs or Infs) are detected, the scale factor is reduced. If training proceeds smoothly, the scale factor is gradually increased. This balances stability and efficiency.

In practice, dynamic scaling is the standard. Libraries like PyTorch's `GradScaler` automatically handle this logic, so users don't have to tweak values manually.

How It Appears in *llm.c*

The minimal design of *llm.c* doesn't yet include automatic loss scaling, but the idea fits neatly into its training loop. Before calling `gpt2_backward`, you would scale the loss. After gradients are computed, you would unscale them before `gpt2_update`. Conceptually:

```
float scale = 1024.0f; // example scale factor
float scaled_loss = model.mean_loss * scale;
gpt2_backward_scaled(&model, scaled_loss); // backprop with scaled loss
unscale_gradients(&model, scale); // divide gradients by scale
gpt2_update(&model, lr, beta1, beta2, eps, weight_decay, step);
```

This is not yet in the repository, but it's how one could extend *llm.c* to support stable FP16 training.

Why It Matters

Without loss scaling, mixed precision can fail silently. Training might appear to run, but the gradients may be effectively zero for many parameters. This wastes GPU time and produces poor results. With loss scaling, FP16/BF16 training becomes both fast and reliable, combining the hardware speedups with numerical stability.

Example Scenario

Suppose you are training GPT-2 with FP16 and notice that the validation loss barely decreases after several hundred steps. One possible reason is gradient underflow. By enabling loss scaling with a scale factor of 512 or 1024, you might suddenly see the loss curve behave normally again, matching the FP32 baseline.

Try It Yourself

1. Train with FP16 but without loss scaling. Monitor whether the loss decreases meaningfully.
2. Add a static scale factor (like 512) and rerun. Observe improvements in stability.
3. Implement a simple dynamic scaler: start with 128, double it if no NaNs appear for 100 steps, halve it if NaNs are detected.
4. Compare training curves (FP32 vs. FP16 with and without scaling) to see the effect.
5. Experiment with very large scale factors to trigger overflow intentionally, then watch how dynamic scaling recovers.

The Takeaway

Loss scaling is the hidden ingredient that makes mixed precision training practical. By rescaling the loss and gradients, we protect tiny numbers from disappearing in FP16 while still enjoying the massive performance and memory benefits. Even in a minimal codebase like *llm.c*, understanding loss scaling bridges the gap between a model that trains poorly and one that matches FP32 performance at half the cost.

66. Activation Checkpointing and Memory Tradeoffs

Training deep networks like GPT-2 involves storing a large number of activations—the intermediate outputs produced at every layer during the forward pass. These activations are needed later in the backward pass to compute gradients. The problem is that they take up a huge amount of GPU memory. For a 12-layer GPT-2 with long sequences and large batch sizes, activations can consume more memory than the model weights themselves.

Why Activations Matter

Let's say you have a batch size of 8, sequence length of 1024, hidden size of 768, and 12 layers. Each layer produces an activation tensor of shape `(batch_size, sequence_length, hidden_size)`, or `8 × 1024 × 768`. That's about 6.3 million numbers per layer. Multiply by 12 layers, and you have ~75 million numbers. At FP16, that's around 150 MB per forward pass just for storing activations, and this grows with larger models.

If you scale up to GPT-2 Medium or GPT-2 XL, this number balloons quickly into gigabytes, which may not fit in GPU memory.

The Idea of Checkpointing

Activation checkpointing offers a tradeoff: instead of storing all activations, you only keep a small subset (the checkpoints) and recompute the rest during the backward pass.

1. During forward pass: save only checkpoints (for example, the activations at the end of each transformer block).
2. During backward pass: when gradients for a layer are needed, recompute the missing activations by running part of the forward pass again.

This saves memory at the cost of extra computation.

How It Works in GPT-2

A GPT-2 block has multiple steps: embedding lookup, attention, MLP, layer norm, residual connections. Normally, you'd store every output tensor. With checkpointing, you might only store the input to each block and discard the intermediate results. When backpropagation reaches that block, you rerun the forward pass locally to regenerate those results, then compute gradients.

This reduces memory usage almost linearly with the number of discarded activations, at the cost of roughly 30–40% more compute.

In *llm.c* Context

llm.c doesn't yet include activation checkpointing in its minimal implementation, but it's a natural extension. In CUDA, this might be implemented by wrapping blocks of code with a "checkpoint" function that decides whether to save or discard activations. In PyTorch, the equivalent is `torch.utils.checkpoint`.

If you train with longer sequences (e.g., 2048 tokens instead of 1024), checkpointing could mean the difference between fitting in memory or running into out-of-memory (OOM) errors.

Why It Matters

Modern GPUs have enormous compute capacity, but memory remains the bottleneck. Checkpointing shifts the tradeoff: you spend a bit more compute (re-running some forward passes) in exchange for freeing up gigabytes of memory. This lets you:

- Train larger models on the same hardware.
- Use longer sequence lengths for better context handling.
- Increase batch size for more stable gradients.

In practice, this technique is used in nearly every large-scale Transformer training run today.

Example Analogy

Think of it like studying for an exam. You could take detailed notes on every page of the textbook (storing all activations), but your notebook would get huge. Alternatively, you could only mark the chapter headings (checkpoints) and re-read sections when you need them during review (recomputation). It takes more time, but saves notebook space.

Try It Yourself

1. Run training on a GPU with long sequences until you hit an out-of-memory error.
2. Implement a simple checkpointing scheme where you only store activations every other layer.
3. Measure how much memory usage decreases (using `nvidia-smi`) and how much runtime increases per step.
4. Experiment with different checkpointing frequencies—every layer, every 2 layers, every 4 layers—and find the balance between memory savings and compute overhead.
5. Compare validation loss curves to confirm that checkpointing does not affect training quality (only runtime).

The Takeaway

Activation checkpointing is a clever strategy to bend the memory limits of GPUs. By discarding and recomputing activations on demand, you can fit models or sequence lengths that would otherwise be impossible. The tradeoff is extra computation, but with today's hardware, compute is usually cheaper than memory. This technique is one of the quiet enablers behind scaling Transformers to billions of parameters.

67. GPU Memory Planning: Parameters, Gradients, States

When training a GPT-2 model on GPU, one of the most important practical challenges is managing memory. Every tensor—the model's parameters, gradients, optimizer state, and activations—must fit into limited GPU RAM. Unlike CPUs, where you can often rely on swap space or large RAM pools, GPU memory is tight and unforgiving. If you exceed the limit, the program crashes with an out-of-memory error.

Breaking Down What Takes Memory

1. Parameters (Weights) These are the trainable values of the model: embeddings, attention projections, MLP weights, and so on. For GPT-2 124M, there are about 124 million parameters.

    - In FP32, that's roughly 500 MB.
    - In FP16/BF16, it's about 250 MB. Larger GPT-2 models (355M, 774M, 1.5B) scale this up proportionally.

2. Gradients For each parameter, the backward pass produces a gradient tensor of the same size. If parameters take 500 MB, gradients also take ~500 MB in FP32. Mixed precision can halve this.

3. Optimizer States Optimizers like AdamW don't just store gradients—they also track moving averages (`m` and `v`). Each adds another full-sized tensor. With AdamW, you often end up with 3× the parameter size: weights + m + v.

4. Activations During the forward pass, every layer's intermediate outputs must be stored for the backward pass. This is often the largest single consumer of memory. For a 12-layer GPT-2 with sequence length 1024 and batch size 8, activations can easily exceed several GB. Checkpointing (as discussed earlier) helps reduce this.

A Simple Calculation Example

For GPT-2 124M in FP32:

- Parameters: ~500 MB
- Gradients: ~500 MB
- AdamW states: ~1 GB (two copies)
- Activations: 2–4 GB (depends on batch size and sequence length)

Total: ~3–6 GB, which fits on most modern GPUs.

For GPT-2 774M in FP32:

- Parameters: ~3 GB
- Gradients: ~3 GB
- AdamW states: ~6 GB
- Activations: 8–12 GB

Total: ~20+ GB—too large for many GPUs unless you use tricks like FP16 and checkpointing.

Strategies for Memory Efficiency

1. Mixed Precision Using FP16/BF16 cuts parameter, gradient, and optimizer sizes in half. Instead of 20 GB, you may get by with ~10 GB.
2. Activation Checkpointing Store fewer activations and recompute them during backpropagation. This often saves multiple GB.
3. Gradient Accumulation Instead of training with a huge batch at once, split it into smaller micro-batches and accumulate gradients across them. This reduces activation memory requirements.
4. Parameter Sharding (Advanced) In multi-GPU setups, parameters and optimizer states can be split across devices (e.g., ZeRO optimizer in DeepSpeed). While not in *llm.c*, it's a common technique at scale.

In *llm.c*

The `GPT2` struct organizes memory into fields like `params_memory`, `grads_memory`, `m_memory`, and `v_memory`. These are allocated as flat arrays, making it easy to calculate their size. This minimal design highlights the reality: for every parameter, there's at least one matching gradient and potentially two optimizer state values.

This structure mirrors how full frameworks like PyTorch allocate memory, but *llm.c* exposes it transparently so you can see exactly what's taking up space.

Why It Matters

When training models, memory is usually the first limit you hit, not compute. Even if your GPU is powerful enough to handle the math, if you run out of memory, you can't proceed. Understanding how parameters, gradients, optimizer states, and activations interact helps you design training runs that actually fit.

Try It Yourself

1. Calculate memory usage for GPT-2 124M, 355M, and 774M using FP32 vs FP16. Compare your numbers to your GPU's memory size.
2. Run *llm.c* with increasing batch sizes until you hit an out-of-memory error. Record the exact point where it breaks.
3. Enable mixed precision and checkpointing to see how much further you can push sequence length or batch size.
4. Write a script to print the sizes of `params_memory`, `grads_memory`, and optimizer states in *llm.c*. Compare this to `nvidia-smi` output during training.
5. Experiment with reducing optimizer states (e.g., try SGD instead of AdamW) to see the memory difference.

The Takeaway

Training Transformers is not just about writing the forward and backward passes—it's about planning memory carefully. Parameters, gradients, optimizer states, and activations all compete for limited GPU RAM. By understanding these categories and using techniques like mixed precision and checkpointing, you can fit bigger models or longer contexts on the same hardware. This balance between memory and compute is at the heart of scaling modern deep learning.

68. Kernel Launch Configurations and Occupancy

When writing CUDA code for training models like GPT-2, one of the most important yet subtle factors in performance is how kernels are launched. A kernel is just a function that runs on the GPU, but the way you configure it—the number of threads, blocks, and how work is divided—can make the difference between a GPU that runs at 10% efficiency and one that's near peak utilization.

Threads, Blocks, and Grids

CUDA organizes computation hierarchically:

- Thread: the smallest unit of execution. Each thread runs the kernel code once.
- Block: a collection of threads that share fast, on-chip memory (called shared memory).
- Grid: a collection of blocks. Together, the grid represents the entire kernel launch.

When you launch a kernel, you decide:

```
my_kernel<<<num_blocks, threads_per_block>>>(...);
```

For example:

```
int threads = 256;
int blocks = (N + threads - 1) / threads;
my_kernel<<<blocks, threads>>>(...);
```

Here, `N` might be the total number of elements to process. The division ensures that all elements get covered.

What Is Occupancy?

Occupancy refers to how many threads are active on a GPU relative to the maximum possible. Higher occupancy usually means the GPU is better utilized, but it's not the only factor—memory access patterns and instruction throughput also matter.

Each GPU has a fixed number of Streaming Multiprocessors (SMs), and each SM can support only a certain number of threads and blocks at once. If your kernel launch doesn't provide enough threads, the GPU will be underutilized. If you launch too many, they may compete for shared memory and registers, leading to inefficiency.

Example: A Simple Vector Add

Suppose you want to add two arrays of size `N = 1e6`.

- If you use `threads_per_block = 32`, you'll have many tiny blocks. This wastes parallelism because modern GPUs are designed to run hundreds of threads per SM.
- If you use `threads_per_block = 1024`, you may hit the hardware limit but run very large blocks that restrict scheduling flexibility.
- A good balance might be 256 or 512 threads per block, which lets the GPU overlap computation and memory access effectively.

In Transformer Training

For GPT-2 in *llm.c*, most heavy lifting is done by:

- Matrix multiplications (handled by cuBLAS/cuBLASLt). These libraries pick kernel launch parameters automatically.
- Attention and normalization kernels (custom or cuDNN). When written by hand, launch configuration becomes crucial.

For example, in a softmax kernel over a sequence of 1024 tokens:

- You might launch one block per sequence row, with 1024 threads per block (one per token).
- Alternatively, you might launch multiple blocks per row, each handling a tile of tokens, if shared memory limits require it.

Choosing wisely can double or triple performance.

Balancing Factors

When configuring kernels, you balance:

1. Occupancy: Are enough threads active to use the GPU fully?
2. Memory Coalescing: Do threads access memory in aligned, sequential chunks (which is fast) or scattered patterns (which is slow)?
3. Shared Memory and Registers: Each block has limited resources. If your kernel uses too much shared memory, fewer blocks can fit per SM, reducing occupancy.
4. Arithmetic Intensity: If the kernel does a lot of math per memory load, occupancy matters less; if it's memory-bound, occupancy matters more.

Why It Matters

In GPU training, kernel launch decisions directly control how efficiently hardware is used. Two kernels implementing the same math can differ by 5–10× in runtime purely because of launch configuration. cuBLAS and cuDNN automate much of this for matrix-heavy ops, but understanding it is crucial when writing custom kernels.

Try It Yourself

1. Write a simple CUDA kernel for vector addition with different `threads_per_block` values (32, 128, 256, 512, 1024). Measure runtime and see which is fastest.
2. Use `nvprof` or `nsys` to inspect occupancy of kernels during GPT-2 training. Note which kernels run at <50% occupancy.
3. Experiment with a softmax kernel: launch one block per row vs. multiple blocks per row. Compare performance and memory use.
4. Explore how shared memory allocation per block affects occupancy by artificially increasing shared memory usage.
5. Compare cuBLAS GEMM (matrix multiply) performance to a naive CUDA implementation and observe how kernel configuration explains the speed difference.

The Takeaway

Kernel launch configuration is the hidden lever of GPU performance. By adjusting how threads and blocks are assigned, you control how much of the GPU is kept busy, how well memory bandwidth is used, and how smoothly computations flow. For models like GPT-2, libraries handle most kernels, but knowing what's happening under the hood is key to writing or debugging efficient CUDA code.

69. CUDA Error Handling and Debugging

When writing or running CUDA code, one of the most frustrating parts is that errors often don't show up immediately. Unlike CPU code, where an invalid pointer or division by zero can crash right away, CUDA launches kernels asynchronously. This means the host code (running on the CPU) queues up GPU work and moves on, while the GPU processes it in the background. If something goes wrong inside the kernel, the error might not be visible until later—sometimes only after you try to synchronize.

Common Error Sources

1. Out-of-bounds memory access A kernel thread tries to read or write past the end of an array. This can silently produce incorrect results or crash the program.
2. Invalid memory alignment Some CUDA operations require pointers to be aligned. Misaligned access can degrade performance or trigger errors.
3. Illegal instruction or unsupported hardware feature Using Tensor Cores on an older GPU, or using instructions not supported by your GPU's compute capability, can fail.

4. Out of memory (OOM) Allocating more GPU memory than available causes runtime errors. Unlike CPU memory, GPUs cannot "swap" to disk.
5. Race conditions Threads within a block or across blocks accessing the same memory location without synchronization can corrupt results.

How CUDA Reports Errors

Every CUDA runtime API call returns an error code. For example:

```
cudaError_t err = cudaMalloc(&ptr, size);
if (err != cudaSuccess) {
    printf("Error: %s\n", cudaGetErrorString(err));
}
```

Similarly, after launching a kernel, you should check:

```
my_kernel<<<blocks, threads>>>(...);
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    printf("Kernel launch failed: %s\n", cudaGetErrorString(err));
}
cudaDeviceSynchronize(); // forces the GPU to finish and report errors
```

This pattern ensures that if something fails, you see it quickly instead of later.

Debugging Tools

1. cuda-gdb A GPU-aware debugger. Lets you step through CUDA kernels much like gdb on CPU code.
2. cuda-memcheck Detects out-of-bounds accesses, race conditions, and misaligned memory operations. Essential when kernels produce "mysterious" wrong outputs.
3. Nsight Systems / Nsight Compute Profiling tools that show kernel timelines, occupancy, memory throughput, and errors.
4. Sanity checks in code Often, simply inserting assertions (`assert(i < N)`) or zero-initializing memory can catch problems earlier.

Debugging in *llm.c*

In *llm.c*, most of the CUDA-heavy lifting is handled by cuBLAS and cuDNN. But when experimenting with custom kernels (e.g., softmax, masking, or layernorm), debugging becomes crucial. A small indexing mistake could make training diverge or crash with `nan` losses. By adding `cudaGetLastError()` checks after every kernel launch, you can catch issues right where they happen.

Example: A Softmax Bug

Imagine a kernel computing softmax across 1024 tokens per row. If one thread index accidentally runs past 1024, it may read garbage memory. Without error checking, you might just see "loss is NaN" 100 steps later. With `cuda-memcheck`, you'd immediately see:

```
Invalid global read of size 4
    at softmax.cu:42
    by thread (1025,0,0) in block (1,0,0)
```

Now you know exactly where to fix the bug.

Why It Matters

Training large models is expensive. A single bug in a CUDA kernel can waste hours of GPU time, produce invalid gradients, or silently corrupt weights. Robust error handling and debugging practices save not only frustration but also significant cost.

Try It Yourself

1. Write a CUDA kernel with an intentional bug (e.g., forget to check array bounds). Run it with and without `cuda-memcheck` to see the difference.
2. Add `cudaGetLastError()` after every kernel in a simple project and watch how it pinpoints issues earlier.
3. Experiment with Nsight Systems: run GPT-2 training and inspect kernel launches, checking for errors or unexpected stalls.
4. Train with bad initialization (e.g., NaNs in inputs) and see how error checking reports failures.
5. Introduce a race condition by having two threads update the same memory without `__syncthreads()`. Debug using `cuda-memcheck`.

The Takeaway

CUDA's asynchronous nature makes error handling less straightforward than CPU programming. But with the right tools—error codes, synchronization, cuda-memcheck, and debuggers—you can systematically catch and fix problems. In *llm.c*, this discipline ensures that CUDA kernels not only run fast but also run correctly, which is just as important when training large-scale models.

70. `dev/cuda/`: From Simple Kernels to High Performance

Inside the *llm.c* repository, there is a folder called `dev/cuda/`. At first glance it may look like a side experiment, but it's actually one of the most instructive parts of the project. The main training files (`train_gpt2.cu`, `train_gpt2_fp32.cu`) rely heavily on cuBLAS and cuDNN—optimized libraries that already deliver near-peak performance. But if you want to understand how CUDA really works under the hood, you have to look at how kernels are written from scratch, and that's exactly what this folder shows.

Why This Folder Exists

The goal of `dev/cuda/` is not to replace cuBLAS or cuDNN. Instead, it acts as a sandbox for:

- Building intuition about how GPU kernels are structured.
- Experimenting with small-scale implementations of operations like vector addition, matrix multiplication, or normalization.
- Comparing naive CUDA implementations to highly optimized library calls.
- Teaching developers how memory layout, thread synchronization, and shared memory affect performance.

It's a bridge: start simple with "hello world" style kernels, then step closer to the performance tricks used by NVIDIA's professional libraries.

A Journey from Naive to Optimized

1. Simple Elementwise Kernels The first step is usually a kernel where each thread processes one element. For example, adding two vectors `C[i] = A[i] + B[i]`. This teaches indexing, memory coalescing, and the idea of grids and blocks.
2. Reduction Kernels Next, you move to slightly harder tasks like summing an array. Now you need thread cooperation and synchronization (`__syncthreads()`), plus shared memory usage.

3. Matrix Multiplication (GEMM) A naive kernel might have each thread compute one output element by looping over the input dimension. It works, but is slow because it reloads data from global memory repeatedly. The optimized version uses tiling: load a tile of the matrix into shared memory, let threads reuse it many times, and then move to the next tile. This can speed up performance by 10× or more.
4. Advanced Optimizations Later examples may add warp-level primitives, vectorized loads, and Tensor Core usage. These bring performance closer to cuBLAS.

Educational Value

Seeing these steps side by side makes the performance story very tangible:

- A naive GEMM kernel might achieve 1% of cuBLAS speed.
- A tiled shared-memory GEMM can jump to 30–40%.
- With careful warp scheduling, it can reach 60–70%.
- cuBLAS goes further with hand-tuned assembly and Tensor Cores, pushing 90–95%.

This teaches that optimization is not magic—it's a sequence of logical improvements, each shaving off inefficiencies.

Why It Matters for GPT-2 Training

Even if you never plan to reimplement matrix multiplication yourself, understanding what happens in `dev/cuda/` helps explain why the main training loop in `train_gpt2.cu` is so fast. You see why cuBLAS/cuDNN kernels are black boxes of efficiency: because writing your own at that level is extremely hard.

But this also means you're better prepared to write custom kernels when you need them. For example, maybe you want to test a new activation function or a different attention mechanism. By borrowing patterns from the experimental kernels, you can build your own, test them, and compare to baselines.

Example: Vector Add Kernel

Here's a simple kernel you might find in this folder:

```
__global__ void vector_add(const float* A, const float* B, float* C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        C[i] = A[i] + B[i];
```

```
    }
}
```

It's trivial compared to GPT-2's attention, but this is where everyone starts. From here you scale up to 2D indexing for matrices, then to tiled shared-memory patterns.

Try It Yourself

1. Run a kernel from `dev/cuda/` that does naive matrix multiplication. Compare its runtime to cuBLAS for the same dimensions.
2. Modify the naive GEMM to use tiling with shared memory. Measure how performance improves.
3. Inspect PTX (the intermediate assembly) generated by NVCC for a simple kernel. Observe how memory loads are translated.
4. Add timing code around kernels to see how much performance scales with different block sizes.
5. Implement a new custom kernel (e.g., ReLU activation) and compare its speed to applying ReLU via cuDNN.

The Takeaway

The `dev/cuda/` folder is not about production training. It's about learning and experimenting. It starts with the simplest CUDA kernels and builds up to performance-conscious designs. This progression mirrors how professional libraries achieve their speed. By studying and experimenting here, you gain a deeper appreciation of what it takes to make GPUs run at full tilt—and you gain the skills to write your own kernels when the libraries don't provide what you need.

Chapter 8. Multi-GPU and Multi-node training

71. Data Parallelism in *llm.c*

When you want to train a large model, a single GPU often isn't enough. Either the model doesn't fit in memory, or the training takes too long. One of the simplest and most widely used ways to scale training across multiple GPUs is data parallelism. The idea is conceptually simple: instead of giving all the training data to one GPU, you split it into smaller batches, send each GPU a piece, let them process it independently, and then combine their results.

The Core Idea

Imagine you have a batch of 128 sequences and 4 GPUs. In data parallelism:

- GPU 0 sees sequences 0–31
- GPU 1 sees sequences 32–63
- GPU 2 sees sequences 64–95
- GPU 3 sees sequences 96–127

Each GPU runs the forward pass, computes the loss, and calculates gradients for its slice. At the end of the step, the gradients are averaged across GPUs, ensuring that all models stay synchronized. Every GPU holds a full copy of the model parameters, so they are always consistent after gradient averaging.

In *llm.c*

The *llm.c* repository keeps things minimal, so there isn't a full-fledged DeepSpeed or PyTorch DDP implementation. But the same principle applies:

- Each GPU gets a copy of the GPT-2 model.
- The batch is split across devices.
- After the backward pass, gradients from all GPUs must be synchronized.

This synchronization is usually done using NCCL all-reduce (covered in the next section), but the design remains data parallel at heart.

Why Data Parallelism Works

The forward and backward passes are embarrassingly parallel across different data samples. A token in sequence A doesn't need to know about a token in sequence B when computing gradients. As long as all GPUs agree on parameter updates after each step, splitting the batch is perfectly valid.

Example Walkthrough

Let's say we're training GPT-2 on TinyStories with batch size `B = 32` and sequence length `T = 64`.

- On a single GPU, the forward pass computes embeddings, attention, MLP, and loss for all 32 sequences.
- With 2 GPUs, we set `B = 16` on each. Each GPU processes 16 sequences in parallel.

- After backpropagation, both GPUs hold gradients for their half of the batch. Before applying the optimizer, the gradients are averaged so that the weight update is equivalent to training with the full batch of 32.

From the model's perspective, it's as if nothing changed—it just sees gradients from the whole batch.

Memory and Speed Benefits

- Memory: Each GPU stores only the activations for its local batch. This reduces per-GPU memory use, making it possible to train with larger global batches.
- Speed: Training steps finish faster because multiple GPUs share the work. For example, doubling the number of GPUs often cuts training time per step nearly in half, though communication overhead prevents perfect scaling.

Limitations

1. Communication Overhead Synchronizing gradients across GPUs can become expensive, especially with large models or when running across multiple nodes.
2. I/O Bottlenecks Feeding data to multiple GPUs fast enough requires efficient dataloaders and prefetching.
3. Optimizer State Replication With AdamW, each GPU also needs to store optimizer states (m and v). This means memory scales with the number of GPUs instead of shrinking.

Why It Matters

Data parallelism is the workhorse of deep learning scaling. It's conceptually easy to understand, straightforward to implement, and works well even for large models. In practice, nearly all large-scale GPT training begins with data parallelism, often enhanced by techniques like gradient accumulation or mixed precision.

Try It Yourself

1. Train GPT-2 in *llm.c* on a single GPU, then split the batch across two GPUs using `CUDA_VISIBLE_DEVICES`. Compare throughput and loss curves.
2. Experiment with increasing global batch size while keeping per-GPU batch size fixed. Notice how validation loss behaves.
3. Simulate gradient averaging by writing a simple script that averages arrays from two processes. Connect this idea back to how NCCL all-reduce works.

4. Measure the difference in memory usage per GPU when training with 1 vs 2 GPUs.
5. Run a small experiment with different numbers of GPUs (1, 2, 4) and plot how training time per step changes.

The Takeaway

Data parallelism splits the workload across GPUs by dividing the batch. Each GPU trains a full model replica on part of the data, then synchronizes gradients so that updates are consistent. It's simple but powerful, forming the foundation of scaling strategies in *llm.c* and in most deep learning frameworks. Without it, training GPT-2 and larger models on modern datasets would be impractical.

72. MPI Process Model and GPU Affinity

When you scale training beyond a single GPU, you need a way to manage multiple processes and devices. In the *llm.c* codebase, the minimalist approach relies on MPI (Message Passing Interface), a library that has been around for decades in high-performance computing. MPI provides a simple abstraction: you launch multiple processes, each assigned a rank (an ID number), and they can communicate with each other by sending and receiving messages.

In distributed deep learning, MPI typically works alongside NCCL (NVIDIA Collective Communications Library). MPI handles process management—spawning workers, assigning GPUs, setting up environment variables—while NCCL handles the actual gradient synchronization.

MPI Processes and Ranks

Suppose you want to train on 4 GPUs. MPI will start 4 processes. Each process:

- Loads the same GPT-2 model code.
- Initializes CUDA on one GPU.
- Reads a shard of the training data or the same dataset, depending on setup.

Each process gets a rank:

- Rank 0 → GPU 0
- Rank 1 → GPU 1
- Rank 2 → GPU 2
- Rank 3 → GPU 3

Ranks are important because they determine roles. For example, rank 0 often acts as the "master," printing logs or handling checkpoints, while the others focus purely on computation.

GPU Affinity

If you don't explicitly map processes to GPUs, they can all try to use the same device. That leads to oversubscription—multiple processes fighting for one GPU while the others sit idle. To prevent this, you set GPU affinity.

The environment variable `CUDA_VISIBLE_DEVICES` is the simplest way to do this. For example:

```
# Run with 4 GPUs
mpirun -np 4 \
  -x CUDA_VISIBLE_DEVICES=0,1,2,3 \
  ./train_gpt2_mpi
```

MPI automatically assigns process 0 to GPU 0, process 1 to GPU 1, and so on. Inside the code, you can confirm this by calling `cudaSetDevice(rank)`.

On multi-node clusters, GPU affinity also needs to consider network topology. You want each process close to its GPU and ideally aligned with the node's network card for faster NCCL communication.

Synchronization and Communication

After each forward and backward pass, each MPI process has its local gradients. These must be averaged across processes to keep the model weights consistent. MPI itself offers collective operations like `MPI_Allreduce`, but in practice, *llm.c* uses NCCL for GPU-to-GPU communication because it is faster and topology-aware. MPI sets up the group, NCCL does the heavy lifting.

Example Workflow

1. Launch: `mpirun -np 4 ./train_gpt2` starts 4 processes.
2. Initialization: Each process determines its rank and sets its GPU with `cudaSetDevice(rank)`.
3. Training loop:
    - Forward pass on each process's GPU.
    - Backward pass to compute gradients.
    - Gradients averaged with NCCL All-Reduce.
4. Update: Every process updates its copy of the weights.
5. Sync: At the next step, all model replicas are identical.

Debugging GPU Affinity Issues

If you accidentally misconfigure GPU affinity, symptoms include:

- Two processes trying to use the same GPU → out-of-memory errors.
- GPUs left idle because no process is assigned.
- Slowdowns because processes are spread inefficiently across sockets or PCIe lanes.

A quick way to debug is to print the rank and the GPU ID at startup:

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int device;
cudaGetDevice(&device);
printf("Process %d is using GPU %d\n", rank, device);
```

Why It Matters

MPI and GPU affinity might feel like low-level plumbing, but they are critical for scaling. If you don't get this right, training may run at a fraction of expected speed or crash outright. For small setups (2–4 GPUs), it may feel like overkill, but for larger clusters with 8, 16, or 64 GPUs, careful mapping is the difference between success and wasted compute time.

Try It Yourself

1. Train GPT-2 with `mpirun -np 2` and verify each process prints a different GPU ID.
2. Intentionally misconfigure `CUDA_VISIBLE_DEVICES` so both processes map to GPU 0, then observe the OOM error.
3. On a multi-GPU machine, experiment with running processes pinned to different GPUs. Measure training throughput.
4. Use `nvidia-smi topo -m` to view the PCIe topology of your GPUs. Try to align MPI ranks with nearby GPUs for better performance.
5. Print the time spent in all-reduce with different mappings to see how GPU affinity affects communication overhead.

The Takeaway

MPI is the backbone for managing multiple processes in distributed training, and GPU affinity ensures each process has exclusive access to the right device. Together, they lay the groundwork for efficient multi-GPU training in *llm.c*. Get these details right, and scaling is smooth; get them wrong, and you run into memory crashes, idle GPUs, or bottlenecked communication.

73. NCCL All-Reduce for Gradient Sync

Once each GPU finishes its forward and backward pass, it has a set of gradients that reflect only its portion of the training data. To keep the model parameters consistent across all GPUs, these gradients must be synchronized. The standard way to do this in modern deep learning systems is with All-Reduce, and NVIDIA's NCCL (NVIDIA Collective Communications Library, pronounced "Nickel") provides the optimized implementation.

What All-Reduce Does

All-Reduce is a collective communication operation. Every process (GPU) starts with its own local buffer of values—here, the gradients—and the operation combines them (using a reduction, usually summation) and distributes the result back to all processes.

Mathematically, if GPU 0 has `g0`, GPU 1 has `g1`, GPU 2 has `g2`, and GPU 3 has `g3`, then after All-Reduce, each GPU has the same result:

```
g_all = (g0 + g1 + g2 + g3) / 4
```

The division by 4 is optional—it depends whether you want a sum or an average—but averaging is common for gradient updates.

This ensures that every GPU applies the same weight update and stays synchronized with the others.

Why NCCL?

While MPI provides an All-Reduce primitive, NCCL is specifically optimized for GPUs. It knows about PCIe, NVLink, NVSwitch, and Infiniband topologies and arranges communication to maximize bandwidth and minimize latency. Some of its key strategies include:

- Ring All-Reduce: GPUs are arranged in a ring. Each GPU sends its data to the next while receiving from the previous, accumulating partial sums as the data flows. This scales well with many GPUs.
- Tree All-Reduce: Organizes communication as a tree, reducing depth (latency) at the cost of bandwidth.
- Hybrid schemes: NCCL dynamically chooses strategies depending on GPU count and topology.

By exploiting topology awareness, NCCL can saturate available communication channels.

Example in *llm.c* Training

In the CPU-only training loop, gradients are updated directly without communication. In the multi-GPU CUDA path, after backpropagation (`gpt2_backward`), each GPU has its local gradients in memory. At this point:

```
ncclAllReduce(model.grads_memory,
              model.grads_memory,
              model.num_parameters,
              ncclFloat32,  // or half precision
              ncclSum,
              comm, stream);
```

After this call, `model.grads_memory` on every GPU contains the summed gradients across all GPUs. Dividing by the number of GPUs turns it into the average.

Why Gradient Sync Matters

Without gradient synchronization, each GPU would drift apart, updating weights independently. This would be equivalent to training multiple smaller models rather than one unified model. Synchronization makes sure all replicas behave like a single large-batch training job.

Memory and Performance Considerations

1. Bandwidth-bound: Gradient synchronization often dominates runtime as model size grows. For GPT-2 774M, gradients alone can be several GB per step.
2. Overlapping Communication with Compute: Advanced systems overlap gradient exchange with backward computation. While later layers are computing gradients, earlier layers are already being synchronized.
3. Precision: Gradients can be synchronized in FP16/BF16 to cut communication bandwidth in half. This is called gradient compression.

Analogy

Think of four chefs cooking the same dish in separate kitchens. Each chef tastes their own version and suggests adjustments (gradients). If they don't talk, their recipes diverge. With All-Reduce, the chefs share notes, average their adjustments, and apply the same changes—so all four kitchens end up cooking the same dish.

Try It Yourself

1. Run training on 2 GPUs with and without gradient synchronization (by commenting out All-Reduce). Watch how quickly the models diverge in loss.
2. Use NCCL's `NCCL_DEBUG=INFO` environment variable to print communication patterns. Observe the chosen ring/tree strategies.
3. Experiment with FP32 vs FP16 gradient synchronization and measure bandwidth savings.
4. Profile training with `nsys` or `nvprof` to see how much time is spent in All-Reduce.
5. Scale from 2 GPUs to 4 or 8 and measure how synchronization overhead grows.

The Takeaway

NCCL All-Reduce is the backbone of multi-GPU training in *llm.c*. It ensures that gradients computed on separate GPUs are combined into a single, consistent update. By leveraging topology-aware algorithms like ring and tree reductions, NCCL keeps synchronization efficient even as models and GPU counts scale up. Without it, distributed training would produce inconsistent, drifting models rather than a unified one.

74. Building and Running Multi-GPU Trainers

Getting multiple GPUs to cooperate isn't automatic—you need to set up the environment, initialize communication, and make sure each process knows which GPU to use. In *llm.c*, the design is intentionally minimalist, but it still has to integrate with MPI (Message Passing Interface) and NCCL to allow training across several GPUs.

Step 1: MPI Launch

Multi-GPU training starts with MPI. You don't run the program once; you launch it with `mpirun` or `mpiexec`, which spawns one process per GPU. For example:

```
mpirun -np 4 ./train_gpt2_cu
```

Here, `-np 4` starts four processes. Each process will attach itself to one GPU.

MPI provides:

- Rank: a unique ID for each process (0, 1, 2, 3).
- World size: the total number of processes (here, 4).

Each process knows who it is and how many peers it has.

Step 2: GPU Assignment

Once MPI assigns ranks, each process must select a GPU. This is often done with:

```
cudaSetDevice(rank);
```

So process 0 gets GPU 0, process 1 gets GPU 1, and so on. Without this step, processes might all pile onto the same GPU, leading to chaos.

Step 3: NCCL Communicator

Next, the code creates an NCCL communicator. Think of it as a "conference call" between all GPUs. NCCL sets up the communication paths (rings, trees) across devices. A typical setup looks like:

```
ncclCommInitRank(&comm, world_size, nccl_id, rank);
```

Here:

- `world_size` is the number of GPUs.
- `nccl_id` is a shared identifier obtained via MPI (all processes must use the same one).
- `rank` is the local ID.

Now the GPUs can talk to each other.

Step 4: Training Loop Integration

Once communication is established, the training loop doesn't look dramatically different. Each GPU:

1. Loads its own batch of data (so the dataset is divided across GPUs).
2. Runs the forward pass.
3. Runs the backward pass.
4. Calls NCCL All-Reduce to sync gradients.
5. Updates parameters.

The only new ingredient is step 4. Without it, each GPU would wander off with its own gradients.

Example Command

Suppose you have 2 GPUs on your machine. You can train with:

```
mpirun -np 2 ./train_gpt2_cu -batch_size 8 -seq_len 128
```

Each GPU trains on 8 sequences of 128 tokens. Combined, it's like training with batch size 16, but split across GPUs.

Common Pitfalls

- Forgetting to set device by rank: all processes fight for GPU 0.
- Mismatched NCCL IDs: communicator fails to initialize.
- MPI vs NCCL versions: some builds are picky, and you may need to recompile with matching CUDA/NCCL.
- Networking issues: on multi-node setups, firewalls or missing InfiniBand drivers can block communication.

Why It Matters

Building a multi-GPU trainer is the gateway to scaling. A single GPU may take weeks to train a large model, but spreading the work across 4, 8, or 16 GPUs cuts the time dramatically. The simplicity of *llm.c* shows that distributed training doesn't require a massive framework—just careful use of MPI and NCCL.

Try It Yourself

1. Launch training with 1 GPU and then 2 GPUs, keeping the global batch size the same. Compare training speed.
2. Launch with 2 GPUs but forget All-Reduce. Notice how validation loss behaves differently on each GPU.
3. Use `NCCL_DEBUG=INFO` to see how NCCL sets up communication.
4. Try deliberately mismatching ranks and devices—observe the crash to understand why assignment matters.
5. Measure GPU utilization with `nvidia-smi` during training to confirm both GPUs are working.

The Takeaway

Multi-GPU trainers in *llm.c* are built around three pillars: MPI to manage processes, NCCL to synchronize gradients, and CUDA to run the math. Once these are in place, the training loop remains familiar, but the computation spreads across GPUs seamlessly. This design keeps the code minimal while still unlocking significant scaling power.

75. Multi-Node Bootstrapping with MPI

So far, running across multiple GPUs on a single machine is relatively straightforward: every process talks through shared memory or high-speed interconnects like NVLink. Things become more interesting when training has to scale across multiple machines (often called "nodes")—for example, when you want to run on 2 servers, each with 4 GPUs, to make a total of 8.

The MPI World

MPI was designed for this. When you run:

```
mpirun -np 8 -hostfile myhosts ./train_gpt2_cu
```

- `-np 8` says you want 8 processes.
- `-hostfile myhosts` lists the machines (and how many processes to run on each).

MPI then launches processes across nodes and assigns each one a rank. From the program's perspective, it doesn't matter if two ranks are on the same machine or different machines—they all see a global communicator of size 8.

Setting Up NCCL Across Nodes

NCCL doesn't know how to find other machines by itself. It relies on MPI to exchange a unique NCCL ID. The typical flow is:

1. Rank 0 creates a new NCCL ID.
2. Rank 0 broadcasts the ID to all other ranks using MPI.
3. Each process calls `ncclCommInitRank` with the shared ID, total world size, and its own rank.

This ensures all GPUs, even across different machines, join the same "conference call."

Networking Considerations

When scaling across nodes, networking becomes critical:

- Ethernet vs InfiniBand: Standard Ethernet works but can be slow. High-performance clusters use InfiniBand for much higher bandwidth and lower latency.
- Firewall rules: NCCL needs open ports to connect nodes. Firewalls or strict security settings can block communication.
- Environment variables: Variables like `NCCL_SOCKET_IFNAME` (to pick the right network interface) often need to be set. For example:

```
export NCCL_SOCKET_IFNAME=eth0
```

Example Hostfile

A simple `myhosts` file could look like this:

```
node1 slots=4
node2 slots=4
```

This says node1 and node2 each have 4 GPUs. MPI will launch 4 processes on each, totaling 8.

Synchronization Across Nodes

Because communication now spans machines, synchronization overhead becomes more visible. Gradient All-Reduce has to move data not only between GPUs in one server but also across the network. Efficient scaling depends on:

- Large enough batch sizes (so compute time outweighs communication).
- Overlapping communication with computation (advanced optimization).
- Fast interconnects between machines.

Why It Matters

Training large models rarely happens on a single machine. Multi-node training is how researchers and companies scale models to billions of parameters. By showing how to bootstrap MPI and NCCL across nodes, *llm.c* demonstrates the foundation of distributed AI training systems, but in a minimal and transparent way.

Try It Yourself

1. Prepare two machines with CUDA and NCCL installed, connected via the same network.
2. Write a hostfile listing both machines, then launch with `mpirun`.
3. Set `NCCL_DEBUG=INFO` to watch how NCCL connects across nodes.
4. Compare throughput between single-node and two-node runs with the same number of GPUs.
5. Experiment with environment variables like `NCCL_SOCKET_IFNAME` or `NCCL_IB_DISABLE=1` to see how network choices affect speed.

The Takeaway

Bootstrapping multi-node training is about extending the same principles as single-node multi-GPU training, but with networking in the mix. MPI handles process management, NCCL sets up communication, and CUDA runs the math. With just a few lines of setup, *llm.c* can stretch from one GPU on your laptop to dozens of GPUs spread across multiple servers.

76. SLURM and PMIx Caveats

On many research clusters or supercomputers, you don't launch jobs manually with `mpirun` and a hostfile. Instead, you interact with a job scheduler, most commonly SLURM. SLURM takes care of allocating resources, starting processes across nodes, and enforcing quotas. While this saves you from manually managing hostfiles, it introduces its own set of details that you need to understand.

SLURM Basics

In SLURM, you typically request GPUs and nodes using a script or command like:

```
salloc -N 2 -G 8 --time=01:00:00
```

- `-N 2` asks for 2 nodes.
- `-G 8` requests 8 GPUs (across those nodes).
- `--time=01:00:00` sets a one-hour time limit.

Once the job starts, SLURM sets environment variables such as:

- `SLURM_NTASKS`: total number of tasks (processes).
- `SLURM_PROCID`: the rank of the current process.
- `SLURM_NODEID`: which node this process is running on.

MPI implementations (OpenMPI, MPICH) and NCCL can use these to bootstrap communication automatically.

PMIx Integration

Modern SLURM often works with PMIx (Process Management Interface for Exascale). PMIx allows MPI and other runtimes to query process information directly from SLURM without relying on older launchers. In practice, this means:

- You might not use `mpirun` at all. Instead, SLURM provides `srun`.
- For example:

  ```
  srun -n 8 ./train_gpt2_cu
  ```

  Here `-n 8` launches 8 tasks across your allocated nodes. SLURM/PMIx handles the rank assignments.

Common Pitfalls

1. MPI version mismatch If your cluster has multiple MPI libraries installed, you may accidentally compile with one and run with another. Always confirm that the `mpicc` and `mpirun` you're using match the library your job is linking against.
2. Environment variable propagation NCCL relies on environment variables like `NCCL_DEBUG`, `NCCL_SOCKET_IFNAME`, and `NCCL_IB_HCA`. Sometimes SLURM doesn't forward these to all nodes unless you configure it to. Using `--export=ALL` or adding exports in your job script can fix this.
3. GPU visibility SLURM manages GPU allocation via `CUDA_VISIBLE_DEVICES`. Each process only "sees" the GPUs it was assigned. If your code assumes a global view of all GPUs, it may break. In *llm.c*, the mapping between rank and GPU ID needs to respect this.
4. Network fabric mismatches On big clusters, you may have multiple network fabrics (Ethernet, InfiniBand). If NCCL picks the wrong one, performance plummets. Explicitly setting `NCCL_SOCKET_IFNAME` or `NCCL_IB_DISABLE` can solve this.

Why It Matters

Learning to run across nodes with SLURM is essential if you want to scale training beyond a single server. While local `mpirun` commands work for development, almost all serious training runs—whether academic or industrial—happen under SLURM or a similar workload manager. Understanding the quirks of SLURM and PMIx ensures that your code scales smoothly without mysterious hangs or slowdowns.

Try It Yourself

1. Write a small SLURM job script that requests 2 GPUs for 10 minutes and runs a dummy *llm.c* training loop.
2. Use `srun` to launch the program and print out the `SLURM_PROCID` and `SLURM_NODEID` for each process.
3. Set `NCCL_DEBUG=INFO` in your job script and observe how NCCL initializes communication.
4. Experiment with `srun --ntasks-per-node` to control how many processes land on each node.
5. Intentionally misconfigure `CUDA_VISIBLE_DEVICES` to see how it affects rank-to-GPU mapping.

The Takeaway

SLURM and PMIx streamline distributed training on large clusters, but they add another layer of complexity. The principles remain the same—MPI ranks, NCCL communicators, and CUDA kernels—but the scheduler decides how processes are placed and how environments are set up. With a bit of practice, these tools allow *llm.c* to move from simple multi-GPU experiments to scalable cluster-wide training runs.

77. Debugging Multi-GPU Hangs and Stalls

When training on multiple GPUs, one of the most frustrating experiences is a job that simply hangs — no errors, no crashes, just frozen processes. In distributed deep learning, hangs are almost always related to synchronization mismatches. Every GPU worker is supposed to "meet up" at communication points (like gradient all-reduce), and if even one process gets lost, the whole group stalls.

Common Causes of Hangs

1. Mismatched Collective Calls If one rank calls `ncclAllReduce` while another rank skips it or calls `ncclBroadcast`, the system deadlocks. All GPUs wait forever because they're not speaking the same "language" at that step.
2. Uneven Batch Sizes If the training data isn't perfectly divisible across GPUs, one process might run out of data earlier than others. The code tries to sync gradients, but some ranks never reach that point.
3. CUDA Errors Silently Ignored A kernel launch failure on one GPU can prevent it from reaching synchronization. If error checks are missing, you won't see the failure until the program is stuck.
4. Networking Issues NCCL depends on reliable network connections. If one node has a bad InfiniBand card, firewall rule, or misconfigured interface, communication halts.

Debugging Strategies

- Enable NCCL Debugging Set:

  ```
  export NCCL_DEBUG=INFO
  export NCCL_DEBUG_SUBSYS=ALL
  ```

  This produces logs showing when each rank enters and leaves collective operations. By comparing ranks, you can see who got stuck.

- Check CUDA Errors Always wrap CUDA calls with error checks, or run with:

  ```
  cuda-memcheck ./train_gpt2_cu
  ```

  This detects invalid memory access or kernel failures that might lead to stalls.

- Simplify the Setup Start with 2 GPUs on a single node. If it works, increase to 4 GPUs, then expand to multiple nodes. This isolates whether the bug is in GPU logic or network communication.
- Timeouts and Watchdogs NCCL provides environment variables like `NCCL_TIMEOUT` that can help detect when a collective is stalled. Although it won't fix the hang, it prevents wasting hours waiting for nothing.

Why It Matters

In multi-GPU training, hangs are not rare — they are part of the debugging journey. Understanding that hangs usually mean "one rank is out of sync" helps you approach the problem methodically. By checking logs, validating batch sizes, and carefully testing collective calls, you avoid endless frustration and wasted GPU hours.

Try It Yourself

1. Run a 2-GPU training job and intentionally misconfigure the code so only one rank calls `gpt2_backward`. Observe how the system hangs.
2. Enable `NCCL_DEBUG=INFO` and compare logs between the two ranks.
3. Modify the dataloader so that one GPU gets fewer batches than the other. Watch how training stalls at the first gradient sync.
4. Experiment with `cuda-memcheck` to catch silent CUDA errors in a simple kernel.
5. Practice scaling up from 1 node to 2 nodes to see where hangs are more likely to appear.

The Takeaway

Hangs in distributed training almost always trace back to mismatched synchronization, unbalanced workloads, or hidden errors. By using NCCL's debug tools, adding error checks, and testing systematically, you can turn mysterious freezes into solvable problems. Multi-GPU training isn't just about raw speed — it's about learning to keep many moving parts in lockstep.

78. Scaling Stories: GPT-2 124M → 774M → 1.6B

One of the most exciting parts of *llm.c* is that it doesn't just stop at toy models. The same code that trains a small GPT-2 model on Tiny Shakespeare can be scaled up to much larger models, like GPT-2 774M and even 1.6B. But scaling isn't just about making the numbers bigger — it changes almost everything about how you train: memory requirements, communication costs, optimizer stability, and even your workflow.

Starting Small: GPT-2 124M

The 124M parameter model is the "hello world" of GPT-2 training. It fits comfortably on a single modern GPU, and you can even run a trimmed-down version on CPU. At this size:

- Batch sizes can stay small (e.g., 4–8).
- Memory requirements are modest — a few gigabytes of VRAM.

- Training speed is relatively fast, so you can iterate quickly.
- Purpose: sanity checks, debugging kernels, verifying correctness.

Think of 124M as the training wheels stage: you're learning to balance, not yet racing.

Moving to GPT-2 774M

At ~774M parameters, the picture changes:

- A single GPU can still *fit* the model, but training speed slows dramatically.
- Gradient synchronization across multiple GPUs becomes essential to get reasonable throughput.
- Communication costs start to matter: an all-reduce of hundreds of megabytes per step stresses both PCIe and network bandwidth.
- Stability becomes more sensitive: learning rates and warmup schedules need more careful tuning.

Here, training is less about "does the code run?" and more about "does the system scale?" This size is often used in academic replications of GPT-2 because it's large enough to be interesting but not impossibly expensive.

GPT-2 1.6B: Scaling to the Edge

At 1.6B parameters, the model is too large for a single GPU to train efficiently. You need:

- Multi-GPU setups with NCCL all-reduce to share gradient updates.
- Multi-node training on clusters when even 8 GPUs aren't enough.
- Careful optimizer tuning — without proper settings for AdamW and schedulers, the model may diverge.
- Memory tricks like mixed precision (FP16/BF16) and gradient checkpointing to fit activations in memory.

Training GPT-2 1.6B is a significant engineering challenge, but it proves that *llm.c* is not just a toy project — it's a minimal yet real implementation that can push to billion-parameter scale.

Scaling Lessons

As you climb from 124M to 774M to 1.6B, several lessons emerge:

1. Debug small, scale big — always test on 124M before attempting larger models.

2. Communication dominates — at 774M and beyond, time spent moving gradients often exceeds compute time.
3. Hyperparameters evolve — a learning rate that works for 124M may explode the loss at 1.6B.
4. Infrastructure matters — GPUs, interconnects, and schedulers become as important as code.

Why It Matters

Scaling stories show that deep learning isn't just about writing a clever algorithm — it's about making that algorithm work under increasingly heavy loads. Each jump in size uncovers new bottlenecks and new engineering challenges. By following this path, you gain intuition for how large models are really trained in practice.

Try It Yourself

1. Train GPT-2 124M on Tiny Shakespeare until the loss stabilizes. Record how long each step takes.
2. Attempt the same experiment on OpenWebText with 124M — watch how dataset size now becomes the limiting factor.
3. Scale up to GPT-2 355M or 774M if you have access to multiple GPUs. Measure how much time is spent in NCCL all-reduce compared to compute.
4. If you have cluster access, try running 774M with `srun` or `mpirun` across nodes.
5. Study published training logs for GPT-2 1.6B and compare them to your own — how does scaling change the shape of the loss curve?

The Takeaway

Scaling isn't just "bigger models need bigger GPUs." Each increase in model size reshapes the training process, introducing new bottlenecks and requiring new techniques. *llm.c* is valuable precisely because it makes these transitions transparent: you can start with a tiny model and gradually experience the real engineering hurdles of training state-of-the-art language models.

79. NCCL Tuning and Overlap Opportunities

Once your training runs extend beyond a single GPU, communication overhead becomes a central challenge. Every training step requires gradients to be exchanged among GPUs so that the optimizer updates stay synchronized. This is where NCCL (NVIDIA Collective Communications Library) comes in. NCCL provides efficient implementations of collective operations like all-reduce, all-gather, and broadcast. But simply using NCCL isn't enough: how

you tune it, and how you overlap communication with computation, can make the difference between sluggish training and near-linear scaling.

How NCCL Works in Training

In *llm.c*, when multiple GPUs train together, each GPU computes its local gradients during backpropagation. At the end of the backward pass, NCCL's all-reduce combines gradients across GPUs so that every GPU ends up with the same values. Only then can the optimizer step forward.

Without NCCL, you'd have to write custom point-to-point code with `cudaMemcpyPeer` or MPI, which would be both slower and harder to maintain. NCCL ensures the communication pattern is efficient for the underlying hardware — PCIe, NVLink, or InfiniBand.

Key NCCL Tuning Parameters

1. NCCL_DEBUG Setting `NCCL_DEBUG=INFO` helps you understand what NCCL is doing. For performance tuning, logs are essential.
2. NCCL_SOCKET_IFNAME On multi-node clusters, this decides which network interface NCCL binds to. Using the wrong interface (like Ethernet instead of InfiniBand) can slow training by orders of magnitude.
3. NCCL_ALGO Determines how collectives are executed:
    - *Ring*: good for large message sizes, stable performance.
    - *Tree*: faster for small messages, less latency. Some training runs benefit from experimenting with both.
4. NCCL_IB_DISABLE Useful if you want to force NCCL to avoid InfiniBand and stick with TCP/IP, usually for debugging network issues.

Overlapping Communication with Computation

The backward pass doesn't need to wait until all gradients are computed before starting to communicate. In fact, gradients for earlier layers can start their all-reduce while later layers are still computing gradients. This is called communication-computation overlap.

For example:

- Without overlap: compute gradients for all layers → run NCCL all-reduce for all gradients → update parameters.
- With overlap: while gradients for higher layers are still being computed, start the all-reduce for earlier layers.

This reduces idle time and often leads to substantial throughput gains. Some frameworks (like PyTorch's DistributedDataParallel) implement this automatically. In a low-level system like *llm.c*, this would require careful kernel launch ordering and stream management.

Practical Example

Imagine you're training GPT-2 774M across 8 GPUs. Each backward pass produces ~3 GB of gradients. If you wait until all gradients are ready before syncing, the all-reduce might take 200 ms. If your compute step also takes 200 ms, then half of your training time is spent idle. With overlap, you can hide much of that communication inside compute time, potentially cutting step time almost in half.

Why It Matters

As model sizes increase, communication costs can rival or exceed computation. Without tuning, GPUs spend more time waiting for data to arrive than actually training the model. By understanding NCCL and applying overlap techniques, you unlock the ability to scale efficiently to dozens or even hundreds of GPUs.

Try It Yourself

1. Run a multi-GPU training job with `NCCL_DEBUG=INFO` enabled and watch the communication patterns.
2. Change `NCCL_ALGO` between `Ring` and `Tree` and measure the effect on step times.
3. Experiment with setting `CUDA_LAUNCH_BLOCKING=1` to remove overlap, then remove it again to see how communication and computation interleave.
4. If you have a cluster, try forcing NCCL to use Ethernet instead of InfiniBand and compare bandwidth.
5. Profile a multi-GPU run using `nvprof` or Nsight Systems and check whether NCCL collectives overlap with kernel execution.

The Takeaway

Efficient distributed training is not only about having more GPUs — it's about keeping them busy. NCCL provides the communication backbone, but how you configure and overlap its operations determines whether you get close to linear scaling or waste resources. Mastering these details transforms multi-GPU training from "just working" into truly efficient large-scale computation.

80. Common Multi-GPU Errors and Fixes

When running *llm.c* across multiple GPUs, errors can range from confusing hangs to cryptic NCCL messages. These problems are normal in distributed training, but they can eat up hours unless you recognize the patterns. The good news is that most errors fall into a handful of common categories, and once you learn the typical causes, they're easier to diagnose and fix.

Error Type 1: Process Hangs with No Output

Symptom: Training starts but then freezes. No error message, no crash, just silence. Cause: Usually, one or more ranks are out of sync. This could mean:

- Different batch sizes on each rank (one rank has fewer tokens left).
- A mismatch in collective calls — for example, one GPU calls `all_reduce` while another skips it.
- A CUDA error in one process that prevents it from reaching synchronization. Fix:
- Check that dataloaders feed the same number of steps to every rank.
- Add error checking to CUDA calls.
- Enable `NCCL_DEBUG=INFO` to trace which rank got stuck.

Error Type 2: `NCCL WARN Net: no interface found`

Symptom: NCCL reports it can't find a network interface, or training is extremely slow. Cause: NCCL can't discover the correct interface to use for inter-node communication. By default, it may try Ethernet instead of InfiniBand. Fix:

- Set `NCCL_SOCKET_IFNAME` to the right interface, e.g.:

  ```
  export NCCL_SOCKET_IFNAME=ib0
  ```

- Confirm with your sysadmin which network interfaces are available for high-performance GPU communication.

Error Type 3: `CUDA_ERROR_OUT_OF_MEMORY`

Symptom: Processes crash when allocating model weights or during the backward pass. Cause:

- Model too large for the available GPU memory.
- Batch size too high.
- Memory fragmentation from repeated allocations. Fix:
- Reduce batch size `B` or sequence length `T`.

- Try mixed precision (FP16/BF16) if supported.
- Restart processes to clear memory fragmentation.

Error Type 4: `unhandled system error, NCCL version mismatch`

Symptom: One process logs NCCL version `2.17` and another logs `2.14`. Training fails. Cause: Different NCCL libraries are being used across nodes. This happens when software environments aren't identical. Fix:

- Use the same container or module environment on all nodes.
- Confirm NCCL versions with `ldd` or `conda list`.

Error Type 5: Validation Loss Diverges on Multi-GPU but Not on Single GPU

Symptom: Loss values explode only when running across multiple GPUs. Cause: Gradient synchronization may be broken — for example, only a subset of parameters are being all-reduced. Another possibility is using a different effective learning rate because of batch size scaling. Fix:

- Confirm that all parameters participate in gradient synchronization.
- Scale the learning rate properly: if you double the global batch size by using more GPUs, you may need to adjust the learning rate.

Why It Matters

Multi-GPU training is powerful but unforgiving: even tiny mismatches in environment, data, or synchronization can cause errors. Instead of treating these as random mysteries, it's helpful to recognize the patterns. Each error message or hang has a likely cause, and learning to map symptoms to fixes will make distributed training much smoother.

Try It Yourself

1. Intentionally reduce the dataset size so one rank runs out of data early — observe the hang.
2. Launch a multi-node run without setting `NCCL_SOCKET_IFNAME` and watch how performance collapses.
3. Increase the batch size until you trigger `CUDA_ERROR_OUT_OF_MEMORY`, then reduce it step by step to see the limit.
4. Experiment with mismatched NCCL versions across nodes if you have access to multiple environments, then fix it by standardizing.

5. Run a small model with different batch sizes per rank and study how the validation loss diverges.

The Takeaway

Most multi-GPU errors aren't mysterious once you understand what's happening under the hood. They usually boil down to synchronization mismatches, network misconfigurations, memory limits, or environment inconsistencies. With the right debugging tools and a systematic mindset, you can fix these problems quickly and keep your training runs moving forward.

Chapter 9. Extending the codebase

81. The `dev/cuda` Library for Custom Kernels

So far, most of the CUDA logic in *llm.c* has relied on NVIDIA's optimized libraries like cuBLAS and cuDNN. These libraries are extremely powerful and efficient, but sometimes you want more control: maybe you're experimenting with a new attention mechanism, or maybe you want to fuse multiple operations into a single kernel to reduce memory traffic. That's where the `dev/cuda` directory comes in. It's the playground for custom kernels.

What Lives in `dev/cuda`

If you look at the repository structure, you'll notice a folder named `dev/cuda`. This is not part of the minimal training path — you can train GPT-2 models without ever touching it. Instead, it contains experimental kernels that showcase how to move from simple CUDA examples toward more advanced, production-level implementations.

Inside, you'll typically find:

- Hello World kernels: basic examples like elementwise addition to get familiar with CUDA.
- Fused operations: simple prototypes for combining steps like bias addition + activation.
- Benchmark code: small programs that measure kernel performance compared to cuBLAS/cuDNN.

These files are not polished production code. They're meant to be read, modified, and played with — like lab notebooks for CUDA development.

Why Custom Kernels Matter

Libraries like cuBLAS are designed to cover a wide range of use cases, but they don't always hit the sweet spot for your specific workload. Writing custom kernels allows you to:

- Fuse operations: Instead of launching separate kernels for bias addition, activation, and dropout, you can do all three in one kernel, saving time on memory reads/writes.
- Experiment with new algorithms: If you invent a new type of attention or normalization, you can't rely on cuDNN — you need to implement it yourself.
- Learn how GPUs actually work: Reading and writing custom kernels teaches you about thread blocks, memory hierarchy, and warp scheduling, all of which deepen your understanding of GPU programming.

Example: A Simple Elementwise Kernel

Here's a very small kernel from a toy example you might find in `dev/cuda`:

```
__global__ void add_kernel(const float* a, const float* b, float* out, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        out[i] = a[i] + b[i];
    }
}
```

This kernel adds two arrays `a` and `b` elementwise. While trivial compared to attention mechanisms, it illustrates the GPU execution model:

- Each thread handles one index `i`.
- Threads are grouped into blocks, and blocks form a grid.
- Memory access is explicit: you control exactly how `out` is written.

Scaling this up to real workloads means adding more complexity — shared memory, warp shuffles, half-precision math — but the principles remain the same.

Why It Matters

The `dev/cuda` directory is not just for fun experiments. It's a bridge between "using GPU libraries" and "designing GPU algorithms." By learning to write kernels here, you gain the freedom to customize and optimize beyond what standard libraries provide. That skill becomes essential if you want to innovate in model architectures or squeeze the last bit of performance out of your hardware.

Try It Yourself

1. Open one of the example `.cu` files in `dev/cuda` and compile it with `nvcc`.
2. Modify the elementwise kernel so it performs `out[i] = a[i] * b[i]` instead of addition.
3. Benchmark your kernel against the equivalent cuBLAS call (e.g., `cublasSaxpy`) and compare performance.
4. Write a fused kernel that does bias addition followed by ReLU activation in one pass.
5. Use `nvprof` or Nsight Systems to measure how many kernel launches occur in a forward pass and imagine how custom fusion might reduce them.

The Takeaway

The `dev/cuda` library is your sandbox for learning and experimenting with CUDA. It's not required for running GPT-2, but it's where you build the skills to go beyond libraries and design your own GPU operations. Whether you're optimizing for speed or testing new research ideas, this directory is where theory meets practice in GPU programming.

82. Adding New Dataset Pipelines (`dev/data/*`)

Training a language model is not just about having a clever model — it's equally about the data you feed it. In *llm.c*, datasets are handled in a very lightweight way compared to frameworks like PyTorch. Instead of complicated abstractions, the project keeps things simple: tokenize your text once, save it as a `.bin` file, and then stream batches of tokens into the model.

The `dev/data/` directory is where this happens. It contains scripts and utilities for preparing different datasets, from tiny toy corpora like Tiny Shakespeare to larger collections like TinyStories or OpenWebText subsets. Understanding how this directory works is the key to plugging in your own datasets.

How Data Pipelines Work in *llm.c*

At a high level, the pipeline follows three steps:

1. Download or provide raw text data. For example, `tiny_shakespeare.txt` is just a plain text file with plays concatenated.
2. Tokenize the data once using the GPT-2 tokenizer. The tokenizer converts text into integers according to `gpt2_tokenizer.bin`.
3. Write the tokens to a binary file (`.bin`). This is a flat array of integers stored as 32-bit values, which makes it fast to memory-map and stream during training.

Once the `.bin` files exist, `dataloader_init` can open them, divide them into training and validation splits, and generate batches of shape `(B, T)` for the model.

What's Inside `dev/data/`

The folder contains small scripts like:

- `download_starter_pack.sh` — downloads Tiny Shakespeare and TinyStories.
- Tokenization scripts — often small Python snippets that run the GPT-2 tokenizer over raw text.
- Prebuilt `.bin` files — these are used in the quickstart so you don't need to regenerate them yourself.

The design choice here is minimalism: instead of a heavy dataset framework, you get plain files and short scripts. You can read and understand everything in a few minutes.

Adding Your Own Dataset

Suppose you want to train on your company's support chat logs or a new dataset you've found. The process looks like this:

1. Prepare raw text in a simple format (one text file is fine).
2. Run the tokenizer from *llm.c* on it:

   ```
   python dev/data/tokenizer.py --input my_corpus.txt --output my_corpus.bin
   ```

   This produces a binary token file.
3. Drop the file into `dev/data/`. You might name it `my_corpus_train.bin` and `my_corpus_val.bin`.
4. Point the dataloader at it in your training code:

   ```
   dataloader_init(&train_loader, "dev/data/my_corpus_train.bin", B, T, 0, 1, 1);
   dataloader_init(&val_loader, "dev/data/my_corpus_val.bin", B, T, 0, 1, 0);
   ```

That's it — you now have a new dataset pipeline integrated with the same training loop.

Why It Matters

Many frameworks hide data preprocessing behind layers of abstractions. *llm.c* takes the opposite approach: it makes the process transparent. You see exactly how text becomes tokens, how those tokens become batches, and how the model consumes them. This transparency makes it easier to debug, extend, and customize. Adding a new dataset is no longer a mystery — it's just a matter of writing one file and updating a path.

Try It Yourself

1. Explore the `dev/data/` directory and read through the provided scripts.
2. Tokenize a new small dataset of your choice (a novel, a set of Wikipedia pages, or your own text).
3. Train a 124M model on your new dataset and observe the loss curve.
4. Compare validation loss between Tiny Shakespeare and your dataset — how does the model behave differently?
5. Try increasing sequence length `T` to see how batching interacts with longer documents.

The Takeaway

The `dev/data` folder is where you connect language models to the real world. It shows how raw text becomes training-ready binary files with almost no overhead. By learning to add your own pipelines, you gain the ability to train *llm.c* on any dataset — from classic literature to domain-specific corpora — while keeping the workflow fast and understandable.

83. Adding a New Optimizer to the Codebase

So far, *llm.c* has focused on AdamW, which is the workhorse optimizer for training transformer models. But deep learning is a fast-moving field: new optimizers appear, older ones sometimes resurface, and certain workloads benefit from alternatives. The simplicity of *llm.c* makes it a great environment to learn how to implement and experiment with optimizers.

Where Optimizers Live in *llm.c*

In the CPU training path, the optimizer logic is implemented directly in C in the function `gpt2_update`. This function iterates through every parameter, applies AdamW's moment updates, applies bias correction, and then modifies the parameter values in place.

Because the parameters, gradients, and optimizer states are all stored as contiguous arrays in memory (`params_memory`, `grads_memory`, `m_memory`, `v_memory`), adding a new optimizer usually means:

1. Allocating any new state arrays you need.
2. Defining the update rule in the training loop.
3. Adding function calls for your new optimizer, similar to `gpt2_update`.

Example: Implementing SGD with Momentum

Stochastic Gradient Descent (SGD) with momentum is much simpler than AdamW. The update rule looks like this:

```
// SGD with momentum update
void gpt2_update_sgd(GPT2 *model, float learning_rate, float momentum) {
    if (model->m_memory == NULL) {
        model->m_memory = (float*)calloc(model->num_parameters, sizeof(float));
    }

    for (size_t i = 0; i < model->num_parameters; i++) {
        float grad = model->grads_memory[i];
        float m = momentum * model->m_memory[i] + grad;
        model->m_memory[i] = m;
        model->params_memory[i] -= learning_rate * m;
    }
}
```

Here, `m_memory` stores the velocity (the exponentially decayed average of past gradients). There's no second moment estimate like in AdamW, so it's leaner in both code and memory usage.

Comparing Optimizers

Adding new optimizers lets you experiment and compare behaviors:

Optimizer	Strengths	Weaknesses	Memory Needs
SGD	Simple, stable, fewer hyperparameters	Slow convergence on large models	Low
SGD + Momentum	Faster convergence, smooths updates	Still less adaptive than Adam	Low
Adam	Adapts learning rates per parameter	Can overfit small datasets	Medium
AdamW	Same as Adam + correct weight decay	More complex	Medium
Adagrad/RMSProp	Good for sparse features	Not always stable for transformers	Medium

In *llm.c*, each optimizer is just a loop over parameters with a few math operations. That makes it the perfect playground to see how different optimizers actually behave in practice.

Why It Matters

Optimizers control how your model learns. While architectures like GPT-2 get a lot of attention, small changes in optimization can make the difference between a model that converges smoothly and one that diverges. By adding your own optimizers, you get a clearer understanding of this often "black box" part of deep learning.

Try It Yourself

1. Implement the SGD with momentum function shown above and swap it into the training loop instead of AdamW.
2. Run training on Tiny Shakespeare and compare how many steps it takes for the loss to reach 2.0.
3. Modify the code to implement RMSProp (similar to Adam, but without momentum on the first moment).
4. Benchmark memory usage: notice how AdamW allocates both `m_memory` and `v_memory`, while SGD only uses one.
5. Try running AdamW with a very small dataset versus SGD — does one overfit faster?

The Takeaway

Optimizers are just math on arrays. By writing and testing new ones inside *llm.c*, you'll demystify how learning actually happens at the parameter level. This makes it easier to appreciate why AdamW became the default for transformers, and also gives you the tools to explore alternatives in a clean, transparent environment.

84. Adding a New Scheduler (cosine, step, etc.)

Training isn't only about choosing an optimizer; the way you adjust the learning rate over time is just as important. A scheduler tells the optimizer *how fast to learn* at each step. Without a scheduler, you'd use a fixed learning rate, which often works poorly for large models. In *llm.c*, schedulers are kept intentionally simple so you can see exactly how they influence training.

Where Schedulers Fit

If you look at the training loop, every step calls the optimizer like this:

```
gpt2_update(&model, lr, beta1, beta2, eps, weight_decay, step+1);
```

The `lr` here doesn't have to be constant. Instead, a scheduler function can compute it based on the step number. In *llm.c*, this logic lives in `schedulers.h` and related helpers.

Common Schedulers You Can Add

1. Step Decay Reduce the learning rate by a fixed factor every N steps.

```
float step_decay(int step, float base_lr, int decay_every, float decay_factor) {
    int k = step / decay_every;
    return base_lr * powf(decay_factor, k);
}
```

2. Cosine Decay Smoothly decrease the learning rate following a cosine curve.

```
float cosine_decay(int step, int max_steps, float base_lr) {
    float progress = (float)step / max_steps;
    return base_lr * 0.5f * (1.0f + cosf(M_PI * progress));
}
```

3. Linear Warmup + Cosine Decay Start with a gradual increase (warmup) to avoid instability, then switch to cosine decay. This is the most common choice for transformers.

Example: Cosine with Warmup

Here's how you might implement cosine with warmup in *llm.c*:

```
float lr_scheduler(int step, int warmup_steps, int max_steps, float base_lr) {
    if (step < warmup_steps) {
        return base_lr * (float)(step + 1) / warmup_steps;
    } else {
        float progress = (float)(step - warmup_steps) / (max_steps - warmup_steps);
        return base_lr * 0.5f * (1.0f + cosf(M_PI * progress));
    }
}
```

This means:

- Steps 0–`warmup_steps`: linearly scale from 0 → `base_lr`.
- After warmup: smoothly decay the learning rate using cosine.

Why It Matters

Schedulers help stabilize training. At the beginning, gradients can be very noisy, so warming up slowly prevents divergence. At the end, lowering the learning rate helps the model converge instead of bouncing around minima. Without schedulers, you'd often need more tuning to get the same results.

Try It Yourself

1. Train with a constant learning rate on Tiny Shakespeare and record the loss curve.
2. Switch to a step decay scheduler and see if convergence improves.
3. Implement cosine decay with warmup and compare against constant LR — which reaches a lower validation loss?
4. Experiment with different warmup lengths (e.g., 10 steps vs 100 steps) and watch how training stability changes.
5. Try running the same experiment on TinyStories and see if dataset size affects which scheduler works best.

The Takeaway

Schedulers are small pieces of code with big impact. They don't change the model or the optimizer, but they control the *pace of learning*. By adding new schedulers to *llm.c*, you get a hands-on way to see why modern training recipes almost always combine warmup with a smooth decay schedule.

85. Alternative Attention Mechanisms

Transformers became famous because of the self-attention mechanism, but "attention" is not a single fixed formula. Researchers have explored many alternatives that trade off memory use, speed, and accuracy. In *llm.c*, the default is scaled dot-product attention, but nothing prevents you from experimenting with new approaches.

The Default: Scaled Dot-Product Attention

The standard attention formula looks like this:

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

- Q = queries

- K = keys
- V = values
- d_k = key dimension (used for scaling)

In *llm.c*, this is implemented using matrix multiplications and masking to enforce causality. It's correct and faithful to GPT-2, but has quadratic cost in sequence length T.

Variants You Could Add

1. Sparse Attention Instead of attending to every token, restrict attention to a local window or a set of important positions.

   - Good for long sequences.
   - Saves compute and memory.
   - Example: "sliding window" attention where each token only looks back 128 steps.

2. Linformer / Low-Rank Attention Approximate QK^T using low-rank projections.

   - Reduces memory from $O(T^2)$ to $O(T)$.
   - Works well when redundancy exists in sequences.

3. Performer (Linear Attention) Replace the softmax with kernel approximations so attention becomes linear in sequence length.

   - Trades exactness for scalability.
   - Allows much longer sequences on the same hardware.

4. ALiBi (Attention with Linear Biases) Adds simple position-dependent biases instead of full positional embeddings.

   - Extremely efficient.
   - Helps extrapolate to longer sequences than seen in training.

How to Experiment in *llm.c*

The attention implementation lives inside the `attention_forward` and `attention_backward` routines in `train_gpt2.c` (and their CUDA equivalents). To try an alternative:

- Replace the part where QK^T is computed with your chosen method.
- Keep the interface the same: given inputs Q, K, V, return outputs shaped `(B, T, C)`.
- Run unit tests (`test_gpt2.c`) against the baseline to make sure the outputs stay reasonable.

Why It Matters

Attention is often the bottleneck in transformers. Quadratic time and memory usage limit how long your sequences can be. By experimenting with alternatives, you not only improve efficiency but also learn how new research ideas are implemented in practice. Many of today's "efficient transformers" came from simple tweaks to this block.

Try It Yourself

1. Modify attention so each token only attends to the last 16 tokens (a toy form of sparse attention). Train on Tiny Shakespeare and compare speed vs. accuracy.
2. Implement ALiBi by adding linear position-dependent bias terms and see if your model generalizes better to longer text.
3. Benchmark the GPU memory footprint of standard attention vs. your custom version using Nsight Systems or `nvidia-smi`.
4. Try removing the scaling factor $1/\sqrt{d_k}$ — does training become unstable?
5. Replace softmax with a simple ReLU and see how the model behaves (hint: it usually diverges, but it teaches why softmax is important).

The Takeaway

The attention block is where much of the magic happens in transformers, but it's also the biggest bottleneck. By experimenting with alternatives in *llm.c*, you'll gain a deeper understanding of why the standard formula works, what its weaknesses are, and how new ideas from research can be tested directly in code.

86. Profiling and Testing New Kernels

When you start adding custom CUDA kernels or experimenting with new attention mechanisms, the next big question is: how do you know if they're correct and efficient? That's where profiling and testing come in. *llm.c* keeps this process minimal but transparent so you can see exactly how to validate both correctness and performance.

Correctness First: Testing Against a Reference

Any new kernel you write should be compared against a known-good implementation. In *llm.c*, PyTorch usually serves as the "oracle." For example:

1. Generate random input tensors in both *llm.c* and PyTorch.
2. Run your custom kernel in *llm.c*.

3. Run the equivalent operation in PyTorch.
4. Compare the outputs within a small tolerance (e.g., differences less than `1e-5`).

This ensures your kernel doesn't silently compute the wrong thing. Without this step, you might train for hours before realizing your model is learning nonsense.

Performance: Profiling the Kernels

Once correctness is established, the next step is performance. NVIDIA provides several tools:

- nvprof (older): still widely used, easy to launch.
- Nsight Systems / Nsight Compute (modern): more detailed, lets you see kernel timings, memory transfers, occupancy, and more.

In practice:

- Run your training loop with profiling enabled.
- Identify which kernels take the most time.
- Check if your custom kernel is faster than the baseline (e.g., cuBLAS or cuDNN).

Common Metrics to Watch

- Kernel time (how long each launch takes).
- Occupancy (how many CUDA cores are active relative to maximum).
- Memory throughput (are you saturating memory bandwidth?).
- Launch count (do you call your kernel too many times instead of fusing operations?).

Even a correct kernel can be slower than a library implementation if it doesn't use the GPU efficiently.

Example Workflow

Suppose you write a fused bias + ReLU kernel. You can test it like this:

- Generate a random tensor in C and in PyTorch.
- Apply your fused kernel vs. PyTorch's separate `+` and `ReLU` ops.
- Compare results for correctness.
- Profile both approaches: is your kernel faster? Did it reduce kernel launch overhead?

Why It Matters

Custom kernels are fun to write, but without testing and profiling they're just guesswork. Many research ideas look promising in theory but fall apart in practice because they run slower or break correctness. By learning to systematically test and profile, you can separate ideas that are genuinely useful from those that are just experiments.

Try It Yourself

1. Write a simple fused kernel for bias + ReLU. Compare it against PyTorch's `x + bias` followed by `relu(x)`.
2. Use `nvprof` to check how many kernel launches happen in each version.
3. Run Nsight Systems and look at the timeline: do you see your fused kernel overlapping better with other GPU activity?
4. Try scaling sequence length `T` to very large values — does your kernel still perform well?
5. Record memory usage before and after your kernel runs. Is there a difference compared to the unfused version?

The Takeaway

Profiling and testing turn kernel hacking from random tinkering into real engineering. With a reference for correctness and tools for performance measurement, you can iterate confidently, knowing when your changes are truly improvements. This is how *llm.c* bridges the gap between a learning project and real GPU systems work.

87. Using PyTorch Reference as Oracle

One of the guiding principles of *llm.c* is to stay small, readable, and minimal — but that doesn't mean you're left without a safety net. When implementing something as mathematically dense as a transformer, how do you know your C or CUDA code is doing the right thing? The answer is to use PyTorch as a reference implementation, often called an "oracle."

What Does "Oracle" Mean Here?

An oracle is simply a trusted system you compare against. PyTorch is trusted because:

- It's widely used in production and research.
- Its operators (matrix multiply, attention, layernorm, etc.) have been thoroughly tested.
- It gives you both CPU and GPU implementations with stable numerical behavior.

If your *llm.c* forward or backward pass matches PyTorch's within a small error tolerance, you can be confident that your implementation is correct.

How the Comparison Works

The workflow usually looks like this:

1. Set up the same model in both PyTorch and *llm.c* with identical weights.
2. Feed the same inputs to both models.
3. Compare outputs — logits, losses, or gradients.
4. Allow for tiny differences due to floating point arithmetic, usually within `1e-5` to `1e-6`.

For example, `test_gpt2.c` in the repository runs a forward pass in C and compares the logits to those produced by a PyTorch GPT-2 checkpoint.

Example

Suppose you're testing the embedding lookup. In PyTorch you might write:

```
import torch
from transformers import GPT2Model, GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2Model.from_pretrained("gpt2")
tokens = torch.tensor([[50256, 200]])  # EOT and an example token
out = model.wte(tokens)  # word token embeddings
print(out.detach().numpy())
```

In *llm.c*, you'd run the same two tokens through the embedding lookup and compare the resulting vectors element by element. If they match, your embedding implementation is correct.

Why PyTorch is the Perfect Oracle

- Transparency: it's easy to extract weights from PyTorch checkpoints (`.bin` files).
- Flexibility: you can test individual layers, not just the whole model.
- Debuggability: if something goes wrong, you can isolate which layer diverges first.

This last point is crucial — instead of training for days only to find your loss curve diverges, you can catch mismatches immediately at the layer level.

Why It Matters

Deep learning models are fragile. Even a tiny mistake in normalization, masking, or gradient flow can ruin training. By anchoring your work to PyTorch, you avoid "trusting your gut" and instead rely on a battle-tested baseline. This practice isn't unique to *llm.c* — many professional frameworks (Megatron-LM, DeepSpeed) also validate against PyTorch during development.

Try It Yourself

1. Extract GPT-2 weights from Hugging Face Transformers and load them into *llm.c*.
2. Run a forward pass in both PyTorch and *llm.c* with the same input tokens. Compare outputs numerically.
3. Focus on a single block: check whether the attention output matches PyTorch's.
4. Modify a kernel (e.g., change softmax to ReLU) and watch how quickly the outputs diverge from PyTorch's.
5. Use PyTorch to verify gradients by calling `.backward()` and comparing with `gpt2_backward` in *llm.c*.

The Takeaway

PyTorch is your map and compass when navigating the dense jungle of transformer internals. By treating it as an oracle, you can move confidently from one layer to the next, knowing that your small, hand-written code matches the behavior of a full-featured deep learning framework. This practice turns *llm.c* into more than a toy project — it becomes a faithful, verifiable reimplementation of GPT-2.

88. Exploring Beyond GPT-2: LLaMA Example

While *llm.c* focuses on GPT-2 for clarity, the same framework can extend to newer, larger, and more modern models such as LLaMA. LLaMA, released by Meta, uses many of the same building blocks as GPT-2 — embeddings, attention layers, MLPs, normalization, and residual streams — but with tweaks that improve efficiency and scaling. Looking at LLaMA through the lens of *llm.c* helps you see how language model designs evolve while still sharing the same DNA.

What Stays the Same

- Token embeddings: both GPT-2 and LLaMA use lookup tables to turn token IDs into dense vectors.
- Transformer blocks: the fundamental loop of attention → MLP → residuals is unchanged.

- Autoregressive training: predict the next token given all previous ones, using causal masking.

This means much of the code in *llm.c* — dataloaders, embeddings, forward loops — would work with LLaMA almost unchanged.

What's Different

1. Normalization
    - GPT-2 uses LayerNorm before each block output.
    - LLaMA uses RMSNorm, which normalizes using only the root mean square of activations (no mean subtraction).
    - This reduces compute slightly and improves stability.
2. Positional Encoding
    - GPT-2 has learned positional embeddings.
    - LLaMA uses Rotary Position Embeddings (RoPE), which rotate queries and keys in attention space to encode positions.
    - RoPE scales better to longer contexts.
3. Vocabulary
    - GPT-2's vocab size is 50,257.
    - LLaMA uses a different tokenizer (SentencePiece/BPE) with a larger vocabulary, closer to 32k for LLaMA-2.
4. Model Scale
    - GPT-2 tops out at 1.6B parameters.
    - LLaMA-2 and LLaMA-3 scale from 7B up to 70B+. This makes distributed training mandatory, with mixed precision and checkpointing as standard.

Adapting *llm.c* to LLaMA

If you wanted to modify *llm.c* to approximate LLaMA, the main tasks would be:

- Replace LayerNorm with an RMSNorm implementation.
- Add RoPE into the attention mechanism. This means modifying the step where Q and K vectors are built, applying a rotation based on token positions.
- Swap out the GPT-2 tokenizer with a SentencePiece tokenizer trained on the desired vocabulary.

The rest of the pipeline — optimizer, schedulers, dataloaders, multi-GPU support — would remain valid.

Why It Matters

By studying LLaMA in the context of GPT-2, you see that modern LLMs aren't completely alien. They're evolutionary improvements on the same transformer backbone. Recognizing these small architectural changes (RMSNorm, RoPE, scaling) helps demystify why newer models outperform older ones, and it shows you exactly what you'd need to tweak in *llm.c* to explore beyond GPT-2.

Try It Yourself

1. Implement RMSNorm in C by adapting the LayerNorm code in *llm.c*.
2. Add a simplified version of RoPE to the attention kernel and run it on Tiny Shakespeare.
3. Swap the GPT-2 tokenizer for a SentencePiece model and train a small LLaMA-like model on your own dataset.
4. Compare training stability between LayerNorm and RMSNorm — does the loss curve look different?
5. Study the memory use of GPT-2 vs. a small LLaMA-style variant and see how scaling behaves.

The Takeaway

Exploring LLaMA through *llm.c* shows how flexible the codebase really is. With only a few targeted changes — normalization, positional encoding, tokenizer — you can shift from replicating GPT-2 to experimenting with the building blocks of modern LLMs. This makes *llm.c* not just a study tool for one model, but a foundation for understanding the entire lineage of transformers.

89. Porting Playbook: C → Go/Rust/Metal

The *llm.c* codebase is written in plain C for maximum readability and minimal dependencies. But in practice, many developers want to experiment with other languages or platforms — for example, writing a Go or Rust version for better tooling, or targeting Apple's Metal API for GPU acceleration on Macs. Porting is not just a copy-paste exercise; it requires careful thinking about how low-level memory, math operations, and parallelism map across ecosystems.

Why Port at All?

- Go: strong concurrency model (goroutines, channels), good for building training services or distributed experiments.
- Rust: memory safety and performance without garbage collection, ideal for writing reliable numerical kernels.
- Metal (Apple): GPU acceleration on macOS/iOS, a must if you want to train or run models efficiently on Apple Silicon.

Each ecosystem has strengths that make *llm.c*'s concepts more approachable or more production-ready.

Mapping the Core Components

Let's look at how the key pieces of *llm.c* translate:

Component	C (llm.c)	Go Equivalent	Rust Equivalent	Metal Equivalent
Memory allocation	`malloc`, `calloc`	`make`, slices, `unsafe.Pointer`	`Vec<T>`, `Box`, `unsafe` if needed	Buffers allocated on GPU
Math kernels	manual loops, OpenMP	loops or cgo bindings to BLAS	loops with iterators, Rayon for CPU	Metal compute shaders
Tokenizer	`fread` binary file	standard file I/O, `encoding/json`	`serde`, binary read	Preprocessing on CPU, feed to GPU
Training loop	for-loops, structs	goroutines for dataloader + trainer	async tasks, channels	CPU driver, GPU kernels
Parallelism	`#pragma omp`	goroutines + sync primitives	Rayon or explicit threads	Warp/thread groups in Metal

Example: LayerNorm in Rust

Here's a small Rust sketch of how a forward pass for LayerNorm might look:

```rust
fn layernorm_forward(out: &mut [f32], inp: &[f32], weight: &[f32], bias: &[f32], c: usize) {
    let mean: f32 = inp.iter().sum::<f32>() / c as f32;
    let var: f32 = inp.iter().map(|x| (x - mean).powi(2)).sum::<f32>() / c as f32;
    let rstd = 1.0 / (var + 1e-5).sqrt();

    for i in 0..c {
        let norm = (inp[i] - mean) * rstd;
```

```
        out[i] = norm * weight[i] + bias[i];
    }
}
```

This looks strikingly similar to the C code, but benefits from Rust's type safety and the absence of manual memory management bugs.

Example: Attention Kernel in Metal

Metal would handle attention differently — you'd write a compute shader in `.metal` language:

```
kernel void attention_forward(
    device float* out [[buffer(0)]],
    device float* qkv [[buffer(1)]],
    uint id [[thread_position_in_grid]]
) {
    // compute a dot product between query and key vectors
    // accumulate into out, using threadgroup memory for efficiency
}
```

This isn't a line-for-line port, but it shows how the concept — multiply queries with keys, apply softmax, weight values — remains the same while the implementation moves into GPU-land.

Challenges You'll Face

- Numerical libraries: C often leans on BLAS/LAPACK or just hand-written loops. In Go and Rust, you'll either bind to these libraries or reimplement them.
- Performance portability: getting code that runs fast both on CPU and GPU isn't trivial. What works in C with OpenMP won't directly translate.
- Tokenizer compatibility: making sure tokenization matches byte-for-byte is essential. One mismatch can ruin training reproducibility.

Why It Matters

Porting *llm.c* forces you to understand what each piece of the code is doing — you can't just rely on `torch.nn.LayerNorm` or `torch.nn.MultiheadAttention`. This makes it an excellent exercise for truly learning transformers, while also giving you practical implementations in different environments.

Try It Yourself

1. Reimplement one kernel (like LayerNorm or matmul) in Go or Rust. Test it against the C version.
2. Write a minimal Metal kernel that adds two vectors, then extend it to matrix multiplication.
3. Build a Rust tokenizer reader that loads `gpt2_tokenizer.bin` and decodes IDs back into text.
4. Compare training speed: C with OpenMP vs. Rust with Rayon vs. Go with goroutines.
5. Try porting just the forward pass first — inference is easier than training because you don't need backprop.

The Takeaway

The *llm.c* design is portable by nature — it doesn't hide behind opaque frameworks. Porting it to Go, Rust, or Metal is not just about performance or language preference. It's about proving to yourself that the transformer algorithm is universal, and you can implement it anywhere once you truly understand it.

90. Keeping the Repo Minimal and Clean

One of the defining features of *llm.c* is its minimalism. The repository avoids the sprawling complexity of large frameworks and instead sticks to a small, readable core. This design choice isn't an accident—it's a philosophy. By keeping the codebase small and clean, contributors can focus on understanding the fundamentals of transformers rather than navigating thousands of lines of boilerplate.

The Philosophy of Minimalism

- Readability over performance: While production frameworks like PyTorch or TensorFlow optimize aggressively, *llm.c* intentionally trades some performance for clarity. A loop in plain C is easier to study than a chain of optimized CUDA calls hidden behind macros.
- Portability: A smaller codebase can be ported more easily to new environments (Go, Rust, Metal) without pulling in dozens of dependencies.
- Learning-first design: Every line of code has a clear purpose. There are no abstractions "just in case."

This philosophy turns *llm.c* into both a working training framework and an educational resource.

How Minimalism Is Enforced

1. Flat structure: The repository avoids deep directory hierarchies. Most files live directly under `llmc/` or `dev/`.
2. No external libraries unless critical: You'll see OpenMP, cuBLAS, or cuDNN for performance, but not sprawling dependency chains.
3. One feature, one file: Tokenization, dataloading, schedulers, and samplers each get their own small C file. This prevents "god files" where too much is lumped together.
4. Consistent naming: Functions and structs use clear, descriptive names (e.g., `dataloader_next_batch`, `tokenizer_decode`) so readers don't get lost.

Practical Examples

If you open `train_gpt2.c`, you'll find that it builds the model, initializes dataloaders, and runs the training loop. It doesn't try to handle every possible model configuration, dataset format, or distributed scenario. Those belong in specialized files or external tools.

Likewise, `llmc/utils.h` only defines the absolute essentials: safe file I/O wrappers (`fopenCheck`, `freadCheck`) and memory allocators. It's not bloated with generic helpers unrelated to training GPT-2.

Why It Matters

A minimal repo lowers the barrier to entry. Beginners can trace execution from `main()` all the way to the optimizer update without detours. Researchers can fork the code and modify it without worrying about breaking dozens of interconnected modules. Even advanced developers benefit because the simplicity forces clarity in reasoning about algorithms.

Try It Yourself

1. Pick any file in the repo and count how many lines it has. Most are under a few hundred. Compare this with a similar file in PyTorch or TensorFlow.
2. Try adding a new feature—say, a different activation function. Notice how easy it is to slot it in because the structure is clean.
3. Explore what happens if you make the repo "heavier." Add too many helpers, abstractions, or configs. Does it make the code harder to read?
4. Practice explaining the training loop to a friend. If the repo is simple, you should be able to walk them through without glossing over details.

The Takeaway

Keeping *llm.c* minimal is not just about saving lines of code. It's about preserving clarity, ensuring reproducibility, and making the repository a place where anyone—from curious learners to experienced engineers—can open a file and *understand what's happening*. The simplicity is the point, and that's what makes *llm.c* a rare and valuable resource in a world of bloated ML frameworks.

Chapter 10. Reproduction, community and roadmap

91. Reproducing GPT-2 124M on Single Node

The first major milestone for anyone exploring *llm.c* is to reproduce the training of GPT-2 124M, the smallest version of the GPT-2 family. This model has around 124 million parameters, which is large enough to be interesting, but still small enough to train on a single modern GPU—or even slowly on CPU for demonstration purposes.

Why Start with 124M?

GPT-2 comes in multiple sizes: 124M, 355M, 774M, and 1.6B parameters. Training the largest requires clusters of GPUs and serious compute budgets. The 124M version, however, fits comfortably on consumer-grade GPUs like an NVIDIA 3090, and can even be run on a laptop CPU if you're patient. It's the "hello world" of transformer reproduction: small, approachable, and still real.

What the Training Setup Looks Like

Training GPT-2 124M involves a few key steps:

1. Model Configuration The config for 124M is baked into *llm.c* with 12 layers, 12 heads, hidden dimension of 768, and max sequence length of 1024.

```
GPT2Config config = {
    .max_seq_len = 1024,
    .vocab_size = 50257,
    .num_layers = 12,
    .num_heads = 12,
    .channels = 768
};
```

2. Dataset You can train on small datasets like Tiny Shakespeare or Tiny Stories for quick runs. For more realistic reproduction, you need something closer to OpenWebText. The data is tokenized with the GPT-2 tokenizer (`gpt2_tokenizer.bin`) and stored in `.bin` files.
3. Batch Size and Sequence Length A common setting is `B = 8, T = 1024`, meaning 8 sequences, each of length 1024 tokens. Adjust these based on available memory.
4. Optimizer AdamW with a learning rate around `3e-4` is the default. Warmup and cosine decay scheduling can be enabled to match published GPT-2 training curves.

What to Expect in Practice

On CPU, training a single step may take several seconds. On a single GPU with CUDA, each step may take under 100 milliseconds. With 124M parameters, training from scratch on a dataset the size of OpenWebText still takes days, but you can reproduce key dynamics (loss curve, sample generations) on smaller datasets in hours.

As an example, here's the type of log you might see:

```
step 0: train loss 6.9321 (took 421.5 ms)
step 100: train loss 4.2137 (took 95.2 ms)
step 200: train loss 3.8914 (took 94.8 ms)
val loss 3.7725
```

Even within a few hundred steps, the model begins to generate text resembling English rather than pure noise.

Why It Matters

Reproducing GPT-2 124M is a confidence check. If your setup is correct, your loss curves should match those in the original OpenAI paper or the PyTorch reference implementation. This validates that *llm.c* is a faithful reproduction, not just a toy. It also teaches you how much compute and data go into even the smallest GPT-2 model, building intuition about scaling laws.

Try It Yourself

1. Train GPT-2 124M for 1000 steps on Tiny Shakespeare. Watch how the generated text improves.
2. Change the batch size `B` from 8 to 4. What happens to the speed and the stability of training?
3. Run training on CPU vs. GPU. Compare how long each step takes.
4. Track both training and validation loss. Notice how they diverge when the model begins to overfit.

The Takeaway

The GPT-2 124M run is more than just a demo—it's your gateway into real LLM training. You see how data, model size, optimizer, and hardware come together. Once you've mastered this reproduction, you're ready to push toward larger models and more complex setups. It's the foundation on which everything else in *llm.c* builds.

92. Reproducing GPT-2 355M (Constraints and Tricks)

Once GPT-2 124M has been successfully reproduced, the next logical step is scaling up to GPT-2 355M. This version is roughly three times larger, with about 355 million parameters. It introduces new challenges that don't appear at the smaller scale: memory pressure, training stability, and compute cost.

Model Configuration

The 355M model still uses 1024 tokens as the maximum sequence length and the same GPT-2 tokenizer. The difference is in the depth and width of the network:

- Layers: 24 transformer blocks instead of 12
- Hidden dimension (channels): 1024 instead of 768
- Heads: 16 instead of 12

The total parameter count rises from ~124M to ~355M. That means not just three times more math per step, but also more memory needed for parameters, gradients, and optimizer state.

The Compute Challenge

With 124M, a single GPU with 8–12 GB of VRAM is enough. For 355M, you need at least 16 GB to run comfortably with sequence length 1024 and batch size of 8. On smaller GPUs, you'll quickly hit "CUDA out of memory" errors.

One trick is to reduce batch size (B) or sequence length (T). For example, instead of training with `(B=8, T=1024)`, you might use `(B=4, T=512)`. This halves the memory footprint but still lets you test scaling dynamics.

Another approach is to use gradient accumulation: simulate a larger batch size by running multiple small steps and accumulating gradients before updating.

Training Stability

Larger models are more sensitive to hyperparameters. The AdamW optimizer still works, but the learning rate schedule becomes more important. Many practitioners use:

- Learning rate: ~3e-4 peak
- Warmup steps: a few thousand
- Cosine decay: to taper the learning rate gradually

If you skip warmup, the larger model may diverge early (loss exploding instead of decreasing).

Tricks for Feasibility

1. Mixed Precision Training (FP16 or BF16): Cuts memory use nearly in half. Supported in CUDA paths of *llm.c*.
2. Activation Checkpointing: Save memory by recomputing activations during backpropagation. Slower, but lets you fit bigger models.
3. Smaller Dataset Runs: Train on Tiny Shakespeare or Tiny Stories to sanity-check the setup, then scale to OpenWebText-like data.

Example Logs

Running GPT-2 355M for a short demo might look like:

```
step 0: train loss 7.1032 (took 321.8 ms)
step 50: train loss 5.4231 (took 310.4 ms)
step 100: train loss 4.8217 (took 311.0 ms)
val loss 4.7322
```

The loss drops more slowly than with 124M because the model has more capacity to learn, but also needs more data to generalize.

Why It Matters

355M is the first step into "medium-sized" LLMs. You start to feel the bottlenecks that dominate larger models: VRAM limits, training speed, and hyperparameter tuning. Solving these prepares you for the 774M and 1.6B experiments, where such problems become even more pronounced.

Try It Yourself

1. Train GPT-2 355M with batch size 4 and sequence length 512. Record how long each step takes.
2. Experiment with warmup steps: run once with warmup=0, and once with warmup=2000. Compare stability.
3. Enable mixed precision if you have a CUDA-capable GPU. Measure memory usage before and after.
4. Try training on Tiny Shakespeare vs. Tiny Stories. Does the model overfit faster on the smaller dataset?

The Takeaway

Reproducing GPT-2 355M is all about learning how to stretch limited resources. You'll discover memory-saving tricks, the importance of learning rate schedules, and the role of data scale. It's a practical exercise in resource management—just like the real challenges faced when training today's billion-parameter models.

93. Reproducing GPT-2 774M (Scaling Up)

The 774M parameter version of GPT-2 is often called "GPT-2 Medium." This is the point where training transitions from a personal experiment to a small-scale research project. It's about six times larger than the 124M baseline, and roughly twice the size of 355M. Running it requires careful planning of hardware, memory, and software tricks.

Model Configuration

For 774M, the architecture expands again:

- Layers (transformer blocks): 36
- Hidden size (channels): 1280
- Attention heads: 20
- Maximum sequence length: 1024 (unchanged)

This jump in size increases both the parameter storage and the number of activations that must be kept during training. Optimizer states (AdamW's m and v vectors) alone consume several gigabytes.

Hardware Requirements

Running GPT-2 774M from scratch generally requires GPUs with 24 GB VRAM or more (e.g., NVIDIA RTX 3090/4090, A100, or H100). With smaller cards, you'll almost certainly hit memory errors unless you aggressively reduce batch size and use techniques like activation checkpointing.

On CPUs, training is technically possible but far too slow to be practical—steps that take milliseconds on GPUs might take many seconds or even minutes.

Practical Constraints

1. Batch Size: In practice, you may need to lower `B` to 2 or even 1 with sequence length 1024.
2. Gradient Accumulation: A must-have to simulate larger batch sizes and stabilize training.
3. Mixed Precision: FP16 or BF16 reduces memory by about half, without hurting convergence much.
4. Checkpointing: Recomputes intermediate results instead of storing them, trading time for memory.

Training Dynamics

The loss curve for GPT-2 774M drops more steadily and requires much more data to reach its potential. If you only train on Tiny Shakespeare or Tiny Stories, it will quickly overfit: the model is too large for such a small dataset. For meaningful reproduction, you need a dataset similar in scale to OpenWebText.

Training logs for a sanity check run might look like:

```
step 0: train loss 8.0123 (took 512.4 ms)
step 50: train loss 5.7892 (took 490.7 ms)
step 100: train loss 5.1428 (took 495.1 ms)
val loss 5.0039
```

Notice that losses start higher (due to the larger random initialization space) but decrease predictably once training gets underway.

Why It Matters

The 774M model is the sweet spot where scaling laws become obvious. Compared to 124M and 355M, it generalizes better, generates more fluent text, and demonstrates the benefits of parameter growth. But it also shows why infrastructure matters: without careful management, it's nearly impossible to train this model on consumer-grade hardware.

This is also where distributed training (covered in Chapter 8) becomes relevant, because one GPU is often not enough for efficient scaling.

Try It Yourself

1. Train GPT-2 774M with `(B=1, T=1024)` and gradient accumulation over 8 steps. Watch how it simulates a batch size of 8.
2. Compare training with FP32 vs. mixed precision. Measure both memory use and speed.
3. Run a short fine-tuning experiment on Tiny Stories. Observe how quickly the model memorizes the dataset.
4. Plot the training and validation loss curves. Does the larger model overfit faster or slower than 355M?

The Takeaway

Reproducing GPT-2 774M is about scaling into the territory of real research workloads. You face serious memory constraints, dataset requirements, and compute costs. But if you succeed, you'll see firsthand why the machine learning community kept pushing toward billion-parameter models: larger networks unlock noticeably stronger capabilities, even with the same architecture.

94. Reproducing GPT-2 1.6B on 8×H100 (24h Run)

The largest GPT-2 model, with 1.6 billion parameters, represents the upper bound of the original GPT-2 family. Training this model from scratch is not something you can casually attempt on a single workstation. It demands cluster-scale resources, distributed training software, and careful tuning to keep everything stable. In this section, we'll walk through what makes the 1.6B model special, what infrastructure is required, and how a full reproduction might look.

Model Configuration

The jump from 774M to 1.6B doubles the parameter count and makes the network both deeper and wider:

- Layers (transformer blocks): 48
- Hidden size (channels): 1600
- Attention heads: 25
- Sequence length: still 1024

With these dimensions, every forward and backward pass requires massive amounts of memory and compute. Just storing the parameters in FP32 takes around 6.4 GB. Once you add gradients, optimizer states (AdamW's m and v), and activations, the memory footprint easily exceeds 100 GB.

Hardware Setup

To reproduce this model realistically, you need access to high-end accelerators such as NVIDIA A100s or H100s. A common baseline is 8 GPUs with 80 GB each. With this setup, it is possible to train GPT-2 1.6B in under 24 hours, assuming efficient utilization.

Without multi-GPU, training is impractical. Even if you could somehow fit the model on one GPU, the runtime would be weeks or months.

Distributed Training

The main strategy is data parallelism: each GPU processes a different mini-batch of data, and gradients are averaged across all devices with NCCL all-reduce. The code paths in *llm.c* support this via MPI integration, so you can scale from single-GPU to multi-node setups.

The training loop looks nearly identical to smaller models, but behind the scenes, every parameter update is coordinated across devices.

Training Dynamics

The loss curve for 1.6B smooths out compared to smaller models. With enough data, the model continues to improve where 774M starts to plateau. This was one of the key insights from the original GPT-2 paper: scaling laws hold, and performance improves predictably with size, data, and compute.

Logs from a distributed run might look like this:

```
[rank 0] step 0: train loss 8.5029 (took 312.6 ms)
[rank 0] step 50: train loss 6.3121 (took 308.2 ms)
[rank 0] step 100: train loss 5.7210 (took 309.4 ms)
[rank 0] val loss 5.5347
```

Notice the speed: each step still takes only a few hundred milliseconds despite the massive size, thanks to parallelism across multiple H100s.

Why It Matters

Reproducing GPT-2 1.6B is less about training a useful model today and more about understanding the scaling challenges of large language models. This exercise demonstrates how compute, memory, and distributed infrastructure become the limiting factors as models grow. It also shows why modern research labs design entire pipelines around multi-GPU and multi-node scaling.

Try It Yourself

1. Simulate a multi-GPU run with fewer resources by reducing the model size but using the same parallel training setup. For example, train GPT-2 124M across 2 GPUs to practice the workflow.
2. Experiment with gradient accumulation to mimic large global batch sizes even on smaller clusters.
3. Try enabling and disabling mixed precision. Watch how memory use drops dramatically with FP16/BF16.
4. Plot validation loss curves for 124M, 355M, 774M, and 1.6B side by side. Notice how the larger models sustain improvements longer.

The Takeaway

The GPT-2 1.6B reproduction is the capstone project of *llm.c*. It forces you to combine everything you've learned: data pipelines, optimizers, schedulers, distributed training, and system-level debugging. While few people will actually train 1.6B themselves, understanding what it takes provides a window into the engineering behind state-of-the-art LLMs and prepares you to engage with even larger modern models.

95. CPU-only Fine-Tune Demo (Tiny Shakespeare)

Not everyone has access to powerful GPUs or large compute clusters. One of the strengths of *llm.c* is that it provides a clean, minimal CPU-only path that lets you run real experiments—even if they are small and slow. A practical way to explore this is by fine-tuning GPT-2 on a small dataset like Tiny Shakespeare, which has only about 1 MB of text.

Why Tiny Shakespeare?

Tiny Shakespeare is a classic toy dataset in machine learning. It's small enough to fit in memory, yet it contains a rich variety of words, characters, and structures. Fine-tuning GPT-2 on this dataset allows the model to mimic Shakespearean style in just a few thousand steps. It's not about building a state-of-the-art model—it's about seeing the training process work end-to-end on modest hardware.

Setup

The fine-tuning process uses the same `train_gpt2.c` CPU path, but with fewer steps, smaller batch sizes, and lower sequence lengths to keep things fast. A typical setup looks like this:

```
int B = 4;    // batch size
int T = 64;   // sequence length
int steps = 1000;  // training iterations
```

The dataset is tokenized once using `gpt2_tokenizer.bin` and stored in binary `.bin` files:

```
dev/data/tinyshakespeare/tiny_shakespeare_train.bin
dev/data/tinyshakespeare/tiny_shakespeare_val.bin
```

These files are only a few megabytes, making them perfect for quick experiments.

Training Dynamics

When you fine-tune on Tiny Shakespeare, the training logs may look like this:

```
step 0: train loss 6.9312 (took 1220.4 ms)
step 50: train loss 4.3217 (took 1175.1 ms)
step 100: train loss 3.7120 (took 1169.4 ms)
val loss 3.5894
```

Within a few hundred steps, the loss drops rapidly. By the time you've run 1000 steps, the model starts producing text that looks recognizably Shakespearean—complete with archaic words, unusual punctuation, and rhythmic patterns.

Example Output

Here's a short sample from a fine-tuned run:

```
generating:

ROMEO: But hark, what light through yonder window breaks?
JULIET: Ay me! the time is near, and I must away.
ROMEO: Fear not, sweet love, for night shall bring us peace.
```

It isn't perfect, but it captures the "feel" of Shakespeare, which is remarkable given the tiny dataset and limited compute.

Why It Matters

The CPU-only Tiny Shakespeare demo proves that LLMs are not just for massive data centers. With a minimal setup, you can watch the model learn, generate text, and overfit to a dataset. This hands-on practice builds intuition about what training does, how loss curves behave, and why scaling up matters.

Try It Yourself

1. Change the sequence length from 64 to 128. How does training speed and loss change?
2. Reduce the number of training steps to 200. Do you still see Shakespeare-like text in generation?
3. Increase the batch size to 8. Does the loss curve become smoother or noisier?
4. Fine-tune again but initialize from scratch (random weights). Compare the results to fine-tuning from pretrained GPT-2 124M.

The Takeaway

Fine-tuning GPT-2 on Tiny Shakespeare with CPU-only training is a simple yet powerful demonstration. You don't need GPUs to understand the mechanics of transformers. Even a modest laptop can teach you how training works, why overfitting happens, and how LLMs adapt to new domains. It's a reminder that the best way to learn machine learning is by rolling up your sleeves and running experiments—even small ones.

96. Cost and Time Estimation for Runs

One of the most eye-opening parts of working with large language models is realizing how expensive and time-consuming training can be. While *llm.c* makes it possible to run models of all sizes with simple, minimal C code, the hardware requirements grow quickly as you scale from GPT-2 124M to GPT-2 1.6B. Understanding cost and time estimation helps set realistic expectations, whether you're running on a laptop CPU, a single GPU, or a rented cluster of accelerators.

Key Factors Affecting Training Time

Several components determine how long training takes and how much it costs:

- Model size (parameters): Larger models mean more multiplications per forward/backward pass, and more memory for parameters, gradients, and optimizer states.
- Batch size and sequence length: Increasing either multiplies the amount of work per step.
- Dataset size: Bigger datasets require more steps to complete one epoch.
- Hardware speed: CPUs are far slower than GPUs; high-end GPUs like H100s can be 100× faster than CPUs for this workload.
- Parallelism: Multi-GPU or multi-node setups let you divide the work, reducing time per step.

Rough Time Estimates

Here's a simplified view of how long it might take to train GPT-2 models depending on hardware and setup. These are very approximate, assuming full training on a dataset like OpenWebText:

Model	Parameters	Hardware	Time per Step	Total Training Time
GPT-2 124M	~124M	Laptop CPU	1–2 s	Months

Model	Parameters	Hardware	Time per Step	Total Training Time
GPT-2 124M	~124M	Single RTX 3090	~100 ms	~2–4 weeks
GPT-2 355M	~355M	Single RTX 3090	~300 ms	~6–8 weeks
GPT-2 774M	~774M	2× A100 40GB	~200 ms	~4–6 weeks
GPT-2 1.6B	~1.6B	8× H100 80GB	~300 ms	~24 hours

These numbers show why scaling matters. The larger models don't just need more compute per step—they also need more data to reach their potential. That means the total cost balloons unless you have a cluster of top-tier GPUs.

Cost in Cloud Environments

If you run these experiments on cloud providers like AWS, GCP, or Azure, costs can add up quickly. For example:

- An NVIDIA A100 40GB instance costs around $2–3 per hour (spot pricing can be cheaper).
- Training GPT-2 124M for a week might cost $500–1,000.
- Training GPT-2 1.6B for 24 hours on 8× H100s might cost $5,000–10,000, depending on the provider.

This is why many researchers test code paths on small datasets and small models first, then only scale up when absolutely necessary.

Why It Matters

Estimating cost and time prevents frustration and wasted money. It teaches you to prototype at small scale (CPU or 124M runs), validate your setup, and only then scale up to medium (355M, 774M) or large (1.6B) models. It also gives you a realistic appreciation for the engineering and budget that went into OpenAI's original GPT-2 training runs.

Try It Yourself

1. Time how long a single training step takes on your hardware for GPT-2 124M. Multiply by 1000 to estimate training time for 1000 steps.
2. Reduce the batch size by half. Does time per step decrease linearly, or not?

3. Run a short fine-tune (e.g., 200 steps) and measure the electricity cost if you're on a home machine.
4. Use a cloud GPU for one hour. Compare the cost and speed to your local CPU.

The Takeaway

Training large language models is a balancing act between ambition, hardware, and budget. *llm.c* gives you the tools to explore everything from toy demos to billion-parameter reproductions, but you'll quickly see why the field has shifted toward big labs and shared infrastructure. With careful planning, though, you can still learn a tremendous amount by running smaller experiments and scaling them up thoughtfully.

97. Hyperparameter Sweeps (`sweep.sh`)

Getting a model like GPT-2 to train well isn't just about writing the code or having enough compute—it's also about finding the right hyperparameters. These include the learning rate, batch size, weight decay, dropout rate, and scheduler configuration. A setting that works well for one dataset or model size might completely fail for another. That's where hyperparameter sweeps come in: systematically trying different configurations to see which ones give the best results.

The Role of `sweep.sh`

In *llm.c*, there's a simple shell script called `sweep.sh` designed to automate hyperparameter testing. It's not a complex experiment management system like Ray Tune or Optuna; instead, it's lightweight, transparent, and easy to adapt. The script usually looks like a loop over different learning rates or batch sizes, running the training executable with each setting, and logging the output.

A very simplified version might look like this:

```
#!/bin/bash
for lr in 1e-3 5e-4 1e-4
do
    echo "Running with learning rate $lr"
    ./train_gpt2 -lr $lr -epochs 5 > logs/lr_$lr.txt
done
```

This way, you can launch multiple experiments with a single command and later compare validation losses to decide which hyperparameters are best.

Why Sweeps Are Important

Training a transformer is highly sensitive to hyperparameters. For example:

- If the learning rate is too high, loss might explode.
- If it's too low, training will be painfully slow.
- Too much weight decay can hurt performance, while too little can cause overfitting.
- The number of warmup steps can make the difference between stable convergence and failure in the first few hundred iterations.

Instead of guessing, sweeps let you see patterns. For instance, you might discover that `3e-4` is optimal for GPT-2 124M, but GPT-2 355M prefers `2e-4`.

Example of a Sweep in Practice

Suppose you want to test three learning rates and two batch sizes. You can write a nested loop:

```
for lr in 3e-4 2e-4 1e-4
do
    for B in 4 8
    do
        echo "Running with lr=$lr and batch_size=$B"
        ./train_gpt2 -lr $lr -B $B -steps 500 > logs/lr_${lr}_B${B}.txt
    done
done
```

Afterward, you could open the logs and compare validation loss at the end of each run. This gives you data-driven evidence about what works best.

Why It Matters

Hyperparameter sweeps are a cornerstone of practical machine learning. Even though *llm.c* is a minimalist project, the ability to quickly test and compare runs is essential. It transforms training from guesswork into an empirical process. You don't just hope your model will converge—you verify it across multiple settings and pick the winner.

Try It Yourself

1. Run a sweep over learning rates `[1e-3, 3e-4, 1e-4]` for GPT-2 124M on Tiny Shakespeare. Which one converges fastest?
2. Try sweeping over sequence length (`T=32, 64, 128`). How does it affect speed and loss?
3. Compare runs with and without weight decay. Which generalizes better to the validation set?
4. Extend the sweep to test different schedulers (cosine vs. step decay).

The Takeaway

Hyperparameter sweeps are the "experimentation muscle" of training LLMs. They teach you that no single setting works everywhere and that systematic testing is far more effective than intuition alone. With a simple script like `sweep.sh`, you can explore dozens of setups in a reproducible way and build confidence that your model is training as well as it can.

98. Validating Evaluation and Loss Curves

Training logs—loss values printed after each step—are just numbers. To really understand whether your model is learning, you need to validate those numbers, plot them, and compare them across runs. This process of analyzing evaluation and loss curves is one of the most important skills in machine learning. It's how you know whether your model is converging, overfitting, or failing entirely.

Training Loss vs. Validation Loss

There are two kinds of loss curves to pay attention to:

- Training loss: computed on the batches the model actually sees during training.
- Validation loss: computed on a held-out dataset (like `*_val.bin` in *llm.c*).

Training loss almost always goes down steadily. Validation loss is the real test: if it decreases alongside training loss, the model is learning useful patterns. If it stalls or increases while training loss keeps dropping, the model is overfitting.

Plotting Loss Curves

Even though *llm.c* is a pure C project, you can redirect its training logs to a file and then use tools like Python + matplotlib to visualize. For example:

```
./train_gpt2 > logs/run1.txt
```

Then parse `logs/run1.txt` in Python:

```
import re, matplotlib.pyplot as plt

steps, train_loss = [], []
for line in open("logs/run1.txt"):
    match = re.match(r"step (\d+): train loss ([0-9.]+)", line)
    if match:
        steps.append(int(match.group(1)))
        train_loss.append(float(match.group(2)))

plt.plot(steps, train_loss, label="train loss")
plt.legend()
plt.show()
```

Adding validation loss to the same plot makes it even more useful: you'll see both curves and their relationship over time.

What a Healthy Curve Looks Like

- Early phase: Both training and validation loss drop quickly.
- Middle phase: Training loss continues downward, validation loss drops more slowly.
- Late phase: Training loss may keep decreasing, but validation loss stabilizes or begins to rise. That's the point of overfitting.

For example, a good curve might look like this:

```
step 0: train loss 6.92, val loss 6.85
step 100: train loss 4.31, val loss 4.52
step 200: train loss 3.78, val loss 4.01
step 500: train loss 2.91, val loss 3.95
```

Training loss keeps going down, but validation loss plateaus around step 500. That's a signal to stop training or adjust hyperparameters.

Why It Matters

Loss curves are your window into model behavior. They reveal whether your model is underfitting (loss too high), overfitting (gap between train and val too large), or training stably (both losses decreasing together). Without them, you're flying blind—just staring at numbers without context.

Try It Yourself

1. Train GPT-2 124M for 500 steps on Tiny Shakespeare and plot both training and validation loss.
2. Reduce the dataset size by half. Watch how validation loss worsens earlier due to overfitting.
3. Run two experiments with different learning rates. Plot both curves and compare stability.
4. Extend plotting to multiple runs (e.g., GPT-2 124M vs. 355M) to see scaling effects.

The Takeaway

Validating evaluation and loss curves turns raw logs into insight. It helps you decide when to stop training, how to tune hyperparameters, and whether scaling up is worth it. In *llm.c*, even though the project is minimal, capturing and plotting these curves is the single most effective way to understand what's happening inside your model.

99. Future Work: Kernel Library, Less cuDNN Dependence

The CUDA path in *llm.c* already uses cuBLAS and cuDNN, NVIDIA's high-performance math libraries, to handle the heavy lifting of matrix multiplications and attention operations. These libraries are battle-tested and extremely fast, but they also act as a "black box": you call into them, they do the work, and you get results without seeing what's inside. While this is convenient, it limits flexibility and makes it harder to experiment with novel optimizations.

That's why one of the most exciting areas of future work is building a lightweight custom kernel library for *llm.c*. This would mean replacing parts of cuDNN with hand-written CUDA kernels for operations like attention, normalization, and activation functions.

Why Reduce cuDNN Dependence?

1. Transparency: With custom kernels, you see exactly how operations are implemented, which is great for learning and debugging.

2. Flexibility: You can experiment with new ideas (e.g., alternative attention mechanisms, sparsity tricks) without waiting for cuDNN support.
3. Portability: cuDNN is NVIDIA-specific. A custom kernel library could make it easier to port *llm.c* to other backends, like AMD GPUs (HIP) or even Metal for Apple silicon.
4. Performance Tuning: For small to medium models, hand-tuned kernels can sometimes outperform generic library calls because they're tailored to the workload.

What a Kernel Library Might Include

A first version of a kernel library for *llm.c* might implement:

- Matrix multiplication (the workhorse of transformers) using tiling and shared memory.
- Softmax kernels with numerical stability built in.
- LayerNorm kernels for both forward and backward passes.
- Attention kernels that integrate matmul + masking + softmax in one fused operation.
- Activation functions like GELU or ReLU in fused forms.

For example, a very simplified CUDA kernel for vector addition might look like this:

```
__global__ void vec_add(float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        c[i] = a[i] + b[i];
    }
}
```

While trivial, this illustrates the idea: instead of calling a library, you write the math yourself and control how threads are launched and memory is accessed.

The Path Forward

Developing a kernel library is a long-term effort. It requires profiling, benchmarking, and iterative tuning. At first, custom kernels may be slower than cuDNN, but over time, they can evolve into a compact, educational library of building blocks for transformer training.

Why It Matters

Reducing dependence on cuDNN isn't just about performance—it's about control and portability. By having your own kernels, you gain the freedom to run *llm.c* on more platforms, test new research ideas, and understand exactly what's happening inside the GPU. For a minimal, educational project like this one, that's a natural next step.

Try It Yourself

1. Write a simple CUDA kernel for vector addition, like the one above, and call it from a C program. Compare its performance to `cublasSaxpy`.
2. Replace one small piece of *llm.c* (e.g., softmax) with a custom kernel. Check if the outputs match cuDNN.
3. Benchmark a custom kernel against cuDNN on a small input size. Does cuDNN still dominate?
4. Try porting your kernel to HIP (for AMD) or Metal (for Apple GPUs).

The Takeaway

Building a kernel library is about moving from consumer of black-box libraries to creator of transparent, flexible tools. It's a lot of work, but it transforms *llm.c* into not just a project for using LLMs, but also a platform for learning the deep internals of GPU programming. By reducing cuDNN dependence, you open the door to true end-to-end control of model training.

100. Community, GitHub Discussions, and Suggested Learning Path

Large language models are complicated, but one of the goals of *llm.c* is to make them accessible. The code is clean, minimal, and approachable—but learning doesn't stop at reading code. The broader community around *llm.c* plays a huge role in helping people understand, experiment, and grow. This section highlights where to connect with others, how to contribute, and how to build your own learning journey.

GitHub as the Hub

The central place for *llm.c* discussions is the GitHub repository. There, you'll find:

- Issues: where users ask questions, report bugs, or propose improvements.
- Discussions: an open forum for sharing results, asking "how do I…?" questions, and comparing training logs.
- Pull Requests: contributions ranging from bug fixes to new features, often with valuable code reviews.

Even just browsing issues and discussions can be an educational experience. Many questions you might have—about CUDA errors, dataset preparation, or optimizer quirks—have already been asked and answered.

The Value of Community

Working alone on language models can feel overwhelming. By engaging with the community, you:

- See how others run experiments with different datasets and hardware.
- Learn troubleshooting strategies for common problems (e.g., out-of-memory errors).
- Get inspiration for extensions—like custom kernels, new optimizers, or non-GPT architectures.
- Find collaborators for experiments that go beyond what one person can do.

Suggested Learning Path

Because *llm.c* is minimal, it works well as a self-study tool. Here's a suggested path to build up your knowledge:

1. Start Small
    - Train GPT-2 124M on Tiny Shakespeare using CPU-only mode.
    - Inspect training logs and watch how loss decreases.
    - Generate text and see how quickly the model memorizes.
2. Step Into CUDA
    - Switch to `train_gpt2.cu` and train with GPU acceleration.
    - Try mixed precision (`FP16`/`BF16`) and observe memory savings.
3. Scale Up
    - Attempt GPT-2 355M or 774M on your hardware (or cloud GPUs).
    - Learn how to use gradient accumulation and checkpointing.
4. Experiment
    - Modify the training loop: try new schedulers, tweak optimizer hyperparameters.
    - Add your own datasets (e.g., your personal text corpus).
5. Explore Internals
    - Step through forward and backward passes in `train_gpt2.c`.
    - Write small experiments to isolate key concepts (e.g., LayerNorm).
6. Join the Discussion
    - Share your results on GitHub Discussions.
    - Contribute improvements, even small ones—like documentation fixes.

Why It Matters

The journey of learning LLM internals isn't just about reading code—it's about active practice, asking questions, and comparing experiences with others. Community provides the feedback loop that accelerates learning and keeps motivation alive.

Try It Yourself

1. Clone the *llm.c* repository and explore open issues. Can you answer one for someone else?
2. Run a training experiment and share your loss curve in GitHub Discussions.
3. Contribute a small improvement (like a new dataset script) as a pull request.
4. Create your own "learning log" to track experiments, much like a public notebook.

The Takeaway

llm.c isn't just a codebase—it's an invitation to join a learning community. By engaging with GitHub, trying experiments, and sharing your results, you move from passive reader to active participant. That's where the deepest understanding comes from: learning together, not alone.