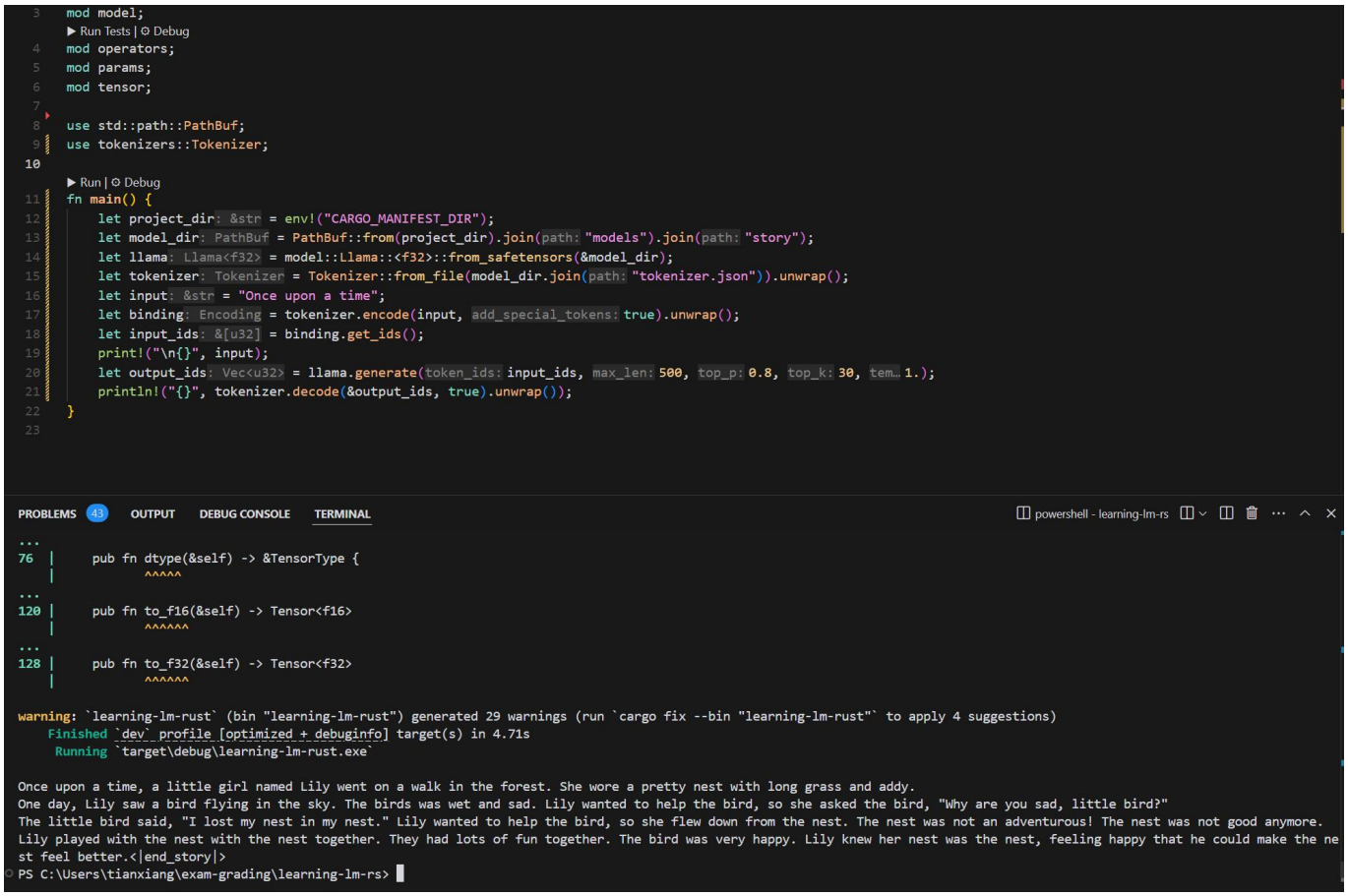
# 大模型推理系统项目报告

## 一、基础故事续写功能



```rust
mod model;
mod operators;
mod params;
mod tensor;

use std::path::PathBuf;
use tokenizers::Tokenizer;

fn main() {
    let project_dir: &str = env!("CARGO_MANIFEST_DIR");
    let model_dir: PathBuf = PathBuf::from(project_dir).join(path: "models").join(path: "story");
    let llama: Llama<f32> = model::Llama::<f32>::from_safetensors(&model_dir);
    let tokenizer: Tokenizer = Tokenizer::from_file(model_dir.join(path: "tokenizer.json")).unwrap();
    let input: &str = "Once upon a time";
    let binding: Encoding = tokenizer.encode(input, add_special_tokens: true).unwrap();
    let input_ids: &[u32] = binding.get_ids();
    print!("\n{}", input);
    let output_ids: Vec<u32> = llama.generate(token_ids: input_ids, max_len: 500, top_p: 0.8, top_k: 30, tem… 1.);
    println!("{}", tokenizer.decode(&output_ids, true).unwrap());
}
```

```
PROBLEMS 43  OUTPUT  DEBUG CONSOLE  TERMINAL                    powershell - learning-lm-rs

...
76 |     pub fn dtype(&self) -> &TensorType {
   |            ^^^^^
...
120 |     pub fn to_f16(&self) -> Tensor<f16>
    |            ^^^^^^
...
128 |     pub fn to_f32(&self) -> Tensor<f32>
    |            ^^^^^^

warning: `learning-lm-rust` (bin "learning-lm-rust") generated 29 warnings (run `cargo fix --bin "learning-lm-rust"` to apply 4 suggestions)
    Finished `dev` profile [optimized + debuginfo] target(s) in 4.71s
     Running `target\debug\learning-lm-rust.exe`

Once upon a time, a little girl named Lily went on a walk in the forest. She wore a pretty nest with long grass and addy.
One day, Lily saw a bird flying in the sky. The birds was wet and sad. Lily wanted to help the bird, so she asked the bird, "Why are you sad, little bird?"
The little bird said, "I lost my nest in my nest." Lily wanted to help the bird, so she flew down from the nest. The nest was not an adventurous! The nest was not good anymore.
Lily played with the nest with the nest together. They had lots of fun together. The bird was very happy. Lily knew her nest was the nest, feeling happy that he could make the ne
st feel better.<|end_story|>
PS C:\Users\tianxiang\exam-grading\learning-lm-rs>
```

基础的故事续写功能没有对计算进行优化，在命令行直接加载续写的结果，计算较慢，且无 UI 交互界面，考虑在拓展阶段使用混合精度优化来加快推理速度，优化内存使用，使用 egui 库搭建人机交互界面等。

# 二、拓展及优化

## （a）混合精度优化推理

我们知道，使用混合精度优化，即舍弃 FP32 类型的低位至 FP16 类型可以减轻缓存的负担加速推理，但是一些对于精度较为敏感的计算就不适于使用混合精度优化（如 RMS 以及包含三角函数的计算），这可能导致推理的精度出现较大差别。在这个项目中，混合精度优化主要体现在 operators.rs 中的 Operators 结构体实现里。

Operator 结构体主要包含以下内容：

1. 混合精度判断

```rust
impl<T: Float + Send + Sync + Default + 'static> Operators<T> {
    // 判断是否适合使用 FP16
    fn is_safe_for_fp16(values: &[T]) -> bool {
        let max_abs = values
            .iter()
            .map(|x| x.to_f32().unwrap().abs())
            .max_by(|a, b| a.partial_cmp(b).unwrap())
            .unwrap();
        // FP16的数值范围限制
        max_abs < 65504.0 && max_abs > 6.1e-5
    }
}
```

这里是在判断是否可以安全使用：

FP16 进行计算的关键函数，FP16 的数值范围限制：最大值: ±65504，最小值: ±6.1e-5，超出这个范围会导致溢出或精度损失。溢出风险：如果数值 > 65504，使用 FP16 会导致溢出。

精度损失：如果数值 < 6.1e-5，使用 FP16 会导致严重的精度损失。

2. 动态精度切换

```rust
pub fn matmul_parallel(
    c: &mut Tensor<T>,
    beta: T,
    a: &Tensor<T>,
    b: &Tensor<T>,
    alpha: T,
    transpose_b: bool,
    counter: Option<(&std::cell::Cell<usize>, &std::cell::Cell<usize>)>,
) {
    // ...
    if T::from(matrix_size).unwrap() > parallel_threshold {
        // 检查是否适合使用 FP16
        let use_fp16 = Self::is_safe_for_fp16(a.data()) && Self::is_safe_for_fp16(b.data());

        if use_fp16 && std::any::TypeId::of::<T>() == std::any::TypeId::of::<f32>() {
            // 转换为 FP16 进行计算
            let a_fp16 = Self::convert_to_fp16(a.data());
            let b_fp16 = Self::convert_to_fp16(b.data());
            // ...
        }
    }
}
```

1.矩阵判断：首先判断矩阵是否为大矩阵，矩阵乘只对大矩阵使用优化，这样可以避免小矩阵优化的开销，提高计算效率。
2.条件判断：若满足精度的条件，就进行 fp16 的计算，可以减少内存带宽，加快计算速度。
3.并行计算：使用并行计算提高效率，使用 rayon 并行库进行计算，实现多线程加速。
3.精度平衡：在进行累加时使用 FP32，在乘法时使用 FP16，这样就能保证精度平衡，保持数值稳定性。

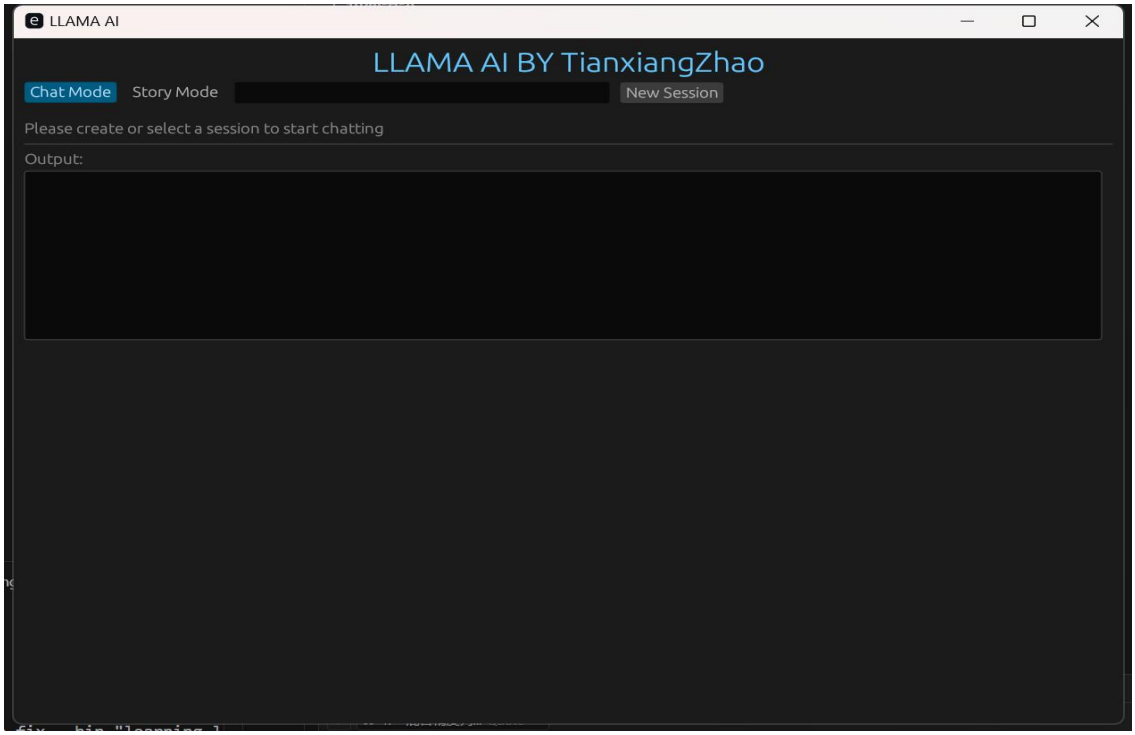
```rust
// 1. FP32 输入
let mut results_fp32 = vec![0.0f32; c_rows * c_cols];

// 2. 转换为 FP16 进行计算
results_fp32
    .par_chunks_mut(c_cols)
    .enumerate()
    .for_each(|(i, chunk)| {
        for j in 0..c_cols {
            let mut sum_fp32 = 0.0f32;
            for k in 0..a_cols {
                // FP16 计算
                sum_fp32 += a_fp16[i * a_cols + k].to_f32() * b_fp16[b_idx].to_f32();
            }
            // FP32 累加
            chunk[j] = beta.to_f32().unwrap() * c_data[i * c_cols + j]
                + alpha.to_f32().unwrap() * sum_fp32;
        }
    });

// 3. 转换回原始类型
unsafe {
    let c_data = c.data_mut();
    for (i, &val) in results_fp32.iter().enumerate() {
        c_data[i] = T::from(val).unwrap();
    }
}
```

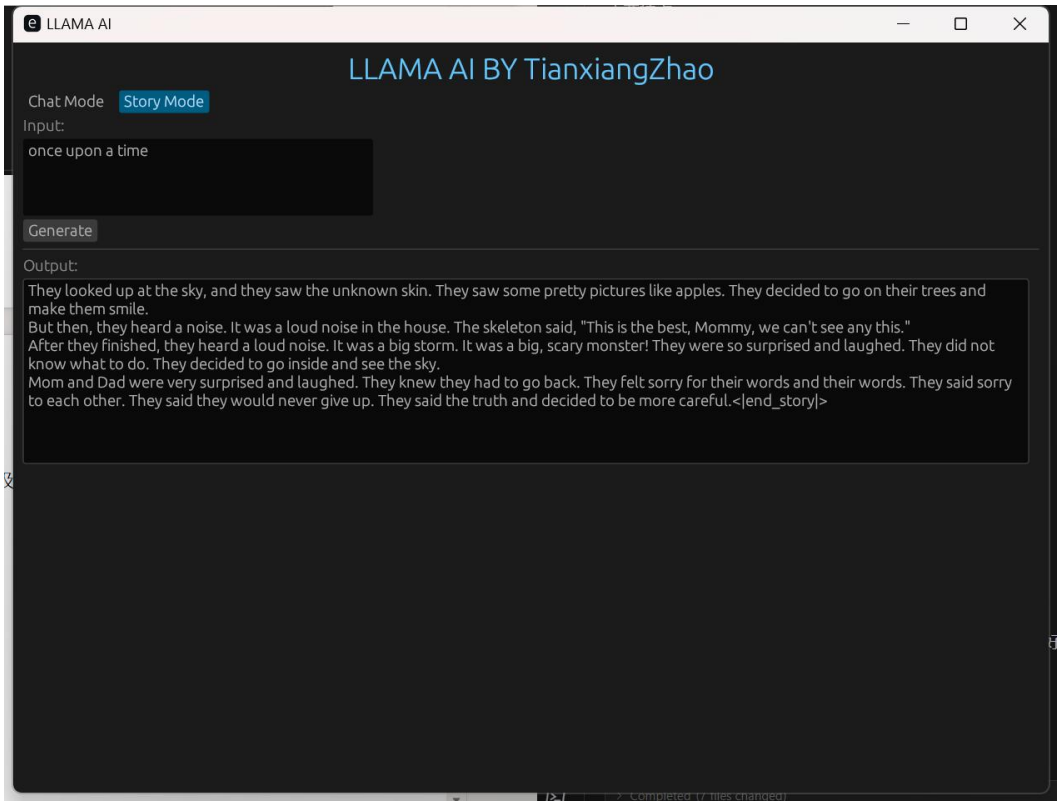
**（b）可交互 UI**

项目使用 egui 库实现交互界面，主要在 main.rs 中实现。

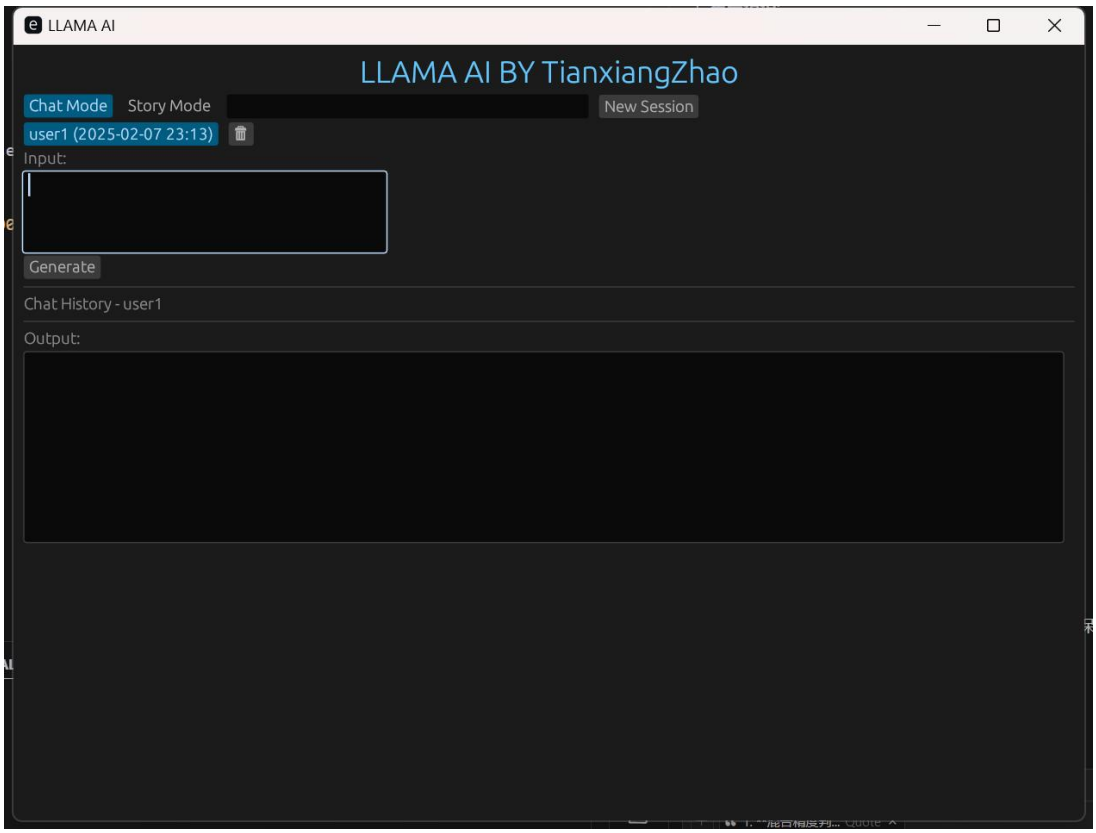



实现了 chat 和 story 两种模式，分别对应加载 chat 以及 story 模型。在界面中可以选择模式，用户输入相应的文本，模型输出相应的结果。

Story mode：

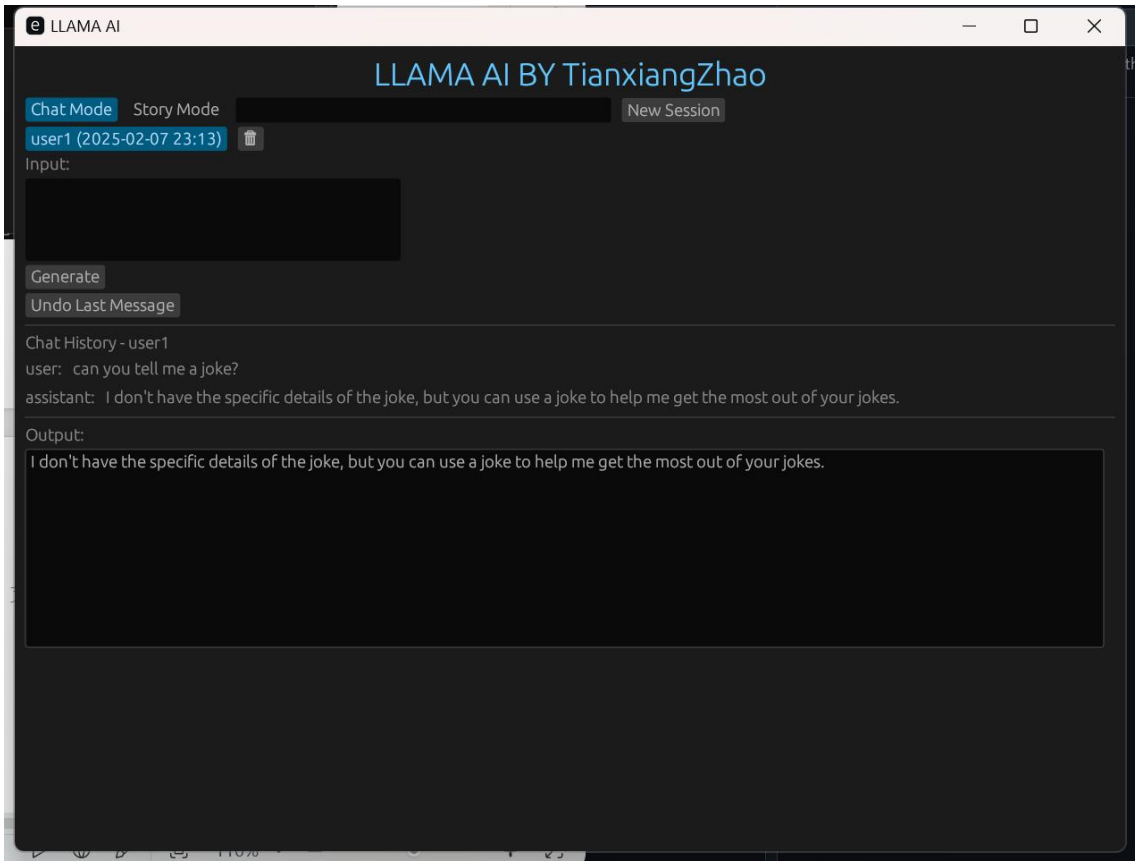



Chat mode:

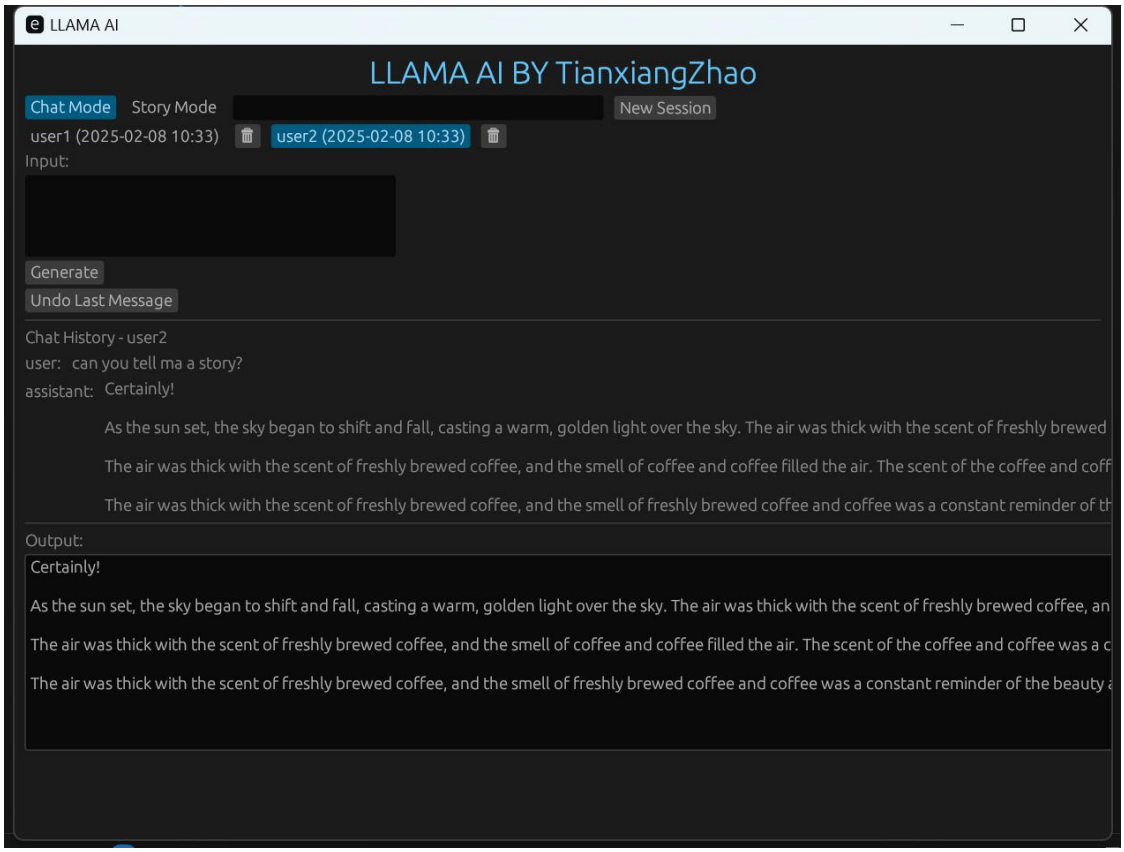



Chat mode 可以创建用户不同的主题，并记录时间，支持保存和删除记录。

Chat mode 的效果：




**（c）会话管理以及历史会话回滚**





这一项主要使用了一下关键的技术：

```rust
pub struct KVCache<T> {
    k_cache: Vec<Tensor<T>>,
    v_cache: Vec<Tensor<T>>,
    max_seq_len: usize,
}
```

```rust
// kvcache.rs
impl<T: Default + Copy + 'static> KVCache<T> {
    pub fn k_cache(&mut self, layer: usize, start: usize) -> Tensor<T> {
        // 返回缓存的视图而不是复制
        self.k_cache[layer].slice(
            start * self.dim,
            &vec![self.length - start, self.dim]
        )
    }
}
```

```rust
// 显示聊天历史
if let Some(session) = self.current_session() {
    ui.separator();
    ui.label(format!("Chat History - {}", session.name));
    for (role, content) in &session.history {
        ui.horizontal(|ui| {
            ui.label(format!("{}:", role));
            ui.label(content);
        });
    }
}
```

1. 内存优化：使用 KV 缓存机制，内存进行预分配，零拷贝操作避免大规模的数据复制，减少了内存分配，降低内存带宽压力，等实现了内存管理的优化。
2. 数据持久：会话消息保存在内存中，每个绘画有一个唯一的 ID 并且记录相应的时间。
3. 状态管理：维护了一个会话列表，显示不同用户不同时间的会话，同时支持删除对话记录。

### （d）SMID 优化矩阵乘

#### 1. 使用 AVX2 指令集

```rust
#[cfg(target_arch = "x86_64")]
use std::arch::x86_64::*;

// 运行时检测CPU特性
if is_x86_feature_detected!("avx2") {
    unsafe {
        matmul_f32_avx2(...);
    }
} else {
    matmul_fallback(...);
}
```

#### 2. 向量化操作

```rust
// 256位向量操作（一次处理8个f32）
let mut sum = _mm256_setzero_ps();          // 向量清零
let a_val = _mm256_set1_ps(*a_val);         // 广播标量到向量
let b_val = _mm256_loadu_ps(b_ptr);         // 加载向量
let result = _mm256_mul_ps(a_val, b_val);   // 向量乘法
```

未向量的标量一次只能处理一个 f32，进行向量化操作（使用 AVX2）之后每次可

以处理 8 个 f32。单条指令可以处理多个数据，提高计算吞吐量，减少指令数量。

```rust
// operators.rs
#[target_feature(enable = "avx2")]
unsafe fn matmul_f32_avx2(a: *const f32, b: *const f32, c: *mut f32, m: usize, k: usize, n: usize) {
    for i in 0..m {
        for j in (0..n).step_by(8) {  // 每次处理8个元素
            let mut sum = _mm256_setzero_ps();  // 256位向量(8个f32)

            for p in 0..k {
                // 广播单个值到8个位置
                let a_val = _mm256_set1_ps(*a.add(i * k + p));
                // 加载8个连续值
                let b_val = _mm256_loadu_ps(b.add(p * n + j));
                // 8个并行乘加操作
                sum = _mm256_add_ps(sum, _mm256_mul_ps(a_val, b_val));
            }
            // 存储8个结果
            _mm256_storeu_ps(c.add(i * n + j), sum);
        }
    }
}
```

**3. 并行优化结合**

```rust
impl<T: Float + Send + Sync + Default + 'static> Operators<T> {
    pub fn matmul_parallel(
        c: &mut Tensor<T>,
        beta: T,
        a: &Tensor<T>,
        b: &Tensor<T>,
        alpha: T,
        transpose_b: bool,
        counter: Option<(&std::cell::Cell<usize>, &std::cell::Cell<usize>)>,
    ) {
        // 大矩阵使用SIMD+并行
        if matrix_size > parallel_threshold {
            if is_x86_feature_detected!("avx2") {
                // SIMD并行实现
                results_fp32.par_chunks_mut(c_cols)
                    .enumerate()
                    .for_each(|(i, chunk)| {
                        unsafe {
                            // 使用AVX2优化的内部循环
                            matmul_f32_avx2(...);
                        }
                    });
            }
        }
    }
}
```

优化后可以充分利用 CPU 向量单元，提高指令级并行，优化缓存使用。运行时检测 CPU 特性，动态选择最优实现。

# 三、后续优化方向

项目的优化阶段曾尝试使用 CUDA 来加速计算，但 rust 内置的包一直出现 bug，正在尝试解决这个问题。对于算子的优化问题，这个项目担心对于精度会有较大误差，就仅仅对于部分矩阵乘进行了优化，后续可以考虑优化更多的算子。