К использованию LLM в системах жёсткого реального времени

Старолетов Сергей Михайлович
serg_soft@mail.ru

OSDAY 2026

5 июня 2026 г.

План доклада

1. Проблема: ИИ в реальном времени
2. Идея и платформа
3. Квантование весов
4. Как работает инференс LLM (ликбез)
5. Решение -- контроль времени на границах слоёв
6. Портирование llama2.c на POK
7. Архитектура демо
8. Улучшенный контроль ответа
9. Результаты и производительность
10. Заключение и будущее

Зачем ИИ в системах жёсткого реального времени?

- Промышленная робототехника, авионика, автономные системы
- В этой презентации -- «Умный советчик» на основе данных датчиков:
  - температура, влажность, давление, скорость ветра
  - естественно-языковой вывод: «Рекомендуется укрыть оборудование»
- Но: **нормативные документы** (DO-178C, ISO 26262) требуют
  - гарантированного **Worst-Case Execution Time** (WCET)
  - строгой временной и пространственной изоляции

Противоречие

ИИ модели считаются «непредсказуемыми по времени»
$\Rightarrow$ их внедрение в безопасно-критические системы сдерживается

Требования стандартов (DO-178C, ISO 26262)

DO-178C (авионика)

- Уровни критичности A–E
- WCET для каждой задачи
- Статическое выделение памяти (Level A)
- Детерминированное поведение

ISO 26262 (автомобили)

- ASIL A–D
- Гарантии времени отклика
- Изоляция компонентов
- Безопасность при сбоях

Главный вопрос: Можно ли встроить LLM в систему, удовлетворяющую этим требованиям?

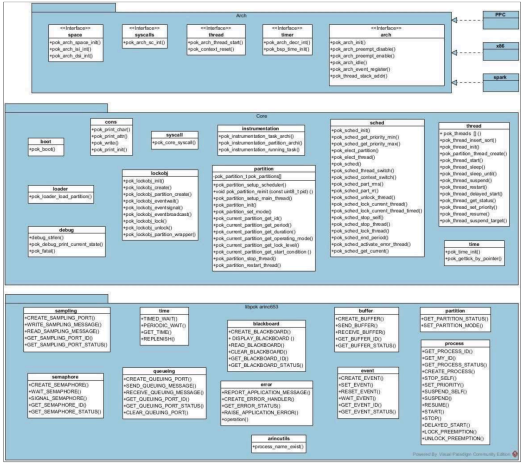
Платформа: POK + ARINC 653

- **POK** – свободное микроядро[1,2], реализующее **ARINC 653**
- ARINC 653: принцип *разделов* (partition)
  - временная изоляция: каждому разделу – фиксированный временной слот
  - пространственная изоляция: разделы не влияют на память друг друга
- Хорошо для потенциальной сертификации
- Позволяет изолировать LLM-компонент от остальной системы

[1] `https://github.com/pok-kernel/pok`

[2] Delange, Julien, and Laurent Lec. "POK, an ARINC653-compliant operating system released under the BSD license." 13th Real-Time Linux Workshop. Vol. 10. 2011.

Изучаем РОК



- Изоляция пространства данных
- Изоляция по времени
- Изоляция на ядро/пользовательский код, которые взаимодействуют через системные вызовы
- Изоляция по API: в каждый раздел загружается elf со своей стандартной библиотекой
- Поверх всего этого работает ARINC 653- интерфейс, AADL-конфигурации и т.д.

Изучаем POK

- POK идет с примерами, отдельно собираемые через Makefile
- Можно начать с `https://github.com/pok-kernel/pok/tree/main/examples/semaphores` (минимальный код многопоточки)
- Запуск на QEMU (x86-код работает и на других платформах при установке qemu-system-i386)

Мои публикации:

- Staroletov, Sergei Mikhailovich. "A formal model of a partitioned real-time operating system in Promela." Труды Института системного программирования РАН 32.6 (2020): 49-66.

Выбранная LLM, кодовая база, модель

https://github.com/karpathy/llama2.c

README MIT license

llama2.c

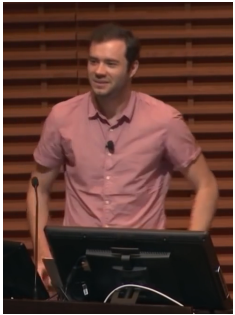
Have you ever wanted to inference a baby Llama 2 model in pure C? No? Well, now you can!

Train the Llama 2 LLM architecture in PyTorch then inference it with one simple 700-line C file (run.c). You might think that you need many billion parameter LLMs to do anything useful, but in fact very small LLMs can have surprisingly strong performance if you make the domain narrow enough (ref: TinyStories paper). This repo is a "fullstack" train + inference solution for Llama 2 LLM, with focus on minimalism and simplicity.

As the architecture is identical, you can also load and inference Meta's Llama 2 models. However, the current code only inferences models in fp32, so you will most likely not be able to productively load models larger than 7B. Work on model quantization is currently ongoing.

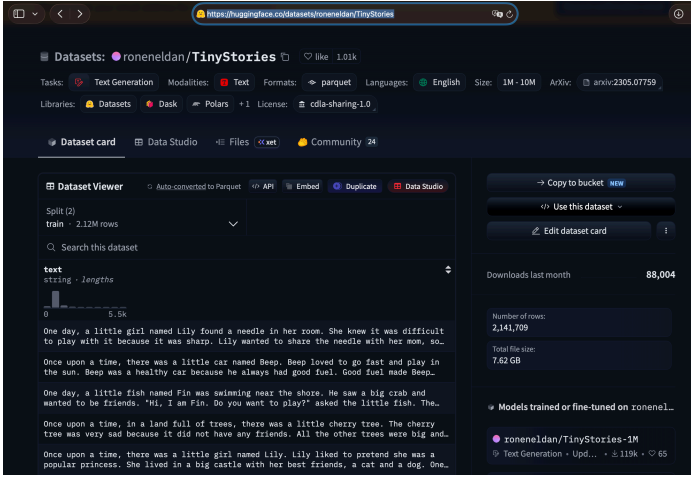
Please note that this repo started recently as a fun weekend project: I took my earlier nanoGPT, tuned it to implement the Llama-2 architecture instead of GPT-2, and the meat of it was writing the C inference engine in run.c. So the project is young and moving quickly. Hat tip to the awesome llama.cpp for inspiring this project. Compared to llama.cpp, I wanted something super simple, minimal, and educational so I chose to hard-code the Llama 2 architecture and just roll one inference file of pure C with no dependencies.

Автор небольших GPT моделей



- Андрей Карпатый (Andrej Karpathy) — словацко-канадский учёный в области машинного обучения
- Занимал должность директора по искусственному интеллекту в компании Tesla.
- Он является сооснователем и бывшим сотрудником OpenAI
- В 2026 предложил microgpt (200 строк кода на Python) `https://karpathy.github.io/2026/02/12/microgpt/`

Выбранная LLM, кодовая база, модель



Eldan, Ronen, and Yuanzhi Li. "Tinystories: How small can language models be and still speak coherent English?." arXiv preprint arXiv:2305.07759 (2023).

Выбранная LLM, кодовая база, модель

Модель: TinyStories 15M

- 15 миллионов параметров
- dim=288, hidden_dim=768, n_layers=6, n_heads=6
- словарь 32000 токенов
- **Квантование до int8** (веса)
- Размер модели $\approx$ 15 МБ

Исполняемый код: llama2.c

- Andrej Karpathy, чистый C
- Минимальные зависимости
- Может работать на слабом железе
- Легко портируется в микроядерную среду

Как встроена модель в РОК-окружение

Веса сконвертированы в статический массив внутри .h-файла и прилинкованы к ELF-образу раздела.

Выбранная LLM, кодовая база, модель

Как встроена модель в РОК-окружение

Веса модели сконвертированы в статический массив внутри .h-файла и прилинкованы к ELF-образу раздела.

```
static const uint8_t model_weights[] = {
    0x32, 0x34, 0x6D, 0x51, 0x02, 0x00, 0x00, 0x00, 0x20, 0x01, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x01, 0x00, 0x00, 0x00, 0x00, 0x7D, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
    0x01, 0x38, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00
```

Квантование, точность vs. ресурсы

- Нейросети обычно тренируются в **float32** (4 байта на вес)
- 15M параметров $\times$ 4 байта $\approx$ 60 МБ (без учёта эмбеддингов)
- Для IoT/встраиваемого реального времени:
  - ОЗУ 15–20 МБ – роскошь
  - Кэш процессора меньше, промахи дороже
  - Операции с плавающей точкой могут быть непредсказуемыми (разное время)
- **Решение:** пожертвовать точностью ради размера и предсказуемости

Квантование

Преобразование чисел с плавающей точкой (float32) в целые числа меньшей разрядности (int8).
Выигрыш: $\times 4$ экономия памяти, целочисленная арифметика $\rightarrow$ стабильнее время вычислений, работа на устройствах без

Симметричное int8 квантование (упрощённо)

Прямое квантование (float → int8)

$$q = \mathrm{round}\left(\frac{r}{S}\right), \quad S = \frac{\max(|r|)}{127}$$

- r – исходное float значение
- S – масштаб (шкала), общий для группы весов
- q – квантованное значение (int8, $-127 \cdots +127$)

Деквантование (int8 → float) при вычислениях

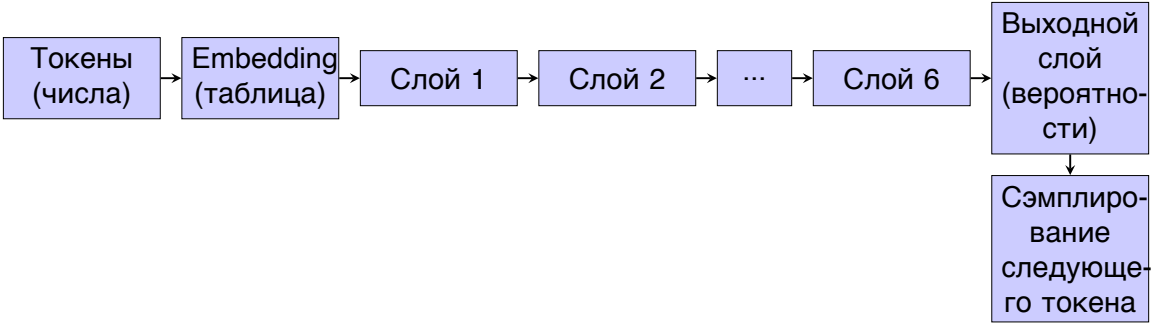
$$r \approx S \cdot q$$

- Храним только q и шкалу S
- При умножении матриц: можно работать прямо в целых числах, а масштабирование применить позже

Пример

Исходные веса: $[0.2, -0.5, 0.8]$, $\max = 0.8$, $S = 0.8/127 \approx 0.0063$
Квантованные: $\mathrm{round}([31.8, -79.4, 127.0]) = [32, -79, 127]$

LLM «под капотом»



- Каждый **прямой проход** генерирует ровно один новый токен
- Слои трансформера (внимание + MLP) выполняются последовательно
- **Время обработки одного слоя при фиксированной длине контекста постоянно!**

Токенизатор

- Модель «понимает» не слова, а **токены** — целые числа от 0 до `vocab_size-1`
- Токенизатор **детерминирован**: одному и тому же тексту всегда соответствует одна и та же последовательность чисел
- В llama2.c используется алгоритм **Byte-Pair Encoding (BPE)** с фиксированным словарём размером 32 000 токенов
- Процесс:
  1. Текст разбивается на подстроки по таблице слияний
  2. Каждая подстрока ищется в словаре → получает числовой идентификатор
- Время токенизации **пропорционально длине входной строки** и не зависит от смысла

Токенизатор

```
static const uint8_t tokenizer_data[] = {
    0x1B, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00,
    0x3C, 0x75, 0x6E, 0x6B, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00,
    0x00, 0x0A, 0x3C, 0x73, 0x3E, 0x0A, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00,
    0x00, 0x00, 0x0A, 0x3C, 0x2F, 0x73, 0x3E, 0x0A, 0x00, 0x00, 0x00, 0x00,
    0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x30, 0x30, 0x3E, 0x00, 0x00,
    0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x30, 0x31, 0x3E,
    0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x30,
    0x32, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30,
    0x78, 0x30, 0x33, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00,
    0x3C, 0x30, 0x78, 0x30, 0x34, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00,
    0x00, 0x00, 0x3C, 0x30, 0x78, 0x30, 0x35, 0x3E, 0x00, 0x00, 0x00, 0x00,
    0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x30, 0x36, 0x3E, 0x00, 0x00,
    0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x30, 0x37, 0x3E,
    0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x30,
    0x38, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30,
    0x78, 0x30, 0x39, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00,
    0x3C, 0x30, 0x78, 0x30, 0x41, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00,
    0x00, 0x00, 0x3C, 0x30, 0x78, 0x30, 0x42, 0x3E, 0x00, 0x00, 0x00, 0x00,
    0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x30, 0x43, 0x3E, 0x00, 0x00,
    0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x30, 0x44, 0x3E,
    0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x30,
    0x45, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30,
    0x78, 0x30, 0x46, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00,
    0x3C, 0x30, 0x78, 0x31, 0x30, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00,
    0x00, 0x00, 0x3C, 0x30, 0x78, 0x31, 0x31, 0x3E, 0x00, 0x00, 0x00, 0x00,
    0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x31, 0x32, 0x3E, 0x00, 0x00,
    0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x31, 0x33, 0x3E,
    0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x31,
    0x34, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30,
    0x78, 0x31, 0x35, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00,
    0x3C, 0x30, 0x78, 0x31, 0x36, 0x3E, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00,
    0x00, 0x00, 0x3C, 0x30, 0x78, 0x31, 0x37, 0x3E, 0x00, 0x00, 0x00, 0x00,
    0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x31, 0x38, 0x3E, 0x00, 0x00,
    0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x31, 0x39, 0x3E,
    0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x3C, 0x30, 0x78, 0x31,
```

tokenizer.bin(400kb) -> tokenizer_data.h (2.5M)

Embedding -- превращаем токен в вектор

- Каждому токену t соответствует вектор фиксированной размерности $d_{model} = 288$
- Матрица эмбеддингов $E \in \mathbb{R}^{32000 \times 288}$ (квантована до int8)
- Операция: $\mathbf{x} = E[t]$ (просто чтение строки таблицы)
- В нашем решении: **вычисляем эмбеддинг «на лету»** без хранения всей распакованной float-таблицы

Предсказуемость

Извлечение одного эмбеддинга — это:

- адресная арифметика (базовый адрес + $t \cdot d_{model}$)
- чтение 288 последовательных байт (или их деквантизация)

Время **строго постоянно** для любого токена.

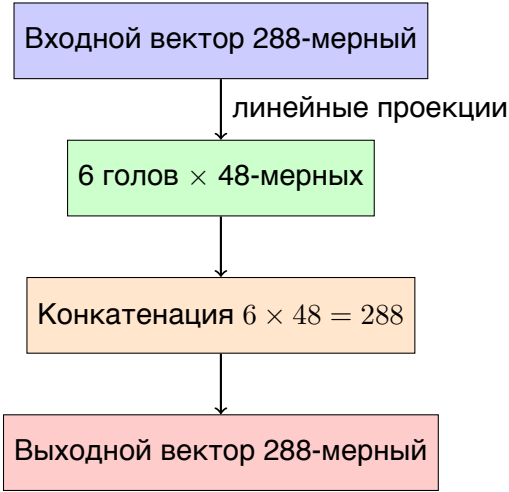
Embedding

```
static void get_token_embedding(TransformerWeights* w, int
   vocab_size, int dim,
int token, float* out) {
 QuantizedTensor* qt = w->q_tokens;
 int offset = token * dim;
 for (int i = 0; i < dim; i++) {
  out[i] = (float)(qt->q[offset + i]) * qt->s[(offset + i) / GS];
 }
}
```

Головки внимания (Multi-Head Attention)

- Одна голова внимания учится «смотреть» на одну взаимосвязь между токенами
- Несколько голов ($n_{heads} = 6$) анализируют **разные отношения**, например:
  - синтаксические (подлежащее–сказуемое)
  - семантические (объект–свойство)
  - позиционные (соседние токены)
  - и т.д.
- Результаты всех голов объединяются и линейно проецируются
- **Все головы имеют одинаковую размерность**
  $d_{head} = d_{model}/n_{heads} = 288/6 = 48$

Головки внимания (Multi-Head Attention)



Головки внимания (Multi-Head Attention)

Для каждого токена в последовательности длины T:

$$\mathbf{q}_i = W^Q \mathbf{x}_i, \quad \mathbf{k}_i = W^K \mathbf{x}_i, \quad \mathbf{v}_i = W^V \mathbf{x}_i$$

- W^Q, W^K, W^V — матрицы размера 48×288 (целочисленные)
- Затем вычисляются **скалярные произведения** между запросами и ключами:

$$\text{score}_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{48}}$$

- Матрица $\mathbf{S}$ размера $T \times T$ пропускается через `softmax` по строкам
- Результат взвешивает векторы значений: $\mathbf{o}_i = \sum_j \text{softmax}(\mathbf{S})_{ij} \mathbf{v}_j$

Оценка

Все матрицы — **константного размера**. Циклы всегда выполняют одно и то же число умножений и сложений. Нет ветвлений, зависящих от данных.

Головки внимания

```
for (h = 0; h < p->n_heads; h++) {
 float* q = s->q + h * head_size;
 float* att = s->att + h * p->seq_len;
 for (int t = 0; t <= pos; t++) {
  float* k = s->key_cache + loff + t * kv_dim + (h / kv_mul) *
     head_size;
  float score = 0.0f;
  for (int i = 0; i < head_size; i++) score += q[i] * k[i];
  score /= sqrtf(head_size);
  att[t] = score;
 }
 softmax(att, pos + 1);
 float* xb = s->xb + h * head_size;
...
}
```

Головки внимания

```
{ ...
memset(xb, 0, head_size * sizeof(float));
for (int t = 0; t <= pos; t++) {
 float* v = s->value_cache + loff + t * kv_dim + (h / kv_mul) *
    head_size;
 float a = att[t];
 for (int i = 0; i < head_size; i++) xb[i] += a * v[i];
}
```

Головки внимания

```
void softmax(float* x, int size) {
 float max_val = x[0];
 for (int i = 1; i < size; i++) i
 if (x[i] > max_val) max_val = x[i];
 float sum = 0.0f;
 for (int i = 0; i < size; i++) { x[i] = expf(x[i] - max_val);
    sum += x[i]; }
 for (int i = 0; i < size; i++) x[i] /= sum;
}
```

Отсутствие условных ветвлений (кроме сэмплирования)

- Внутри трансформерных слоёв используются только:
  - матричные умножения над массивами фиксированного размера
  - поэлементные нелинейности (SwiGLU, softmax), которые реализованы без `if`
  - сложение векторов, нормализация (LayerNorm) — всё предсказуемо
- Нет циклов с переменным числом итераций, зависящим от значений
- Нет поиска, сортировки, switch-case по содержимому
- **Единственное место ветвления:** выбор следующего токена
  - Жадный выбор (`argmax`) — детерминирован и требует фиксированного числа сравнений (`vocab_size`)
  - Случайное семплирование — тоже выполняется за константное время

Sampling

```
int sample(Sampler* sampler, float* logits) {
 if (sampler->temperature == 0.0f) {
  next = sample_argmax(logits, sampler->vocab_size);
 } else {
  for (int q=0; q<sampler->vocab_size; q++) logits[q] /= sampler
     ->temperature;
  softmax(logits, sampler->vocab_size);
  float coin = random_f32(&sampler->rng_state);
  if (sampler->topp <= 0 || sampler->topp >= 1) {
   next = sample_mult(logits, sampler->vocab_size, coin);
  } else {
   next = sample_topp(logits, sampler->vocab_size, sampler->topp,
       sampler->probindex, coin);
  }
 }
```

Отсутствие условных ветвлений (кроме сэмплирования)

Итог

Время прямого прохода одного токена можно **точно рассчитать** аналитически или измерить как WCET.
Это и даёт квант для нашего механизма проверки дедлайнов.

Почему время слоя постоянно?

- Все операции внутри слоя: матричные умножения фиксированного размера
- Нет условных ветвлений, зависящих от данных (кроме сэмплирования на выходе)
- Квантованные целочисленные вычисления – ещё стабильнее
- Прогнозируемая длительность обработки одного слоя T_{layer}

Основная идея решения

Использовать границы слоёв как **точки кооперативной проверки** соблюдения дедлайна!

Механизм «слой – проверка времени»

1. Получаем начальный промпт (вопрос к модели)
2. Запускаем авторегрессионную генерацию: **токен за токеном**
3. При генерации каждого токена идём по слоям:
   - перед входом в следующий слой вызываем `pok_time_get()`
   - сравниваем с установленным дедлайном
   - если дедлайн превышен – **немедленно завершаем генерацию**

Что это даёт?

Поток **гарантированно** уступит управление не позже, чем через T_{layer} после превышения дедлайна.
WCET задачи $= \{T_{prompt_processing} + k \cdot T_{layer}\}$ ограничена

MVP: Генерация с контролем времени

```
void generate(Transformer *transformer, Tokenizer *tokenizer,
   Sampler *sampler, char *prompt, int steps) {
 transformer->deadline_exceeded = 0;
 while (pos < steps) {
  float* logits = forward(transformer, token, pos);
  if (transformer->deadline_exceeded) {
   printf("\n[TRUNCATED at %d tokens due to deadline]\n", pos);
   break;
  }
  next = sample(sampler, logits);
  pos++;
  char* piece = decode(tokenizer, token, next);
  printf("%s", piece);
  token = next;
 }
```

MVP: Генерация с контролем времени

```
float* forward(Transformer* transformer, int token, int pos) {
 get_token_embedding(w, p->vocab_size, dim, token, x);
 for(int l = 0; l < p->n_layers; l++) {
        // attention rmsnorm
        // qkv matmuls
        // RoPE
        // KV cache
        // multihead attention
        // ffn rmsnorm
        // SwiGLU
  pok_time_t ns; pok_time_get(&ns);
  if (transformer->deadline_us > 0 && ns >= transformer->
     deadline_us) {
   transformer->deadline_exceeded = 1; return s->logits;
  }
```

Преимущества кооперативного подхода

- Не нужно вытеснять поток (preemption) – добровольная проверка
- Не нарушается внутреннее состояние модели
- Нет гонок данных: синхронизация встроена
- Возможность мягкого дедлайна (см. далее)
- Подходит для сертифицируемых систем, где прерывания могут быть ограничены

Трудности портирования и их преодоление

Проблема 1: Де-квантизация embedding-таблицы

- Оригинальный код: после загрузки сразу превращает int8 в float32 $\Rightarrow$ 32000 $\times$ 288 $\times$ 4 байта $\approx$ 36 МБ дополнительной памяти
- Менеджер кучи РОК не мог выделить такой блок
- **Решение:** полностью отказались от де-квантизированной таблицы **Вычисляем embedding «на лету»** при каждом прямом проходе

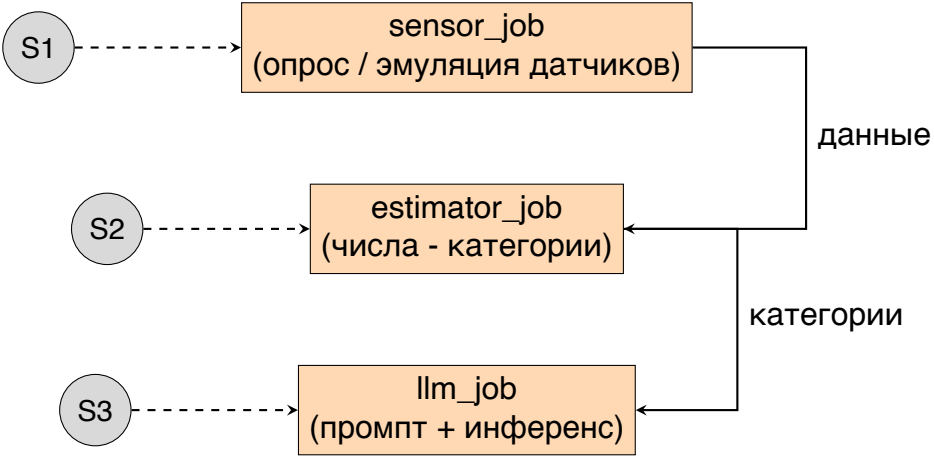
Проблема 2: Динамическая память

- Код использовал `malloc` на этапе инициализации
- **Решение:** замена на статические массивы и арифметику указателей $\Rightarrow$ память выделяется на этапе компиляции

Дополнительные изменения

- `libc`: в РОК отсутствовали `qsort`, `bsearch`, `sprintf`...
- Нужно было добавить минимальные аналоги
- Сборка в единый ELF-образ раздела РОК
- Модель встроена как статический массив `const unsigned char model_data[]`
- Полный отказ от динамического выделения памяти после инициализации

Три потока в одном разделе



- Синхронизация через семафоры POK
- Каждые N миллисекунд: опрос → категоризация → генерация совета

Пример рабочего цикла

(Так хотел, но нужна другая модель, stories оказалось не подходит)

1. `sensor_job`:
   - эмулирует датчики (случайные числа + скользящее среднее)
2. `estimator_job`:
   - температура 35°C → «жарко»
   - влажность 20% → «сухо»
   - ветер 15 м/с → «ветрено»
3. `llm_job (упрощенно, нужна обученная модель)`:
   - промпт: «Температура жарко, влажность сухо, ветер ветрено. Какой совет?»
   - генерация: «Рекомендуется обильный полив и защита от ветра»

Двухступенчатый дедлайн для грамматической завершенности

Проблема

Простое прерывание может оставить ответ синтаксически незавершённым «`Рекомендуется обильный пол`»

Решение

1. **Мягкий дедлайн** – за $X\%$ времени до жёсткого дедлайна
2. Разрешаем модели выдать ещё не более K токенов (например, 3–5)
3. Если в этом окне встречается знак конца предложения – завершаем чисто
4. Иначе – заканчиваем по жёсткому дедлайну как есть

Демонстрация (скриншот)

```
[Estimator] classified temp=26 pressure=735 humidity=53 wind_speed=10

--- scheduling partition: 0, low:0, high:10
--- Scheduling processor: 0
    scheduling thread 2 (priority 42)
elected 2 !!!
    non-ready: 0 (1/stopped), 1 (40/waiting), 3 (42/lock), 4 (0/stopped), 5 (0/stopped), 6 (0/stopped), 7 (0/stopped), 8 (0/sto

[Sensor] get temperature 25 C
[Sensor] get humidity 53 %
[Sensor] get pressure 733 mm
[Sensor] get wind speed 8 m/s

--- scheduling partition: 0, low:0, high:10
--- Scheduling processor: 0
    scheduling idle thread
    non-ready: 0 (1/stopped), 1 (40/waiting), 2 (42/waiting), 3 (42/lock), 4 (0/stopped), 5 (0/stopped), 6 (0/stopped), 7 (0/st
--- scheduling partition: 0, low:0, high:10
--- Scheduling processor: 0
    scheduling thread 2 (priority 42)
elected 2 !!!
    non-ready: 0 (1/stopped), 1 (40/waiting), 3 (42/lock), 4 (0/stopped), 5 (0/stopped), 6 (0/stopped), 7 (0/stopped), 8 (0/sto
--- scheduling partition: 0, low:0, high:10
--- Scheduling processor: 0
    scheduling idle thread
    non-ready: 0 (1/stopped), 1 (40/waiting), 2 (42/lock), 3 (42/lock), 4 (0/stopped), 5 (0/stopped), 6 (0/stopped), 7 (0/stopp
--- scheduling partition: 0, low:0, high:10
--- Scheduling processor: 0
    scheduling thread 1 (priority 40)
elected 1 !!!
    non-ready: 0 (1/stopped), 2 (42/lock), 3 (42/lock), 4 (0/stopped), 5 (0/stopped), 6 (0/stopped), 7 (0/stopped), 8 (0/stoppe
[LLM] generation:
Temperature is hot and pressure is comfortable and wind is windy and humidity is comfortable. What to do?
Her mom had a surprise for her. She brought out a bag of colorful popcorn and said, "Let's all eat this popcorn together!"
Alice was very excited and said, "Yay!" They all shared the popcorn and enjoyed it in the sunshine.
[.]
[TRUNCATED at 91 tokens due to deadline]
```

Что мы получили

- Демо работает под QEMU (i386)
- Модель TinyStories 15M int8 внутри раздела POK
- Полный цикл с контролем дедлайна

Эксперимент	Скорость генерации
Raspberry Pi (Cortex-A)	2.18 токенов/сек

LLM способна работать в рамках секундных дедлайнов

А как же микросекундные дедлайны?

Типичные области применения LLM-советчика

- Интерпретация показаний датчиков на естественном языке
- Формирование рекомендаций оператору (человеку)
- Логирование и диагностика сложных событий

Во всех этих сценариях **допустимые задержки — от 100 мс до единиц секунд**, а не микросекунды.

Главное — не абсолютная скорость, а предсказуемость

Мы показали, как **доказательно ограничить WCET** для инференса LLM. Механизм «проверка на границе слоя» не зависит от того, сколько длится слой. Он гарантирует соблюдение любого наперёд заданного дедлайна.

Дедлайны и гарантии

Когда нужны нано-/микросекунды

- Замкнутый контур управления двигателем
- Обработка прерываний от датчиков

→ LLM здесь не применяется

Когда нужны миллисекунды/секунды

- Анализ ситуации
- Совет человеку
- Текстовый вывод для бортового журнала

→ LLM работает, **гарантированно укладываясь**

Выводы

- LLM **может** быть адаптирована к парадигме жёсткого реального времени
- Основные элементы решения:
  - статическая память и отказ от динамического выделения
  - квантованные целочисленные вычисления
  - кооперативная проверка дедлайна на границах слоёв
  - временная изоляция средствами ARINC 653
- Можно получить предсказуемую верхняя граница времени выполнения

Вопросы?

Старолетов С.М.
serg_soft@mail.ru
https://github.com/SergeyStaroletov/LLM-POK