LLaMA 3.x Inference Implementation in Pure C

Vishal Vardhan Adepu
Texas A&M University

December 2024

Abstract

This report presents the implementation of inference capabilities for Meta's LLaMA 3.1 and 3.2 models in pure C. The project focuses on creating an efficient inference engine for the 8B parameter model of LLaMA 3.1 and the 1B and 3B parameter models of LLaMA 3.2. Building upon Andrej Karpathy's llama2.c implementation, we introduce several optimizations and improvements, including a trie-based tokenization system. The report details the implementation challenges, architectural decisions, performance metrics, and future directions for enhancing the system.

1 Introduction

Large Language Models (LLMs) have revolutionized natural language processing, with Meta's LLaMA series being at the forefront of open-source models. While these models are typically implemented in high-level languages like Python, implementing inference in C can provide significant performance benefits and better resource control.

This project implements inference capabilities for LLaMA 3.1 (8B) and LLaMA 3.2 (1B, 3B) models in pure C, focusing on efficiency and scalability. The full source code for this project is available on GitHub at https://github.com/vva2/llama3x.c. A demonstration video showcasing the implementation is available at https://youtu.be/5ydhuw3koTs.

2 Background

2.1 LLaMA Architecture

The LLaMA models utilize a decoder-only transformer architecture with several key improvements over traditional transformers:

- Pre-normalization using RMSNorm
- SwiGLU activation function
- Rotary positional embeddings (RoPE)
- Grouped-Query Attention (GQA)

2.2 Key Papers and Resources

The implementation drew insights from several key papers:

- "Attention Is All You Need" - Foundation of transformer architecture [1]
- "The LLaMA 3 Herd of Models" - Architecture details of LLaMA 3.x [2]
- "GQA: Training Generalized Multi-Query Transformer Models" [3]
- "Round and Round We Go: What Makes Rotary Positional Encodings Useful?" [4]

3 Implementation

3.1 Core Components

3.1.1 Trie-based Tokenization

A key innovation in our implementation is the use of a space-efficient trie data structure for tokenization. Unlike traditional trie implementations that use fixed-size arrays for children, our implementation uses a linked list approach for memory efficiency. The core structure is defined as:

```c
typedef struct TrieNode {
    char c;                    // Character stored in this node
    struct TrieNode *children; // Pointer to first child
    struct TrieNode *next;     // Pointer to next sibling
    int id;                    // Token ID (-1 if not a complete token)
} TrieNode;
```

Listing 1: Trie Node Structure

Key features of the implementation include:

- **Memory Efficiency**: Using linked lists instead of arrays saves memory, especially important for large vocabularies with sparse character distributions
- **Dynamic Node Management**: Each node dynamically manages its children through a linked list structure
- **Efficient Search**: Implementation of `find_child` function for O(k) lookup where k is the average number of children per node
- **Clean Memory Management**: Comprehensive deallocation through recursive `free_trie` function

The trie supports three main operations:

```
// Create new trie with root node
TrieNode* create_trie();

// Insert a token with its ID
void insert(TrieNode* node, char* key, int id);

// Search for a token, returns ID or -1
int trie_search(TrieNode *trie, char *str);
```

Listing 2: Core Trie Operations

This trie implementation is particularly well-suited for the LLaMA tokenizer vocabulary because:

- It efficiently handles the variable-length subword tokens used in LLaMA models
- The linked list structure adapts well to the non-uniform distribution of characters in the vocabulary
- Memory usage scales with actual vocabulary content rather than potential character space
- Quick prefix matching enables efficient tokenization of input text

Implementation details like `find_child` utilize linear search through the linked list of children:

```
TrieNode* find_child(TrieNode* node, char c) {
    if(node != NULL && node->children != NULL) {
        TrieNode *ptr = node->children;
```

```c
        while(ptr != NULL && ptr->c != c) {
            ptr = ptr->next;
        }
        return ptr;
    }
    return NULL;
}
```

Listing 3: Child Node Search Implementation

This implementation choice trades some lookup speed for memory efficiency, which is crucial when dealing with large model vocabularies.

3.1.2 Transformer Architecture

The core transformer implementation includes:

- Memory-mapped weight loading for efficient model initialization
- RMSNorm implementation for layer normalization
- Multi-head attention with RoPE
- SwiGLU activation in feed-forward networks

3.2 Optimization Techniques

Several optimization techniques were employed:

- Use of -Ofast compiler optimization
- Memory mapping for efficient weight loading
- OpenMP parallelization for matrix operations
- Cache-friendly data structures and operations

4 Performance Analysis

4.1 Inference Speed

Performance measurements were conducted using identical prompts across all models with temperature set to 0.0. The results demonstrate the expected trade-off between model size and inference speed:

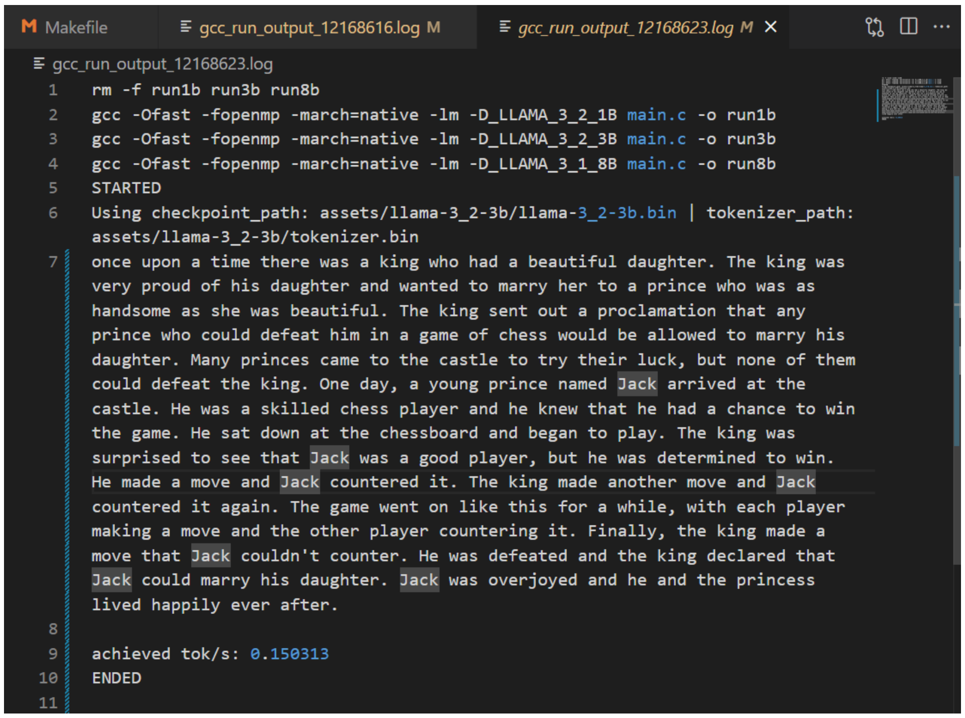
Model	Parameters	Speed (tokens/sec)
LLaMA 3.2 1B	1 billion	2.38
LLaMA 3.2 3B	3 billion	0.15
LLaMA 3.1 8B	8 billion	0.13

Table 1: Inference Speed Comparison

The measurements show a clear correlation between model size and inference speed, with the 1B model performing significantly faster than its larger counterparts. This performance difference is particularly notable when scaling from 1B to 3B parameters, where we observe an approximately 15x slowdown.

4.2 Execution Examples

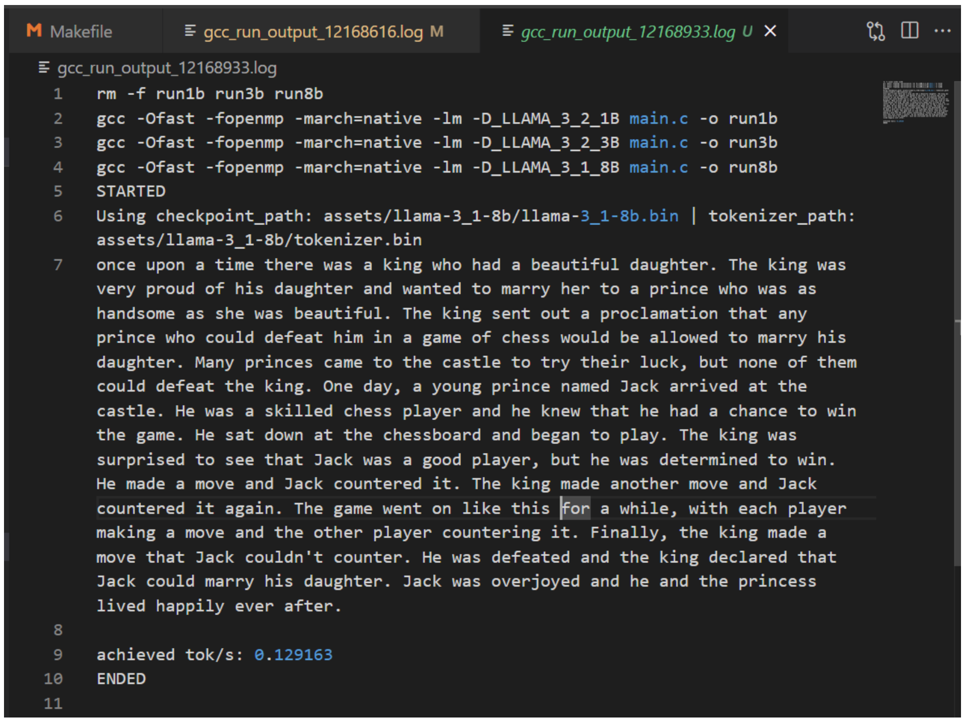
Below are examples of the model execution with compiler optimizations enabled (-Ofast -fopenmp -march=native):


Makefile
gcc_run_output_12168616.log M
gcc_run_output_12168623.log M
gcc_run_output_12168623.log

```
rm -f run1b run3b run8b
gcc -Ofast -fopenmp -march=native -lm -D_LLAMA_3_2_1B main.c -o run1b
gcc -Ofast -fopenmp -march=native -lm -D_LLAMA_3_2_3B main.c -o run3b
gcc -Ofast -fopenmp -march=native -lm -D_LLAMA_3_1_8B main.c -o run8b
STARTED
Using checkpoint_path: assets/llama-3_2-3b/llama-3_2-3b.bin | tokenizer_path:
assets/llama-3_2-3b/tokenizer.bin
once upon a time there was a king who had a beautiful daughter. The king was
very proud of his daughter and wanted to marry her to a prince who was as
handsome as she was beautiful. The king sent out a proclamation that any
prince who could defeat him in a game of chess would be allowed to marry his
daughter. Many princes came to the castle to try their luck, but none of them
could defeat the king. One day, a young prince named Jack arrived at the
castle. He was a skilled chess player and he knew that he had a chance to win
the game. He sat down at the chessboard and began to play. The king was
surprised to see that Jack was a good player, but he was determined to win.
He made a move and Jack countered it. The king made another move and Jack
countered it again. The game went on like this for a while, with each player
making a move and the other player countering it. Finally, the king made a
move that Jack couldn't counter. He was defeated and the king declared that
Jack could marry his daughter. Jack was overjoyed and he and the princess
lived happily ever after.

achieved tok/s: 0.150313
ENDED
```

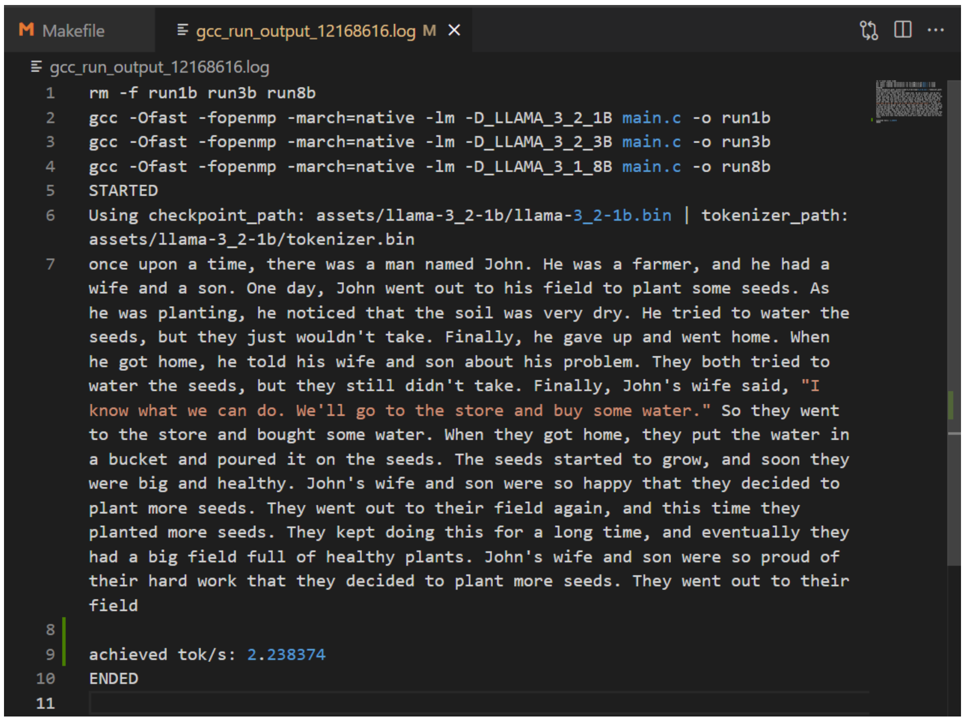
Figure 1: LLaMA 3.2 3B Model Execution (0.15 tokens/sec)


Makefile
gcc_run_output_12168616.log M
gcc_run_output_12168933.log U
gcc_run_output_12168933.log

```
rm -f run1b run3b run8b
gcc -Ofast -fopenmp -march=native -lm -D_LLAMA_3_2_1B main.c -o run1b
gcc -Ofast -fopenmp -march=native -lm -D_LLAMA_3_2_3B main.c -o run3b
gcc -Ofast -fopenmp -march=native -lm -D_LLAMA_3_1_8B main.c -o run8b
STARTED
Using checkpoint_path: assets/llama-3_1-8b/llama-3_1-8b.bin | tokenizer_path:
assets/llama-3_1-8b/tokenizer.bin
once upon a time there was a king who had a beautiful daughter. The king was
very proud of his daughter and wanted to marry her to a prince who was as
handsome as she was beautiful. The king sent out a proclamation that any
prince who could defeat him in a game of chess would be allowed to marry his
daughter. Many princes came to the castle to try their luck, but none of them
could defeat the king. One day, a young prince named Jack arrived at the
castle. He was a skilled chess player and he knew that he had a chance to win
the game. He sat down at the chessboard and began to play. The king was
surprised to see that Jack was a good player, but he was determined to win.
He made a move and Jack countered it. The king made another move and Jack
countered it again. The game went on like this for a while, with each player
making a move and the other player countering it. Finally, the king made a
move that Jack couldn't counter. He was defeated and the king declared that
Jack could marry his daughter. Jack was overjoyed and he and the princess
lived happily ever after.

achieved tok/s: 0.129163
ENDED
```

Figure 2: LLaMA 3.1 8B Model Execution (0.13 tokens/sec)



```
rm -f run1b run3b run8b
gcc -Ofast -fopenmp -march=native -lm -D_LLAMA_3_2_1B main.c -o run1b
gcc -Ofast -fopenmp -march=native -lm -D_LLAMA_3_2_3B main.c -o run3b
gcc -Ofast -fopenmp -march=native -lm -D_LLAMA_3_1_8B main.c -o run8b
STARTED
Using checkpoint_path: assets/llama-3_2-1b/llama-3_2-1b.bin | tokenizer_path:
assets/llama-3_2-1b/tokenizer.bin
once upon a time, there was a man named John. He was a farmer, and he had a
wife and a son. One day, John went out to his field to plant some seeds. As
he was planting, he noticed that the soil was very dry. He tried to water the
seeds, but they just wouldn't take. Finally, he gave up and went home. When
he got home, he told his wife and son about his problem. They both tried to
water the seeds, but they still didn't take. Finally, John's wife said, "I
know what we can do. We'll go to the store and buy some water." So they went
to the store and bought some water. When they got home, they put the water in
a bucket and poured it on the seeds. The seeds started to grow, and soon they
were big and healthy. John's wife and son were so happy that they decided to
plant more seeds. They went out to their field again, and this time they
planted more seeds. They kept doing this for a long time, and eventually they
had a big field full of healthy plants. John's wife and son were so proud of
their hard work that they decided to plant more seeds. They went out to their
field

achieved tok/s: 2.238374
ENDED
```

Figure 3: LLaMA 3.2 1B Model Execution (2.38 tokens/sec)

The execution examples demonstrate:

- Consistent compilation flags across all models for fair comparison
- OpenMP parallelization benefits for matrix operations
- Impact of model size on generation speed
- Memory-mapped weight loading efficiency

These results highlight the challenges of running larger models on consumer hardware and emphasize the importance of optimization techniques for practical deployment scenarios.

5 Challenges and Solutions

5.1 Technical Challenges

1. **Performance Optimization**

  - Challenge: Slow execution with larger models
  - Solution: Implemented compiler optimizations and OpenMP parallelization

2. **Memory Management**
   - Challenge: OS-specific memory allocation issues
   - Solution: Implemented robust memory mapping and error handling

3. **Architecture Implementation**
   - Challenge: Complex transformer architecture understanding
   - Solution: Detailed study of research papers and reference implementations

5.2 Model-Specific Issues

- FFN dimension multiplier discrepancies in params.json
- Memory constraints with larger models

6 Future Work

The current implementation provides a foundation for running LLaMA 3.x models in pure C, but several promising areas for enhancement have been identified.

6.1 Short-term Improvements

6.1.1 Beam Search Implementation

The current implementation uses simple greedy and top-p sampling. Implementing beam search would:

- Maintain multiple candidate sequences during generation
- Allow for better exploration of the probability space
- Potentially improve output quality by considering multiple paths
- Require careful memory management to store beam states

6.1.2 Quantization Support

Adding support for quantized models would significantly improve performance:

- Implement 4-bit and 8-bit quantization for weights
- Add support for different quantization schemes (e.g., GGML format)
- Optimize matrix operations for quantized values
- Reduce memory footprint while maintaining accuracy
- Enable running larger models on consumer hardware

6.1.3 Advanced Sampling Techniques

Expand beyond current sampling methods:

- Implement nucleus sampling with dynamic temperature
- Add typical sampling for reducing repetition
- Support for contrastive search
- Add adaptive temperature scaling
- Implement frequency and presence penalties

6.1.4 Performance Optimization

Current performance metrics indicate several areas for improvement:

- Implement SIMD instructions for matrix operations
- Optimize memory access patterns
- Add support for GPU acceleration via CUDA/OpenCL
- Implement efficient batching for multiple requests
- Profile and optimize critical paths in attention computation

6.2 Long-term Goals

6.2.1 Multi-modal Capabilities

Extend the implementation to support multi-modal features of LLaMA:

- Add support for image input processing
- Implement vision encoder components
- Support for multi-modal attention mechanisms
- Handle different types of tokens (text, image, etc.)
- Integrate with vision preprocessing libraries

6.2.2 Distributed Inference

Enable distributed computation for larger models:

- Implement model parallelism across multiple devices
- Add support for pipeline parallelism
- Create efficient communication protocols between nodes
- Optimize weight sharing across distributed systems
- Implement load balancing mechanisms

6.2.3 Evaluation Framework

Develop a comprehensive evaluation system:

- Integrate standard LLM benchmarks (MMLU, TriviaQA, etc.)
- Implement automated testing pipelines
- Add support for custom evaluation metrics
- Create comparison tools against reference implementations
- Generate detailed performance reports

6.2.4 Benchmark Integration

Create a standardized benchmarking system:

- Implement HuggingFace datasets integration
- Add support for custom dataset formats
- Create automated benchmarking pipelines
- Implement result analysis and visualization
- Support for continuous performance monitoring

6.2.5 Infrastructure Improvements

Essential improvements for production deployment:

- Create a C API for external integration
- Implement proper error handling and recovery
- Add logging and monitoring capabilities
- Create documentation and usage examples
- Implement configuration management system

These improvements would significantly enhance the utility and performance of the implementation, making it more suitable for production environments and research applications. The focus on both short-term optimizations and long-term architectural improvements ensures a balanced approach to development.

7 Conclusion

This project successfully demonstrates the feasibility of implementing LLaMA 3.x inference in pure C, while highlighting both the challenges and opportunities in such implementations. The use of trie-based tokenization and various optimization techniques provides a foundation for future improvements in performance and functionality.

8 Acknowledgments

Special thanks to Andrej Karpathy for the llama2.c implementation which served as a valuable reference, and to Meta AI for making the LLaMA models publicly available.

References

[1] Vaswani, A., et al. ”Attention is all you need.” Advances in neural information processing systems 30 (2017). Available at: https://arxiv.org/abs/1706.03762

[2] Touvron, H., et al. ”The LLaMA 3 Herd of Models.” arXiv preprint arXiv:2407.21783 (2024). Available at: https://arxiv.org/abs/2407.21783

[3] Ainslie, J., et al. ”GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints.” arXiv preprint arXiv:2305.13245 (2023). Available at: https://arxiv.org/abs/2305.13245

[4] Su, J., et al. ”Round and Round We Go! What Makes Rotary Positional Encodings Useful?” arXiv preprint arXiv:2410.06205 (2024). Available at: https://arxiv.org/abs/2410.06205

[5] Karpathy, A. ”llama2.c” GitHub repository (2023). Available at: https://github.com/karpathy/llama2.c