**МИНИСТЕРСТВО ОБРАЗОВАНИЯ КРАСНОЯРСКОГО КРАЯ**
**КРАЕВОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ**
**«АЧИНСКИЙ ТЕХНИКУМ НЕФТИ И ГАЗА ИМЕНИ Е.А. ДЕМЬЯНЕНКО»**

# КУРСОВАЯ РАБОТА

ПМ.01 Разработка модулей программного обеспечения для компьютерных систем

## Тема: Разработка движка «Xenith» с компактным хранением GGUF-весов на Vulkan

Выполнил: _____

Проверил: _____

Ачинск, 2026

Министерство образования Красноярского края
Краевое государственное автономное профессиональное образовательное учреждение
«Ачинский техникум нефти и газа имени Е.А. Демьяненко»

# ЗАДАНИЕ

на выполнение курсовой работы

Рассмотрено на заседании предметно-цикловой комиссии математических и естественно-научных дисциплин, протокол № ___ от «___» _____ 2026 г.
Председатель _____

«УТВЕРЖДАЮ»
зам. директора по УПР
_____
«___» _____ 2026 г.

**Содержание:**

Исследовать архитектуру Xenith и пути загрузки GGUF-моделей. Сравнить решение с llama.cpp по исходникам. Спроектировать и реализовать packed-хранилище Tensor, прямые GPU-форматы f32, f16, bf16, q8_0, q4_0, q4_1, compute-шейдеры GEMM/embedding и единый параметр weight_type в CLI, Python и Ollama API. Проверить корректность упаковки, сохранить FP32-границу обучения и подготовить схемы, примеры кода и базовые измерения памяти.

Дата выдачи задания: «___» _____ 2026 г.

Срок сдачи: «___» _____ 2026 г.

Руководитель работы: _____

Студент: _____

# СОДЕРЖАНИЕ

**ВВЕДЕНИЕ** **4**
**1 Общая часть** **6**
1.1 Назначение и цель проекта 6
1.2 Требования к системе 7
1.3 Архитектура Xenith 8
1.4 Форматы моделей и квантования 9
1.5 Бэкенды CPU и Vulkan 11
**2 Специальная часть** **13**
2.1 Анализ проблемы и постановка задачи 13
2.2 Сравнение Xenith и llama.cpp на уровне исходников 14
2.3 Исследуемая модель MiniCPM5 15
2.4 Проектирование хранилища Tensor 17
2.5 Изменения загрузчика GGUF 18
2.6 Vulkan-шейдеры и pinned-буферы 20
2.7 Интерфейсы CLI, Python и Ollama 22
2.8 Создание модели и выбор формата 23
2.9 Учёт памяти и бюджет RX 570 24
2.10 CPU fallback и граница обучения 26
2.11 Базовая экспериментальная проверка 27
2.12 Тестирование и оценка результатов 28
**ЗАКЛЮЧЕНИЕ** **30**
**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ** **31**
**ПРИЛОЖЕНИЕ А** **33**
**ПРИЛОЖЕНИЕ Б** **42**

# ВВЕДЕНИЕ

Современные локальные языковые модели требуют не только корректной реализации трансформера, но и рационального управления памятью. Файл GGUF уже содержит квантованные веса, однако naïve-загрузка может деквантовать их в FP32 и тем самым потерять основной выигрыш формата. Для модели примерно на миллиард параметров это означает несколько гигабайт дополнительной оперативной памяти ещё до создания активаций, KV-кэша и буферов графического ускорителя. Проблема особенно заметна на домашних системах с ограниченным объёмом VRAM, где часть ресурсов драйвер вынужден размещать в GTT.

Xenith развивается как единый движок на C++17: он поддерживает создание и обучение моделей, генерацию, Python-библиотеку, Ollama-совместимый HTTP-сервер и агентный слой. Такая интеграция удобна для исследований, потому что один проект охватывает полный жизненный цикл модели. Вместе с тем инженерная задача компактного GPU-хранения имеет собственную специфику: необходимо сохранить точную раскладку GGUF, не смешать логические и физические размеры тензора, обеспечить декодирование в compute-шейдере и не сломать проверенные CPU-пути.

Актуальность работы обусловлена ростом размеров открытых моделей и распространением локального инференса. Пользователь ожидает, что выбор Q8_0 или Q4_0 уменьшит не только файл на диске, но и рабочее множество в памяти. Если движок всё равно разворачивает матрицы в FP32, семантика формата выполняется лишь частично. Поэтому packed-режим следует рассматривать как сквозное свойство загрузчика, тензора, бэкенда и интерфейсов, а не как отдельную функцию квантования.

Цель курсовой работы — спроектировать и реализовать в Xenith путь сохранения GGUF-весов в компактном виде при инференсе на Vulkan, включая единый выбор формата через CLI, Python и Ollama API. Для достижения цели необходимо решить несколько взаимосвязанных задач: проанализировать текущую архитектуру и выявить место полной деквантации; сравнить подход Xenith с llama.cpp на уровне исходников; ввести метаданные packed-раскладки; добавить прямые GPU-форматы и шейдеры; унифицировать параметры интерфейсов; описать учёт памяти, CPU fallback и границу обучения; проверить решение доступными автоматическими тестами.

Объектом исследования является процесс загрузки и выполнения квантованных трансформерных моделей. Предметом — программная реализация компактного хранения тензоров и их использования в Vulkan-бэкенде Xenith. Практическая значимость состоит в возможности запускать модели на ограниченных по памяти конфигурациях, сохраняя при этом совместимость с GGUF и не изменяя файл модели при выборе внутреннего формата.

В работе используется модель MiniCPM5-1B-Claude-Opus-Fable5-V2-Thinking-Q8_0.gguf как представитель dense LLaMA-подобной архитектуры с GQA. Её карточка содержит 219 тензоров: 170 матриц Q8_0 и 49 векторов F32. Такое соотношение удобно для проверки гипотезы: большая часть параметров может оставаться упакованной, а нормы, требующие повышенной точности, продолжают храниться в FP32. Измерения, приведённые в специальной части, явно отделены как baseline до реализации нового packed-пути; пост-реализационные показатели не выдумываются и должны быть получены отдельным воспроизводимым экспериментом.

# 1 Общая часть

## 1.1 Назначение и цель проекта

Xenith предназначен для локальной работы с трансформерными моделями: от создания словаря и первичного обучения до диалога, пакетной генерации и вызова через сетевой API. В отличие от узкой библиотеки только для инференса, проект объединяет вычислительное ядро, тренировочный контур, интерфейсы и дополнительные агентные механизмы. Это определяет архитектурное требование: новое хранилище не должно превращать инференс в единственный режим существования модели.

Компактное хранение весов решает сразу несколько задач. Во-первых, оно уменьшает объём данных, передаваемых из host-памяти в Vulkan. Во-вторых, позволяет избежать постоянной FP32-копии всех матриц. В-третьих, переносит деквантование ближе к месту потребления значения: шейдер читает блок Q8_0 или Q4_0 и немедленно использует его в умножении. Наконец, метаданные физической раскладки дают возможность работать с исходным GGUF без дорогостоящего преобразования каждой матрицы.

Важно различать формат файла и формат в памяти. Исходный Q8_0-файл не обязан быть перезаписан, чтобы пользователь выбрал q4_0 для текущего запуска. Параметр weight_type относится к внутреннему представлению загруженной модели. При source система сохраняет тип каждого исходного тензора, при явном значении выполняется перепаковка в выбранный поддерживаемый тип. Файл на диске остаётся неизменным, а результат зависит от политики загрузки.

Целью изменений является устранение лишнего полного FP32-unpack на supported dense-пути без потери совместимости и без распространения

packed-режима на обучение. Нормы, векторы смещений, сложные fused- и stacked-тензоры, а также архитектуры, не обеспеченные прямыми шейдерами, должны продолжать работать через проверенный FP32-путь. Такой консервативный выбор предпочтительнее молчаливого включения неправильного ядра.

Таблица 1 – Требования к packed-режиму

| Требование | Реализация / проверка |
|---|---|
| Совместимость | GGUF v1–v3, типы ggml, существующие CLI и API |
| Точность раскладки | logical [in][out], physical [out][in], флаг RoPE |
| Прямые GPU-типы | f32, f16, bf16, q8_0, q4_0, q4_1 |
| Нормы | всегда F32; малоразмерные векторы не квантуются |
| Отказоустойчивость | fallback к FP32 при неподдержанной форме или типа |
| Обучение | packed отключён; .xnh и master weights остаются FP32 |
| Наблюдаемость | журнал числа тензоров, форматов и packed-байтов |

## 1.2 Требования к системе

Функциональные требования включают загрузку .xnh, GGUF и safetensors, выполнение генерации на CPU или Vulkan, работу с контекстом и GQA, а также доступ к тем же возможностям через CLI, Python и HTTP. Новый параметр должен иметь одинаковые допустимые значения во всех интерфейсах. Значение source является режимом загрузки по умолчанию, поскольку оно наименее изменяет исходные данные и позволяет сохранить уже выбранный автором модели уровень квантования.

Нефункциональные требования связаны с памятью, производительностью и безопасностью. Packed-режим не должен выделять FP32-буфер размером n×4 для каждого тензора без необходимости. Время загрузки может увеличиваться при явной перепаковке, но это осознанный обмен на меньший рабочий набор. Ошибки размеров, обрезанные данные и

неподдерживаемые типы должны приводить к понятной диагностике, а не к чтению за границами буфера.

Средой разработки является Linux с C++17-компилятором, CMake, Python и необязательным Vulkan SDK. Для документооборота используется LibreOffice, для проверки PDF — pdfinfo, pdftotext и pdftoppm. Экспериментальная конфигурация описана в разделе 2.11: AMD Radeon RX 570 с 4 ГиБ VRAM, RADV Mesa 26.0.8 и 16 ГиБ оперативной памяти. Эти параметры важны, поскольку поведение GTT и резидентность mmap зависят от драйвера и настроек ОС.

Отдельное требование — воспроизводимость результатов. Генератор документов читает шаблоны только для оформления, сохраняет исходные фоновые JPEG презентации побайтно и строит PDF из готовых DOCX/PPTX. Исходные схемы копируются в out/, а PNG-версии создаются LibreOffice. Это позволяет повторить сборку без ручного редактирования PowerPoint или Word.

### 1.3 Архитектура Xenith

Ядро Xenith разделено на представление модели, вычислительные операции, бэкенды и прикладные интерфейсы. Config хранит размерности, число слоёв и голов, тип архитектуры, параметры RoPE и FFN. Params объединяет веса, градиенты и состояние оптимизатора. Tensor обеспечивает доступ к элементам и управляет владеющим FP32-буфером, mmap-отображением либо packed-байтами. Tokenizer отвечает за словарь, BPE и чат-форматирование.

Матрицы слоёв в Xenith логически хранятся как [in][out], что удобно для операции x·W. GGUF и HF Linear используют физическую ориентацию

[out][in]. Ранее загрузчик исправлял несовпадение транспозицией и деквантацией. Packed-путь сохраняет физические байты и добавляет признак transposed. Тогда логический Tensor по-прежнему возвращает ожидаемую форму при host-доступе, а GPU-шейдер сам выбирает brow и bcol.

Backend abstractions скрывают различия CPU и Vulkan. CPU-реализация использует многопоточные ядра и работает с FP32. Vulkan-бэкенд создаёт device-буферы, кэширует их, синхронизирует dirty-состояния и запускает compute-шейдеры. Для packed-весов добавлены отдельные операции GEMM NN, GEMM NT и embedding gather. Активации остаются FP32, поэтому менять формат каждого промежуточного тензора не требуется.

Интерфейсный слой включает main.cpp, Python-привязки и Ollama-обработчики. Все они формируют LoadOptions и передают его общей функции загрузки. Благодаря этому валидация weight_type может отличаться в деталях сообщений, но фактический путь упаковки один. Агентный слой и тренировочные компоненты используют те же Params, что помогает сохранить единую семантику модели.

Таблица 2 – Основные модули и их ответственность

| Модуль | Ответственность |
|---|---|
| tensor.h / tensor_storage.cpp | логическая форма, packed metadata, lazy decode, encode |
| gguf_model.cpp | метаданные GGUF, импорт, packed attach, перепаковка |
| model.h / model.cpp | Config, Params, forward/backward, pack_weights |
| vk_backend.cpp | VkBuffer, pinned cache, upload, выбор packed kernel |
| shaders/*.comp | декодирование блока и packed GEMM/embedding |
| python_bindings.cpp | Model.load/create, training boundary |
| ollama_api/handler.cpp | JSON weight_type и PARAMETER weight_type |

## 1.4 Форматы моделей и квантования

Формат .xnh является родным для Xenith и предназначен для полного жизненного цикла модели. Он содержит конфигурацию, словарь, BPE-

данные, веса и опциональное состояние оптимизатора. Веса .xnh остаются FP32: обучению нужны master weights, градиенты и согласованное обновление параметров. Mmap позволяет отображать файл без немедленного копирования, но не меняет тип элементов.

GGUF — контейнер экосистемы ggml/llama.cpp. В нём хранятся метаданные архитектуры, словарь и таблица тензоров с типом, размерностями и смещением. Xenith читает версии v1–v3 и пишет v3. В packed-режиме указатель tensor_data_view ссылается прямо на отображённый файл, а Tensor хранит тип источника, целевой тип, физические размеры, смещение и флаги раскладки.

Safetensors использует JSON-заголовок и бинарные данные. Импорт HF LlamaForCausalLM сопровождается чтением config.json и tokenizer.json. При генерации safetensors можно декодировать в FP32, а при явном weight_type упаковать матрицы выбранным GPU-форматом. Это более дорогой путь, чем zero-copy GGUF, но он расширяет совместимость.

Квантование Q8_0 использует блок из 32 значений: fp16-масштаб и 32 знаковых int8, итого 34 байта. Q4_0 хранит масштаб и 16 упакованных байт значений со смещением −8, итого 18 байт. Q4_1 добавляет fp16-смещение и занимает 20 байт. F16 и BF16 занимают по 2 байта, F32 — 4. Для прямой GPU-работы длина физической строки должна быть кратна размеру блока; иначе применяется таблица откатов либо FP32-путь.

Таблица 3 – Форматы прямого GPU-хранения

| Тип | Блок | Байт/блок | Бит/вес |
|---|---|---|---|
| F32 | 1 | 4 | 32.00 |
| F16 | 1 | 2 | 16.00 |
| BF16 | 1 | 2 | 16.00 |
| Q8_0 | 32 | 34 | 8.50 |
| Q4_0 | 32 | 18 | 4.50 |
| Q4_1 | 32 | 20 | 5.00 |

1.5 Бэкенды CPU и Vulkan

CPU-бэкенд остаётся базовым и наиболее универсальным. Он выполняет те же логические операции с FP32-тензорами, поддерживает обучение и не требует графического устройства. Для packed-тензора host-доступ вызывает ensure_host(), которая деквантует данные в логический кэш. Это удобно для отладки, небольших операций и fallback, но полный кэш большой модели съедает выигрыш packed-хранения.

Vulkan-бэкенд управляет буферами и очередью вычислений. В кэше GpuBuf хранятся размер, device-буфер, флаги host_dirty/dev_dirty и признак pinned. Веса после prepare_weight() закрепляются: сборщик кэша не выгружает их по TTL. Активации и scratch могут освобождаться, поэтому immutable model weights и временные данные имеют разный жизненный цикл.

Выбор памяти предпочитает DEVICE_LOCAL, а при отсутствии подходящего типа допускает HOST_VISIBLE. На дискретной GPU это может привести к размещению части буферов в системной памяти, доступной через GTT. Staging и readback отображаются один раз и переиспользуются. Раздельный учёт bytes_allocated_ и device_local_bytes_ помогает отличать общий объём Vulkan-выделений от действительно device-local части.

Для маленьких GEMM сохраняется CPU-путь, поскольку запуск шейдера и синхронизация могут стоить дороже вычисления. Packed-формат сам по себе не гарантирует ускорение: он прежде всего уменьшает трафик и рабочее множество. Производительность новой реализации должна

оцениваться отдельно на фиксированных параметрах ctx, batch, числа потоков и драйвера.

# 2 Специальная часть

## 2.1 Анализ проблемы и постановка задачи

Исходная проблема видна из цепочки загрузки. GGUF-файл может содержать 170 матриц Q8_0, но старый универсальный путь вызывал read_tensor_f32 и создавал FP32-массивы для логической формы. Транспозиция и GQA-развёртка дополнительно требовали временных данных. В результате размер оперативной памяти определялся не компактным файлом, а полным представлением модели во float.

Baseline Xenith подтверждает эту модель поведения: до изменения процесс достиг защитного предела RSS 1.55 ГиБ за 1.72 с и не завершил загрузку. Это не измерение готового packed-режима, а диагностический факт исходной конфигурации. Он показывает, что оптимизировать нужно не только compute-ядро, но и момент аллокации. Если полная FP32-копия возникает до запуска генератора, последующие улучшения шейдеров не устраняют главный узкий участок.

Постановка задачи формулируется как набор инвариантов. Логические размеры Tensor должны оставаться прежними. Исходные байты разрешено читать напрямую из mmap. Физическая ориентация и RoPE-порядок должны быть описаны явно. Целевой тип выбирается из шести поддерживаемых значений. Нормы остаются F32. Любая неподдержанная ситуация должна иметь корректный FP32-fallback, а обучение не должно молча получить квантованные master weights.

Приоритетным сценарием является dense LLaMA/Qwen-подобная модель с отдельными матрицами Q, K, V, O и FFN. GPT-2 с fused QKV и MoE со stacked экспертами используют проверенный FP32-импорт. Это

сознательное сужение packed-пути позволяет избежать большой временной матрицы и снижает число ошибок в первой реализации.

### 2.2 Сравнение Xenith и llama.cpp на уровне исходников

Сравнение выполнялось по Xenith и llama.cpp ревизии 7347430f4466d4f55cfb841974ee64b80fc18d93. В llama.cpp загрузчик llama-model-loader.cpp читает GGUF-метаданные, ведёт weights_map и создаёт тензоры через buft-списки. Функция create_tensor выбирает подходящий ggml_backend_buffer_type_t для входных, выходных и повторяющихся тензоров слоёв. Проверка weight_buft_supported создаёт пробную ggml-операцию и спрашивает устройство, поддерживается ли она с данным буфером.

Xenith использует более прямой механизм. load_tensor_from_gguf проверяет форму и при возможности вызывает attach_packed(), передавая mmap, смещение, исходный QType и целевой storage_type. Выбор supported-формы происходит заранее в guarded-коде загрузчика, а VkBackend::prepare_weight_nolock выделяет буфер и загружает байты. Это меньше абстракций, но требует аккуратного синхронного описания каждой поддерживаемой раскладки.

Модель llama.cpp представлена семейством llama_model_* и графом ggml. llama_context создаёт backend scheduler, output buffer и memory-модуль, резервирует графы prompt processing и token generation. KV-кэш инкапсулирован отдельными memory-реализациями и может использовать различные type_k/type_v. В Xenith генерация реализована собственным циклом и LayerCache; GQA в packed-весах остаётся сжатой, а attention работает с логическими K/V.

Vulkan-путь llama.cpp существенно шире: ggml-vulkan.cpp содержит pipelines для матричного умножения, dequant-mul-mat, get_rows, fused RMSNorm/RoPE, flash attention, MoE и других операций. Есть выбор shared-memory layout, cooperative matrix, integer dot product и специализации под архитектуру GPU. Xenith добавляет три компактных packed-шейдера, которые декодируют значение непосредственно внутри GEMM/embedding, но не претендуют на эквивалентную оптимизационную зрелость.

Поэтому вывод сравнения не должен сводиться к «кто быстрее». Xenith выигрывает интеграцией обучения, Python/Ollama API и собственного агентского слоя в одном проекте. llama.cpp предоставляет более широкую поддержку архитектур, развитый scheduler и большое количество высокопроизводительных GPU-путей. Новый packed-режим Xenith — шаг к рациональной памяти, а не доказательство паритета по скорости.

Таблица 4 – Сравнение архитектурных решений

| Аспект | Xenith | llama.cpp |
|---|---|---|
| Загрузчик | GgufFile + attach_packed | llama_model_loader + buft lists |
| Граф | собственный forward/generate | ggml graph + backend scheduler |
| GPU-веса | 6 прямых типов, 3 packed kernel | много pipelines и архитектур GPU |
| GQA | packed K/V без развёртки | нативные GQA/KV memory paths |
| KV-кэш | LayerCache, FP32 | memory module, type_k/type_v |
| Обучение | FP32 master weights | в основном inference-экосистема |
| API | CLI, Python, Ollama в проекте | C API, CLI, серверы/обёртки |

## 2.3 Исследуемая модель MiniCPM5

Для проверки используется файл MiniCPM5-1B-Claude-Opus-Fable5-V2-Thinking-Q8_0.gguf. Его SHA256 равен fc3ee1eddd305c155f63b6bd7bb189daa4d5f226ca325ab219bd7acd3b00ec77. Размер составляет 1,153,529,184 байт. Контейнер GGUF версии 3 содержит 219 тензоров: 170 имеют тип Q8_0 и 49 — F32. Такая структура

соответствует принятому правилу: большие матрицы квантованы, а нормы и другие чувствительные векторы оставлены в полной точности.

Архитектура содержит 24 слоя, скрытое измерение d=1536, FFN=4608, 16 query-голов и 2 KV-головы. Размер головы hd=128, отношение GQA равно 8. Словарь включает 130560 токенов, заявленный контекст — 131072, база RoPE — 5 000 000. Число параметров оценивается примерно в 1030.57M. Контекст файла не следует путать с контекстом smoke-теста: в baseline использован ctx=128 и n=1.

Размер Q8_0-блока позволяет вычислить компактный объём матрицы: 34 байта на каждые 32 значения. Приблизительный FP32-объём тех же параметров в четыре раза больше по числу элементов, но точное сравнение должно учитывать 49 F32-тензоров, выравнивание и метаданные. В работе не делается вид, что весь файл строго равен сумме весов: GGUF содержит также словарь, таблицу и выравнивание.

GQA особенно важен для packed-режима. K/V имеют 2 головы вместо 16, поэтому их матрицы уже меньше. В старом FP32-импорте GQA могла разворачивать K/V в MHA и создавать дополнительные данные. Новый guarded-путь сохраняет сжатые K/V и permet шейдеру читать физическую матрицу без предварительной репликации. Логическое выполнение attention при этом не меняется.

Таблица 5 – Точная карточка модели

| Параметр | Значение |
|---|---|
| Файл | MiniCPM5-1B-Claude-Opus-Fable5-V2-Thinking-Q8_0.gguf |
| SHA256 | fc3ee1eddd305c155f63b6bd7bb189daa4d5f226ca325ab219bd7acd3b00ec77 |
| Размер | 1,153,529,184 байт |
| GGUF | версия 3 |
| Тензоры | 219: 170 Q8_0 + 49 F32 |
| Слои / d / FFN | 24 / 1536 / 4608 |
| Головы Q / KV / hd | 16 / 2 / 128 |

| Параметр | Значение |
|---|---|
| Словарь / контекст | 130560 / 131072 |
| RoPE base | 5 000 000 |
| Параметры | ≈1030.57M |

2.4 Проектирование хранилища Tensor

Tensor расширяется полями packed_type_, storage_type_, packed_transposed_, packed_phys_rows_, packed_phys_cols_, rope_heads_ и rope_permute_. Владельцем данных выступает shared_ptr<MappedFile> для GGUF либо shared_ptr<const vector<uint8_t>> для результата перепаковки. Логические rows/cols не изменяются, поэтому существующие операции forward/backward продолжают видеть привычную форму.

Функция attach_impl проверяет произведение логических и физических размеров, соответствие ориентации, кратность строки блоку, наличие владельца и границы данных. При transposed требуется phys_rows==cols и phys_cols==rows. После проверки clear_storage() удаляет прежний FP32/mmap-буфер, а Tensor переходит в packed-состояние. Это исключает ситуацию, когда оба представления случайно остаются живыми.

ensure_host() реализует ленивое декодирование. Для каждой физической строки вызывается dequantize_row(). Если данные транспонированы, значения распределяются по logical[in][out]. При rope_permute выполняется обратная перестановка из rotate-half в interleave. Полный host_cache_ создаётся только по явному host-доступу, CPU-fallback или подготовке к редактированию.

encode_packed() имеет быстрый путь: если целевой тип совпал с packed_type_, исходные байты копируются без деквантования. Иначе выполняется построчная перепаковка. Это важно для source-режима, где цель

— не «повторить квантование», а сохранить точные байты автора файла. pack_from_host() используется для .xnh и safetensors, где исходный тензор уже находится во FP32.

Копирование Tensor разделяет packed bytes, но не разделяет host_cache_. Иначе изменение одной логической копии незаметно влияло бы на другую. Такое решение подтверждено тестом copy.ptr()[0] += 1.0f: исходный packed-тензор после собственного decode не должен измениться.

### 2.5 Изменения загрузчика GGUF

Packed-путь включается только при активном Vulkan-бэкенде, opt.packed_weights, отсутствии alloc_grads и dense-архитектуре без MoE/GPT-2. До allocate storage loader вычисляет packed и вызывает alloc_storage(!packed), то есть создаёт только формы без FP32-данных. Это критический момент: если сначала выделить полные массивы, позднейшее attach_packed уже не устранит пик памяти.

Для норм используется load_vec() с allow_packed=false. Для матриц load_mat() передаёт transpose, целевой тип и native-флаг. Q/K дополнительно получают rope_permute и число голов. token_embd и output загружаются без транспозиции. Если guarded attach не удался, loader предупреждает и вызывает обычный FP32-import конкретного тензора, а не всей модели.

После загрузки выполняется обход ck.params.all(). Каждый packed-тензор отправляется в backend()->prepare_weight(), подсчитываются packed_count и packed_bytes, а журнал получает распределение по фактическим target type. Благодаря qtype_fallback_for_row фактический тип может отличаться от запрошенного, поэтому диагностика строится по packed_target_type().

Явный source не требует конверсии: native_storage=true заставляет storage_type совпасть с исходным ti->type. Явный q8_0 для Q8_0-файла также использует быстрый путь. Явный q4_0 приводит к построчной перепаковке в prepare_weight; полный FP32-кэш при этом не нужен, поскольку encode_packed работает с одной строкой за раз.

**Листинг А.1 – Условие packed-загрузки (xenith/core/gguf_model.cpp)**

```
                  "`Xenith convert ... --ctx N`", a.cfg.block_size,
opt.gguf_ctx_cap);
        engine_info(ib);
        a.cfg.block_size = opt.gguf_ctx_cap;
    }

    Config& c = ck.params.cfg = a.cfg;
    // Packed-путь предназначен для инференса dense LLaMA/Qwen на Vulkan.
    // Обучение требует FP32 master weights; fused GPT-2 QKV и stacked MoE
    // пока используют проверенный FP32-импорт.
    const bool packed = backend_is_gpu() && opt.packed_weights && !opt.alloc_grads
&&
                        !c.moe() && c.arch != (int32_t)Arch::Gpt2;
    ck.params.alloc_storage(!packed);

    const int d = c.n_embd, hd = c.head_dim(), qd = c.q_dim(), kvd = c.kv_dim();
    const int H = c.n_head, KV = c.kv_heads();

    auto load_vec = [&](const char* name, Tensor& dst) -> bool {
        // Нормы/смещения остаются FP32: это рекомендация llama.cpp и они малы.
        return load_tensor_from_gguf(g, name, dst, /*transpose=*/false, err,
                                     0, false, 1, false, QType::F32, true);
    };
    auto load_mat = [&](const std::string& name, Tensor& dst, bool transpose) ->
bool {
        return load_tensor_from_gguf(g, name, dst, transpose, err,
                                     0, false, 1, packed, opt.weight_type,
opt.weight_native);
    };
    auto load_rope_mat = [&](const std::string& name, Tensor& dst, int heads) ->
bool {
        return load_tensor_from_gguf(g, name, dst, /*transpose=*/true, err,
                                     0, /*rope_permute=*/true, heads, packed,
                                     opt.weight_type, opt.weight_native);
    };

    if (!load_tensor_from_gguf(g, "token_embd.weight", ck.params.wte, false, err,
                               0, false, 1, packed, opt.weight_type,
opt.weight_native)) return false;
    if (!a.tied) {
```

2.6 Vulkan-шейдеры и pinned-буферы

Общий декодер находится в shaders/common.glsl. Буфер объявлен как uint[], поэтому загрузка байтов не зависит от выравнивания float и работает с F16/BF16/Q8/Q4. Макрос PACKED_BYTE проверяет границу и собирает little-endian значение. packed_f16 преобразует половину в float, включая субнормальные и специальные значения. BF16 восстанавливается сдвигом верхних 16 бит.

Для Q8_0 шейдер читает fp16 scale и знаковый int8. Для Q4_0/Q4_1 он распаковывает два nibble из байта, применяет масштаб и, при необходимости, смещение. Размер строки рассчитывается одинаково на хосте и в шейдере: cols×4, cols×2 либо (cols/32)×34/18/20. packed_meta_valid() проверяет совпадение ожидаемого размера до запуска ядра.

gemm_nn_packed.comp вычисляет C += A·B. Флаг bit0 означает физическую [M][K], bit1 — RoPE-порядок. Для Q/K src_j переводит логический выход в GGUF rotate-half индекс. gemm_nt_packed.comp читает B физически [M][K] и использует четыре аккумулятора для развёртки цикла. embed_gather_packed.comp выбирает строку embedding по token id и декодирует каждый элемент.

Pinned-буфер создаётся prepare_weight_nolock(). Ключ кэша включает адрес Tensor и размер целевых байтов. Если запись уже синхронна, она помечается pinned и переиспользуется. При конверсии upload указывает на converted, затем record_upload_locked копирует данные в device-буфер. После flush host_dirty/dev_dirty сбрасываются, а pinned=true запрещает TTL-выгрузку.

FLAG BF_PACKED необходим, чтобы runtime не пытался повторно интерпретировать packed bytes как FP32-активацию. При host-

редактировании packed_host_modified_ заставляет бэкенд декодировать host-кэш, восстановить физическую раскладку и перекодировать вес. Для инференса это дорогая, но редкая операция; она сохраняет корректность Python numpy-view.

**Листинг А.2 – Декодер packed-типов (xenith/backend/shaders/common.glsl)**

```
// --- Packed/native GGUF weights ----------------------------------------
// QType ids match xenith/core/quant.h. Packed buffers are uint[] so every
// load is byte-exact and does not depend on GPU buffer alignment.
const int PK_F32  = 0;
const int PK_F16  = 1;
const int PK_Q4_0 = 2;
const int PK_Q4_1 = 3;
const int PK_Q8_0 = 8;
const int PK_BF16 = 30;

int packed_row_bytes(int qtype, int cols) {
    if (qtype == PK_F32) return cols * 4;
    if (qtype == PK_F16 || qtype == PK_BF16) return cols * 2;
    if (qtype == PK_Q8_0 && (cols % 32) == 0) return (cols / 32) * 34;
    if (qtype == PK_Q4_0 && (cols % 32) == 0) return (cols / 32) * 18;
    if (qtype == PK_Q4_1 && (cols % 32) == 0) return (cols / 32) * 20;
    return 0;
#define XENITH_DEFINE_PACKED_VALUE(fn, data) \
float fn(int row, int col, int qtype, int phys_cols, int row_bytes) { \
    if (row < 0 || col < 0 || col >= phys_cols || row_bytes <= 0) return 0.0; \
    uint base = uint(row) * uint(row_bytes); \
    if (qtype == PK_F32) return PACKED_F32(data, base + uint(col) * 4u); \
    if (qtype == PK_F16) return packed_f16(PACKED_U16(data, base + uint(col) *
2u)); \
    if (qtype == PK_BF16) \
        return uintBitsToFloat(PACKED_U16(data, base + uint(col) * 2u) << 16); \
    int block = col / 32; \
    int lane = col - block * 32; \
    if (qtype == PK_Q8_0) { \
        uint off = base + uint(block) * 34u; \
        float d = packed_f16(PACKED_U16(data, off)); \
        int q = int(PACKED_BYTE(data, off + 2u + uint(lane))); \
        if (q >= 128) q -= 256; \
        return float(q) * d; \
    } \
    if (qtype == PK_Q4_0 || qtype == PK_Q4_1) { \
        uint off = base + uint(block) * uint(qtype == PK_Q4_0 ? 18u : 20u); \
        float d = packed_f16(PACKED_U16(data, off)); \
        uint qs_off = off + (qtype == PK_Q4_0 ? 2u : 4u) + uint(lane / 2); \
        int q = int(PACKED_BYTE(data, qs_off)); \
        q = (lane < 16) ? (q & 15) : (q >> 4); \
        if (qtype == PK_Q4_0) return float(q - 8) * d; \
        float m = packed_f16(PACKED_U16(data, off + 2u)); \
        return float(q) * d + m; \
    } \
    return 0.0; \
```

## 2.7 Интерфейсы CLI, Python и Ollama

CLI принимает --weight-type source|f32|f16|bf16|q8_0|q4_0|q4_1. Для загрузки значение по умолчанию — source. parse_weight_type_arg() приводит строку к нижнему регистру, распознаёт native как синоним source и запрещает типы без прямого GPU-шейдера. Ошибка в параметре не должна незаметно включить FP32: для new команда завершается, для load выводится предупреждение и используется безопасный source.

Новая команда создания имеет семантику new --weight-type q8_0 model.gguf. Если расширение не .gguf, любой тип кроме f32 отвергается, потому что .xnh остаётся FP32. В GgufSaveOptions устанавливаются quant, emb_type, output_type и norms_f32=true. Таким образом, выбор формата на создании затрагивает и embedding/output, но исключает нормы.

Python предоставляет Model.load(path, weight_type="source") и Model.create(..., weight_type="f32"). В do_load() общий LoadOptions передаётся load_checkpoint(). В do_create() тип, отличный от f32, требует активного Vulkan и вызывает Params::pack_weights(). build_tc() проверяет все тензоры и бросает XenithError при попытке обучения packed-модели.

Ollama /api/create читает JSON weight_type либо строку PARAMETER weight_type в Modelfile. Пустое значение нормализуется в source. Явный тип без Vulkan возвращает HTTP 400. Entry хранит policy до ленивой загрузки, поэтому create и последующий generate используют одну настройку. Это важно для совместимости с клиентами, которые передают Modelfile, а не расширенный JSON.

**Листинг А.3 – Python API и граница обучения**

```
static bool py_weight_type(const std::string& value, bool creation,
                           LoadOptions& out, std::string& err) {
    std::string v = value;
    std::transform(v.begin(), v.end(), v.begin(), [](unsigned char c){ return
(char)std::tolower(c); });
    if (v == "source" || v == "native") {
        if (creation) { err = "при создании нужен конкретный тип весов"; return
false; }
        out.weight_native = true;
        return true;
    }
    QType t;
    if (!qtype_from_name(v, t, err)) return false;
    if (t != QType::F32 && t != QType::F16 && t != QType::BF16 &&
        t != QType::Q8_0 && t != QType::Q4_0 && t != QType::Q4_1) {
        err = "GPU-форматы: f32, f16, bf16, q8_0, q4_0, q4_1";
        return false;
    }
    out.weight_native = false; out.weight_type = t;
    return true;
TrainConfig build_tc(const TrainArgs& a, const PyModel& m) {
    for (const Tensor* t : m.ck.params.all())
        if (t->is_packed())
            throwx("обучение требует FP32 master weights: сохраните модель в .xnh
и загрузите её с weight_type=f32");
    TrainConfig tc = make_train_config(a.steps, a.batch, a.block, a.lr, a.wd,
a.clip,
                                       a.beta1, a.beta2, a.eps, a.warmup,
a.lr_min_frac,
                                       (uint64_t)a.seed, a.log_every,
a.ckpt_every,
                                       a.val_every, a.val_tokens, a.val_windows,
                                       a.optimizer, a.accum, a.label_smoothing,
a.threads);
    if (tc.block > m.ck.params.cfg.block_size) tc.block =
m.ck.params.cfg.block_size;
```

2.8 Создание модели и выбор формата

Создание начинается с проверки конфигурации до построения словаря. Проверяются положительные размеры, делимость эмбеддинга на число голов, согласованность FFN, контекста и RoPE. Затем строится токенайзер, инициализируются FP32-веса, и выполняется сохранение. Для GGUF выбранный вес становится форматом матриц; для .xnh вызывается штатный save_checkpoint(), который не квантует master weights.

Выбор source недопустим при создании, поскольку исходного файла ещё нет. Для new нужен конкретный f32/f16/bf16/q8_0/q4_0/q4_1. F32

допустим и для .xnh, и для .gguf. Явный GGUF-тип уменьшает файл, но для обучения модель следует convert/dequantize в .xnh либо создать отдельную FP32-копию.

При загрузке выбор явного типа — это transient policy. Он не перезаписывает GGUF и не меняет SHA256 исходного файла. Это позволяет одному и тому же файлу запускать с source для максимального сохранения исходных байтов или с q4_0 для эксперимента с памятью. Цена — перепаковка при загрузке и потенциальное отличие численного результата из-за повторного квантования.

**Листинг А.4 – Примеры использования нового API**

```
# CLI: создать GGUF с явным типом
Xenith new model.gguf --corpus corpus.txt --weight-type q8_0

# CLI: сохранить исходный Q8_0 при загрузке
Xenith gen MiniCPM5-Q8_0.gguf --prompt "Привет" --n 32 --vulkan \
    --ctx 128 --weight-type source

# CLI: transient repack
Xenith gen MiniCPM5-Q8_0.gguf --prompt "Привет" --n 32 --vulkan \
    --ctx 128 --weight-type q4_0

# Python
import xenith as xe
m = xe.Model.load("model.gguf", mmap=True, weight_type="q8_0")
print(m.generate_text("Привет", n=32))

# Ollama Modelfile
# FROM /models/MiniCPM5-Q8_0.gguf
# PARAMETER weight_type q8_0
```

2.9 Учёт памяти и бюджет RX 570

Учёт памяти следует разделять на четыре области: резидентные страницы mmap/owned packed bytes, FP32 host cache при fallback, Vulkan device allocations и временные staging/readback. Сумма размеров файлов не равна RSS, поскольку mmap подгружает страницы лениво. С другой стороны, pinned GPU-буфер не означает отсутствия host-источника: исходные байты могут оставаться отображёнными или храниться в owned vector.

Для n весов размер прямого хранения вычисляется по таблице 2. F32 занимает 4n, F16/BF16 — 2n, Q8_0 — 34⌈n/32⌉ при построчной упаковке, Q4_0 — 18⌈n/32⌉, Q4_1 — 20⌈n/32⌉. Практический loader требует точной кратности физической строки; округление здесь показано как верхняя оценка блочного формата, а не обход проверки.

KV-кэш для исследуемой модели при n=1 и ctx=128 оценивается как 2×24×2×128×128×4 = 6 291 456 байт, то есть 6 МиБ. Сюда не входят активации, logits и буферы sampler. Logits одного шага при словаре 130560 занимают 130560×4 = 522 240 байт, около 510 КиБ. Эти величины малы по сравнению с весами, но растут линейно с контекстом и числом последовательностей.

GTT (Graphics Translation Table) — механизм подсистемы GPU, позволяющий использовать часть системной памяти как видимую для устройства. Если DEVICE_LOCAL не хватает или драйвер выбирает host-visible heap, драйвер может мигрировать страницы между VRAM и RAM. Поэтому положительный прирост GTT при почти неизменном RSS не означает утечку: это перемещение рабочих данных, а итоговая стоимость включает и PCIe-трафик, и RAM.

Для RX 570 с 4 ГиБ нельзя планировать размещение всех 1.15 ГиБ Q8_0 плюс полной FP32-копии, активаций и driver overhead без проверки. Packed-режим снижает нижнюю границу, но не гарантирует, что весь working set поместится в VRAM. Критерием успеха должна быть завершённая загрузка и генерация при заданном ctx, а не только уменьшение одного счётчика.

Таблица 6 – Формулы и ориентиры памяти

| Компонент | Формула / значение |
|---|---|
| F32 / F16 / BF16 | 4n / 2n / 2n байт |
| Q8_0 / Q4_0 / Q4_1 | 34 / 18 / 20 байт на 32 веса |
| KV, ctx128, n1 | 2·24·2·128·128·4 = 6 МиБ |
| Logits, V=130560 | 522 240 байт ≈ 510 КиБ |
| GGUF-файл | 1 153 529 184 байта |
| RSS | resident host + mapped pages + allocator + runtime |
| VRAM | device-local allocations; зависит от heaps драйвера |
| GTT | системные страницы, выбранные драйвером для GPU |

## 2.10 CPU fallback и граница обучения

CPU fallback включается для небольших GEMM, неподдержанного packed metadata и ситуаций, когда целевой тип не входит в шесть прямых. В gemm_nn перед FP32-путём проверяется packed_nn_usable(). Если проверка не прошла, backend вызывает host_written_nolock(C), а host-операция обращается к B через ptr(), что при необходимости декодирует packed-тензор. Для полностью неподдержанного типа loader заранее оставляет FP32.

Этот путь обеспечивает корректность, но не следует считать его эквивалентным по памяти. После ensure_host() память модели может приблизиться к FP32-размеру. Поэтому диагностика должна различать успешный packed-kernel и lazy decode fallback. Иначе визуально работающая генерация может скрывать полную деквантацию.

Обучение намеренно исключено из packed-режима. CLI train/gen_only=false выставляет alloc_grads, и условие packed становится ложным. .xnh сохраняет FP32. Python build_tc() отклоняет любой packed Tensor. Даже если вычисление forward идёт на Vulkan, master weights и градиенты остаются FP32, поскольку квантование градиентов и обновления требует отдельного алгоритма и иной численной политики.

Граница не является временным недостатком интерфейса; она отражает различие задач. Инференс допускает approximation весов, поскольку forward только читает параметры. Обучение изменяет их и должно сохранять малые обновления, состояние Adam и воспроизводимость. Смешивание этих режимов привело бы к неконтролируемой потере точности.

2.11 Базовая экспериментальная проверка

Ниже приведены только исходные baseline-измерения, полученные до нового packed-пути. Они не характеризуют скорость или память post-implementation. Конфигурация хоста: AMD Radeon RX 570 4 ГиБ, RADV Mesa 26.0.8, 16 ГиБ RAM. Тест является smoke-проверкой с ctx=128 и n=1, а не качественным бенчмарком качества модели, долгого контекста или батча.

Xenith до изменения достиг RSS guard 1.55 ГиБ за 1.72 с до завершения загрузки. Это незавершённый запуск, поэтому сравнивать его время с успешным llama.cpp как load time некорректно. llama.cpp CPU с ngl=0 показал peak RSS 1 263 244 КиБ и 2.021 с. Vulkan ngl=8: RSS 1 260 776 КиБ, прирост VRAM +432.9 МиБ, GTT +205.6 МиБ, время 2.708 с. Vulkan ngl=24: RSS 1 260 708 КиБ, VRAM +23.8 МиБ, GTT +1070.9 МиБ, время 2.641 с.

Рост GTT при увеличении ngl объясняется политикой размещения: больше слоёв назначается GPU, но при ограниченном device-local бюджете драйвер использует системные страницы. Почти неизменный RSS в таблице — это peak host-процесса в рамках методики, а не доказательство отсутствия системной памяти. GTT-прирост учитывается отдельно.

Из baseline нельзя сделать вывод о post-change выигрыше. Для этого нужен отдельный прогон с source и каждым явным типом, одинаковым ctx/n, несколькими повторами, /usr/bin/time, счётчиками VRAM/GTT и проверкой

факта packed kernel. Также нужны quality checks: совпадение logits/токенов в допустимых пределах, отсутствие NaN и завершение генерации.

Таблица 7 – Исходные baseline-измерения (ctx=128, n=1; не post-change)

| Конфигурация | RSS peak | VRAM / GTT | Время |
|---|---|---|---|
| Xenith до изменения | guard 1.55 ГиБ | загрузка не завершена | 1.72 с |
| llama.cpp CPU, ngl=0 | 1 263 244 КиБ | — | 2.021 с |
| llama.cpp Vulkan, ngl=8 | 1 260 776 КиБ | +432.9 / +205.6 МиБ | 2.708 с |
| llama.cpp Vulkan, ngl=24 | 1 260 708 КиБ | +23.8 / +1070.9 МиБ | 2.641 с |

## 2.12 Тестирование и оценка результатов

Автоматический tools/tensor_storage_test.cpp проверяет все шесть прямых типов на матрице 64×96. Тест упаковывает её с transpose=true, убеждается, что host-буфер не выделен, вызывает ensure_host() и сравнивает результат с независимым построчным эталоном quantize_buffer/dequantize_row. Затем encode_packed() того же типа обязан вернуть побайтно исходные данные. Отдельно проверяется q8_0→q4_0 и независимость decoded cache при копировании Tensor.

Интеграционные проверки должны охватывать source Q8_0, каждый явный тип, транспонированные Q/K с RoPE, embedding gather, неподдержанную форму, обрезанный буфер, некратную строку и загрузку .xnh с weight_type=q8_0. Для Vulkan необходимо проверить, что выбран packed kernel, а не CPU fallback. Журнал GPU packed и счётчик форматов дают первую линию диагностики.

Проверка документации включает точную карточку модели, одинаковые значения API, отсутствие оптических скриншотов и текста, 9 слайдов, 3 схемы, сохранность фоновых JPEG и две секции DOCX. PDF проверяется pdfinfo/pdftotext, а эскизы страниц — pdftoppm. Это не заменяет

вычислительные тесты, но защищает комплект от технических ошибок сборки.

Полученный результат — реализованная архитектура compact storage и воспроизводимый документальный комплекс. Эффект на скорость и итоговый RSS после изменений остаётся предметом отдельного измерения. Честное разделение baseline и будущих замеров является важной частью инженерной работы: оно не позволяет красивой схеме подменить экспериментальные данные.

# ЗАКЛЮЧЕНИЕ

В курсовой работе исследован путь GGUF-модели через загрузчик, Tensor, Vulkan-бэкенд и внешние интерфейсы Xenith. Спроектировано packed-хранилище, которое сохраняет исходные байты, физическую раскладку [out][in], логическую форму [in][out] и RoPE-перестановку. Введены шесть прямых GPU-форматов: f32, f16, bf16, q8_0, q4_0 и q4_1.

Добавлены lazy host decode, byte-preserving fast path, перепаковка по строкам, pinned VkBuffer и compute-шейдеры GEMM NN, GEMM NT и embedding gather. Нормы остаются F32. Сложные fused/stacked-архитектуры и неподдержанные формы используют FP32-fallback. Параметр weight_type унифицирован в CLI, Python и Ollama API; команда new позволяет создавать GGUF выбранного типа.

Сохранена граница обучения: packed-режим предназначен для инференса, .xnh и master weights остаются FP32. Точная карточка MiniCPM5 и исходные baseline-замеры оформлены отдельно. Пост-реализационные показатели памяти и скорости не заявлены, поскольку для них требуется отдельный воспроизводимый эксперимент.

Цель работы достигнута в части проектирования и реализации compact storage. Дальнейшее развитие включает расширение прямых типов, оптимизацию matmul, flash attention, количественные KV-типы и автоматический профиль размещения под бюджет VRAM. Отдельным приоритетом остается набор quality/performance тестов на нескольких GPU и длинном контексте.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Xenith. README проекта : официальный репозиторий. – URL: https://git.bipfr.ru/BIPfR/Xenith (дата обращения: 24.11.2026). – Текст : электронный.
2. Xenith. tensor.h : исходный код представления тензора. – URL: https://git.bipfr.ru/BIPfR/Xenith/raw/branch/master/xenith/core/tensor.h. – Текст : электронный.
3. Xenith. tensor_storage.cpp : упаковка, декодирование и перепаковка тензоров. – URL: https://git.bipfr.ru/BIPfR/Xenith/raw/branch/master/xenith/core/tensor_storage.cpp. – Текст : электронный.
4. Xenith. gguf_model.cpp : импорт GGUF и packed metadata. – URL: https://git.bipfr.ru/BIPfR/Xenith/raw/branch/master/xenith/core/gguf_model.cpp. – Текст : электронный.
5. Xenith. vk_backend.cpp : Vulkan-буферы и packed kernels. – URL: https://git.bipfr.ru/BIPfR/Xenith/raw/branch/master/xenith/backend/vk_backend.cpp. – Текст : электронный.
6. Xenith. common.glsl, gemm_nn_packed.comp, gemm_nt_packed.comp, embed_gather_packed.comp : compute-шейдеры. – URL: https://git.bipfr.ru/BIPfR/Xenith/tree/branch/master/xenith/backend/shaders. – Текст : электронный.
7. Xenith. main.cpp, help.h : CLI и параметр --weight-type. – URL: https://git.bipfr.ru/BIPfR/Xenith/tree/branch/master. – Текст : электронный.
8. Xenith. python_bindings.cpp : Model.load/create и граница обучения. – URL: https://git.bipfr.ru/BIPfR/Xenith/raw/branch/master/xenith/python/python_bindings.cpp. – Текст : электронный.
9. Xenith. ollama_api/handler.cpp : /api/create и PARAMETER weight_type. – URL: https://git.bipfr.ru/BIPfR/Xenith/raw/branch/master/xenith/ollama_api/handler.cpp. – Текст : электронный.
10. Xenith. model.h, checkpoint.h, quant.h : конфигурация, LoadOptions и типы ggml. – URL: https://git.bipfr.ru/BIPfR/Xenith/tree/branch/master/xenith/core. – Текст : электронный.
11. llama.cpp. Исходный код ревизии 7347430f4466d4f55cfb841974ee64b80fc18d93. – URL: https://github.com/ggml-org/llama.cpp/tree/7347430f4466d4f55cfb841974ee64b80fc18d93 (дата обращения: 24.11.2026). – Текст : электронный.
12. llama.cpp. llama-model-loader.cpp, ревизия 7347430f4466d4f55cfb841974ee64b80fc18d93. – URL: https://github.com/ggml-org/llama.cpp/blob/7347430f4466d4f55cfb841974ee64b80fc18d93/src/llama-model-loader.cpp. – Текст : электронный.
13. llama.cpp. llama-model.cpp и llama-context.cpp, ревизия 7347430f4466d4f55cfb841974ee64b80fc18d93. – URL: https://github.com/ggml-org/llama.cpp/tree/7347430f4466d4f55cfb841974ee64b80fc18d93/src. – Текст : электронный.

14. llama.cpp. ggml-vulkan.cpp, ревизия 7347430f4466d4f55cfb841974ee64b80fc18d93. – URL: https://github.com/ggml-org/llama.cpp/blob/7347430f4466d4f55cfb841974ee64b80fc18d93/ggml/src/ggml-vulkan/ggml-vulkan.cpp. – Текст : электронный.
15. ggml-org. GGUF specification : формат контейнера и типы тензоров. – URL: https://github.com/ggml-org/gguf/blob/master/docs/gguf.md. – Текст : электронный.
16. Vaswani A. Attention Is All You Need : статья. – URL: https://arxiv.org/abs/1706.03762. – Текст : электронный.
17. Su J. RoFormer: Enhanced Transformer with Rotary Position Embedding : статья. – URL: https://arxiv.org/abs/2104.09864. – Текст : электронный.
18. Dettmers T. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale : статья. – URL: https://arxiv.org/abs/2208.07339. – Текст : электронный.
19. Khronos Group. Vulkan Guide : memory types, buffers and synchronization. – URL: https://github.com/KhronosGroup/Vulkan-Guide. – Текст : электронный.
20. Ollama. API documentation : /api/create и Modelfile. – URL: https://github.com/ollama/ollama/blob/main/docs/api.md. – Текст : электронный.
21. pybind11. Documentation : Python/C++ bindings. – URL: https://pybind11.readthedocs.io/. – Текст : электронный.
22. Mesa. RADV documentation : Vulkan driver for AMD GPUs. – URL: https://docs.mesa3d.org/drivers/radv.html. – Текст : электронный.
23. ГОСТ 7.32–2017. Отчёт о научно-исследовательской работе. Структура и правила оформления. – Москва : Стандартинформ, 2018. – 20 с. – Текст : непосредственный.
24. ГОСТ 2.104–2006. Единая система конструкторской документации. Основные надписи. – Москва : Стандартинформ, 2007. – 18 с. – Текст : непосредственный.
25. ГОСТ 2.105–95. Единая система конструкторской документации. Общие требования к текстовым документам. – Москва : Стандартинформ, 2006. – 31 с. – Текст : непосредственный.

# ПРИЛОЖЕНИЕ А

Конспективные фрагменты реальных исходников Xenith

## A.1. Packed metadata Tensor (xenith/core/tensor.h)

```
    // --- упакованные веса (GGUF/явно выбранный GPU-формат) ------------------
    // Логическая форма остаётся [rows][cols] Xenith. Физические данные могут
    // лежать в исходной раскладке GGUF [out][in]; shader сам выполняет чтение
    // с транспозицией и перестановку RoPE, поэтому распаковывать всю модель
    // в FP32 не требуется. packed_type_ — тип источника, storage_type_ —
    // затребованный формат в памяти Vulkan.
    QType   packed_type_   = QType::F32;
    QType   storage_type_  = QType::F32;
    bool    packed_transposed_ = false;
    int     packed_phys_rows_ = 0, packed_phys_cols_ = 0;
    int     rope_heads_ = 0;
    bool    rope_permute_ = false;
    MappedFilePtr packed_map_;
    std::shared_ptr<const std::vector<uint8_t>> packed_owned_;
    size_t  packed_off_ = 0, packed_bytes_ = 0;
    mutable std::shared_ptr<std::vector<float>> host_cache_;
    QType packed_type() const { return packed_type_; }
    QType storage_type() const { return storage_type_; }
    QType packed_target_type() const;
    bool packed_transposed() const { return packed_transposed_; }
    int  physical_rows() const { return packed_phys_rows_; }
    int  physical_cols() const { return packed_phys_cols_; }
    int  rope_heads() const { return rope_heads_; }
    bool rope_permute() const { return rope_permute_; }
    const uint8_t* packed_ptr() const;
    size_t packed_source_bytes() const { return packed_bytes_; }
    size_t packed_target_bytes() const;
    size_t storage_bytes() const { return is_packed() ? packed_target_bytes() :
n() * sizeof(float); }

    // Декодировать в логический FP32 [rows][cols] лениво. Возвращает false при
    // повреждённых данных/неподдерживаемом типе и заполняет err.
    bool ensure_host(std::string* err = nullptr) const;
    // Закодировать в выбранный формат. Если тип источника совпал, out содержит
    // точные исходные байты; иначе выполняется поквантовая перепаковка строк.
    bool encode_packed(QType target, std::vector<uint8_t>& out, std::string& err)
const;
    // Упаковать уже находящийся в FP32 тензор и освободить d. transpose=true
    // задаёт физическую раскладку [out][in] для матриц gemm_nn.
    bool pack_from_host(QType target, bool transpose, std::string& err);
```

### А.2. Ленивое декодирование (xenith/core/tensor_storage.cpp)

```
    if (!is_packed()) return storage_type_;
    QType effective = storage_type_;
    if (!qtype_fallback_for_row(storage_type_, packed_phys_cols_, effective))
        return storage_type_;
    return effective;
}

size_t Tensor::packed_target_bytes() const {
    if (!is_packed()) return n() * sizeof(float);
    size_t row_bytes = 0;
    std::string err;
    if (!row_bytes_for(packed_target_type(), packed_phys_cols_, row_bytes, err))
return 0;
    return row_bytes * (size_t)packed_phys_rows_;
}

bool Tensor::ensure_host(std::string* err) const {
    if (!is_packed()) {
        if (has_host() || n() == 0) return true;
        if (err) *err = "FP32-буфер тензора не выделен";
        return false;
    }
    if (host_cache_) return true;
    std::string local;
    std::string& e = err ? *err : local;
    const uint8_t* src = packed_ptr();
    if (!src) { e = "упакованный тензор не имеет данных"; return false; }
    size_t src_row = 0;
    if (!row_bytes_for(packed_type_, packed_phys_cols_, src_row, e)) return false;

    auto decoded = std::make_shared<std::vector<float>>(n());
    std::vector<float> row((size_t)packed_phys_cols_);
    const int hd = rope_heads_ > 0 ? cols / rope_heads_ : 0;
    const int half = hd / 2;
    for (int o = 0; o < packed_phys_rows_; ++o) {
        dequantize_row(packed_type_, src + (size_t)o * src_row, row.data(),
packed_phys_cols_);
        if (!packed_transposed_) {
            std::memcpy(decoded->data() + (size_t)o * (size_t)cols, row.data(),
                        (size_t)cols * sizeof(float));
            continue;
        }
        int dst_o = o;
        if (rope_permute_ && hd > 0 && (hd & 1) == 0) {
            const int head = o / hd, lane = o % hd;
            dst_o = lane < half ? 2 * lane : 2 * (lane - half) + 1;
            dst_o += head * hd;
        }
        for (int i = 0; i < packed_phys_cols_; ++i)
            (*decoded)[(size_t)i * (size_t)cols + (size_t)dst_o] = row[(size_t)i];
    }
    host_cache_ = std::move(decoded);
    return true;
}
```

## A.3. Guarded attach из GGUF (xenith/core/gguf_model.cpp)

```
// Загрузить один тензор из GGUF в Tensor (dims/тип проверяются вызывающим).
bool load_tensor_from_gguf(const GgufFile& g, const std::string& name,
                           Tensor& dst, bool transpose, std::string& err,
                           int expand_kv = 0, bool rope_permute = false, int
n_head = 1,
                           bool allow_packed = false, QType storage_type =
QType::F32,
                           bool native_storage = true) {
    const GgufTensorInfo* ti = g.find_tensor(name);
    if (!ti) { err = "в GGUF нет тензора '" + name + "'"; return false; }
    if (!ti->supported()) { err = "тензор '" + name + "': неподдерживаемый тип
ggml id=" +
                                  std::to_string(ti->raw_type); return false; }

    // Нулевое копирование: оставляем исходные байты в mmap GGUF. Транспозиция,
    // GQA-развёртка и RoPE-перестановка остаются метаданными; GPU-shader
    // применяет их при чтении. Сложные fused/stacked тензоры ниже идут через
FP32.
    if (allow_packed && expand_kv == 0 && ti->dims.size() <= 2) {
        int r = dst.R(), c = dst.C(), pr = 0, pc = 0;
        bool layout_ok = false;
        if (transpose && ti->dims.size() == 2) {
            pr = (int)ti->dims[1]; pc = (int)ti->dims[0];
            layout_ok = pr == c && pc == r;
        } else if (!transpose && ti->dims.size() == 2) {
            pr = (int)ti->dims[1]; pc = (int)ti->dims[0];
            layout_ok = pr == r && pc == c;
        } else if (!transpose && ti->dims.size() == 1) {
            pr = 1; pc = (int)ti->dims[0];
            layout_ok = r == 1 && c == pc;
        }
        size_t bytes = 0;
        std::string view_err;
        if (layout_ok) {
            const uint8_t* raw = nullptr;
            if (g.tensor_data_view(*ti, raw, bytes, view_err) &&
                dst.attach_packed(g.shared_map(), ti->data_offset, bytes, ti-
>type,
                                  r, c, pr, pc, transpose, rope_permute ? n_head :
0,
                                  rope_permute, native_storage ? ti->type :
storage_type, err)) {
                return true;
            }
            if (!view_err.empty()) err = view_err;
        }
        if (!layout_ok) err = "неподдерживаемая форма для packed-загрузки";
        engine_warn("тензор '" + name + "': packed-режим недоступен (" + err +
                    ") — деквантую в FP32");
    }
    if (!dst.has_host() || dst.is_packed()) dst.resize(dst.R(), dst.C());
    if (!transpose && expand_kv == 0) {
        // без транспозиции — декодируем ПРЯМО в тензор (без копии размером с
        // тензор: для token_embd больших моделей это сотни МБ пика)
        if ((int64_t)dst.n() != ti->n_elems()) {
            err = "тензор '" + name + "': размер " + std::to_string(ti->n_elems())
+
                  ", ожидалось " + std::to_string(dst.n());
```

```
            return false;
        }
        if (!g.read_tensor_f32_into(*ti, dst.ptr(), dst.n(), err)) {
            err = "тензор '" + name + "': " + err;
            return false;
        }
        return true;
    }
    std::vector<float> f32;
    if (!g.read_tensor_f32(*ti, f32, err)) { err = "тензор '" + name + "': " +
err; return false; }
    if (!import_weights_flat_impl(f32.data(), (int64_t)f32.size(), dst,
```

## А.4. Подготовка pinned Vulkan-буфера (vk_backend.cpp)

```
    const BufKey key = buf_key(sp);
    auto it = cache_.find(key);
    if (it == cache_.end()) {
        if (cache_.size() >= kMaxCacheEntries) sweep_locked(true);
        auto nb = std::make_unique<GpuBuf>();
        if (!alloc_device_buf(*nb, sp.bytes, err)) return nullptr;
        it = cache_.emplace(key, std::move(nb)).first;
    }
    GpuBuf* b = it->second.get();
    b->epoch = epoch_;
    return b;
}

bool VkBackend::prepare_weight_nolock(const Tensor& t, std::string& err) {
    if (!inited_ || dev_ == VK_NULL_HANDLE) {
        err = "Vulkan не инициализирован";
        return false;
    }
    if (degraded_) {
        err = "Vulkan недоступен после ошибки устройства";
                        const int head = o / hd, lane = o % hd;
                        dst_o = lane < half ? 2 * lane : 2 * (lane - half) + 1;
                        dst_o += head * hd;
                    }
                    for (int i = 0; i < t.physical_cols(); ++i)
                        physical(o, i) = host[(size_t)i * (size_t)t.C() +
(size_t)dst_o];
                }
            }
            if (!physical.encode_packed(target, converted, err)) {
                err = "не удалось перепаковать вес из FP32: " + err;
                return false;
            }
            upload = converted.data();
        } else if (target == t.packed_type()) {
            if (t.packed_source_bytes() < bytes) {
                err = "не удалось подготовить вес: исходные упакованные данные
обрезаны";
                return false;
            }
            upload = source;
        } else {
            if (!t.encode_packed(target, converted, err)) {
                err = "не удалось перепаковать вес: " + err;
                return false;
            }
            upload = converted.data();
        }
        if (!upload || (upload == converted.data() && converted.size() != bytes))
{
            err = "не удалось подготовить вес: неверный размер перепакованных
данных";
            return false;
        }
    } else {
        const float* host = t.host_ptr();
        if (!host) {
            err = "не удалось подготовить вес: нет FP32-данных";
```

```
            return false;
        }
        upload = reinterpret_cast<const uint8_t*>(host);
    }

    GpuBuf* b = resolve(BufSpec{upload, bytes, BF_PACKED, nullptr, &t}, err);
    if (!b) {
        err = "не удалось выделить Vulkan-буфер для веса: " + err;
        return false;
    }
    std::string upload_err;
    if (!begin_batch_locked(upload_err) ||
```

### A.5. Packed GEMM NN (gemm_nn_packed.comp)

```
XENITH_DEFINE_PACKED_VALUE(gemm_nn_packed_value, B)

void main() {
    uint gid = gl_GlobalInvocationID.x;
    int N = pk.P0.x, K = pk.P0.y, M = pk.P0.z;
    int qtype = pk.P0.w;
    int flags = pk.P1.x, heads = pk.P1.y;
    int phys_cols = pk.P1.z, row_bytes = pk.P1.w;
    if (gid >= uint(N) * uint(M)) return;

    int i = int(gid) / M;
    int j = int(gid) % M;
    int src_j = j;
    if ((flags & 2) != 0 && heads > 0 && (M % heads) == 0) {
        int hd = M / heads;
        if ((hd & 1) == 0) {
            int head = j / hd;
            int lane = j - head * hd;
            src_j = head * hd + lane / 2 + (lane & 1) * (hd / 2);
        }
    }

    float acc = C[IDX(i, j, M)];
    for (int kk = 0; kk < K; ++kk) {
        int brow = (flags & 1) != 0 ? src_j : kk;
        int bcol = (flags & 1) != 0 ? kk : j;
        acc += A[IDX(i, kk, K)] *
               gemm_nn_packed_value(brow, bcol, qtype, phys_cols, row_bytes);
    }
    C[IDX(i, j, M)] = acc;
}
```

### A.6. Тест шести форматов (tools/tensor_storage_test.cpp)

```
    Tensor packed = host;
    std::string err;
    if (!packed.pack_from_host(type, true, err)) {
        std::fprintf(stderr, "pack %s: %s\n", qtype_name(type), err.c_str());
        return false;
    }
    if (!packed.is_packed() || packed.has_host() || packed.n() != (size_t)R * C)
return false;
    if (!packed.ensure_host(&err)) {
        std::fprintf(stderr, "decode %s: %s\n", qtype_name(type), err.c_str());
        return false;
    }

    std::vector<float> expected((size_t)R * C), row((size_t)R);
    for (int o = 0; o < C; ++o) {
        for (int i = 0; i < R; ++i) row[(size_t)i] = host.at(i * C + o);
        // Сравниваем с независимым эталоном: тот же row, отдельный буфер.
        std::vector<uint8_t> reference;
        if (!quantize_buffer(type, row.data(), R, reference, err)) return false;
        dequantize_row(type, reference.data(), row.data(), R);
        for (int i = 0; i < R; ++i) expected[(size_t)i * C + (size_t)o] =
row[(size_t)i];
    }
    for (size_t i = 0; i < expected.size(); ++i)
        if (!close(packed.at(i), expected[i])) {
            std::fprintf(stderr, "parity %s at %zu: %.9g != %.9g\n",
                         qtype_name(type), i, packed.at(i), expected[i]);
            return false;
        }

    std::vector<uint8_t> same;
    if (!packed.encode_packed(type, same, err)) return false;
    if (same.size() != packed.packed_target_bytes() ||
        std::memcmp(same.data(), packed.packed_ptr(), same.size()) != 0) {
        std::fprintf(stderr, "native byte preservation failed for %s\n",
qtype_name(type));
        return false;
    }

    Tensor copy = packed;
    copy.ptr()[0] += 1.0f;
    if (!close(packed.ptr()[0], copy.ptr()[0] - 1.0f)) {
        std::fprintf(stderr, "decoded cache unexpectedly shared\n");
        return false;
    }
    std::printf("OK %-6s logical=%zu packed=%zu\n", qtype_name(type),
                packed.n(), packed.storage_bytes());
```

# ПРИЛОЖЕНИЕ Б

Схемы загрузки, Vulkan-памяти и генерации

## Б.1 Загрузка и выбор формата весов

Схема построена по реальным вызовам и переходам Xenith. PNG используется для совместимости с LibreOffice, SVG сохраняется отдельно и встраивается в презентацию как native SVG с PNG-fallback.

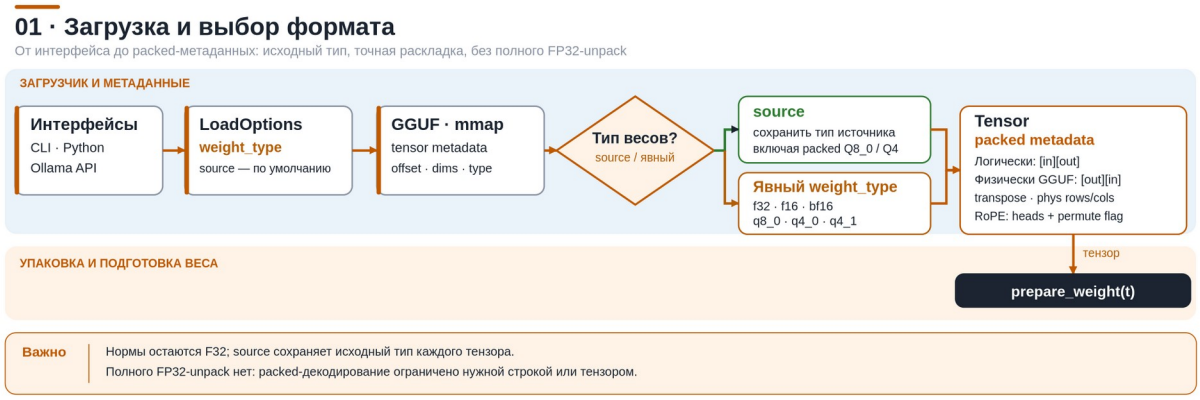



Рисунок 1 – Загрузка и выбор формата весов

Интерфейсы формируют LoadOptions. source сохраняет тип каждого GGUF-тензора; явный тип вызывает поквантовую перепаковку. Логическая и физическая формы, RoPE-флаг и целевой тип фиксируются в Tensor до подготовки Vulkan-буфера.

Б.2 Память и выполнение Vulkan

Схема построена по реальным вызовам и переходам Xenith. PNG используется для совместимости с LibreOffice, SVG сохраняется отдельно и встраивается в презентацию как native SVG с PNG-fallback.

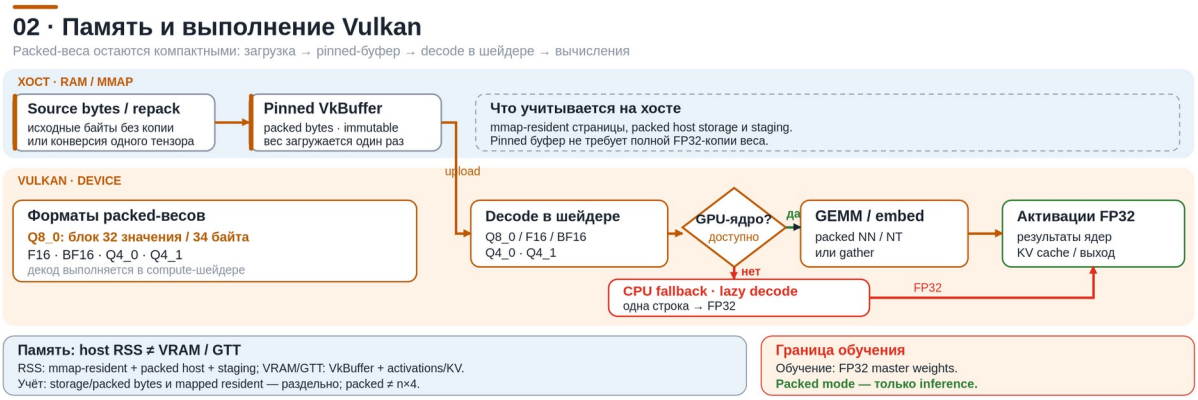



Рисунок 2 – Память и выполнение Vulkan

Pinned-буфер загружается один раз и не выгружается TTL-сборщиком. Шейдер декодирует блок при чтении. При неподдержанном GPU-ядре host-путь может лениво создать FP32-кэш, поэтому packed и fallback следует диагностировать отдельно.

Б.3 Создание, генерация и API

Схема построена по реальным вызовам и переходам Xenith. PNG используется для совместимости с LibreOffice, SVG сохраняется отдельно и встраивается в презентацию как native SVG с PNG-fallback.

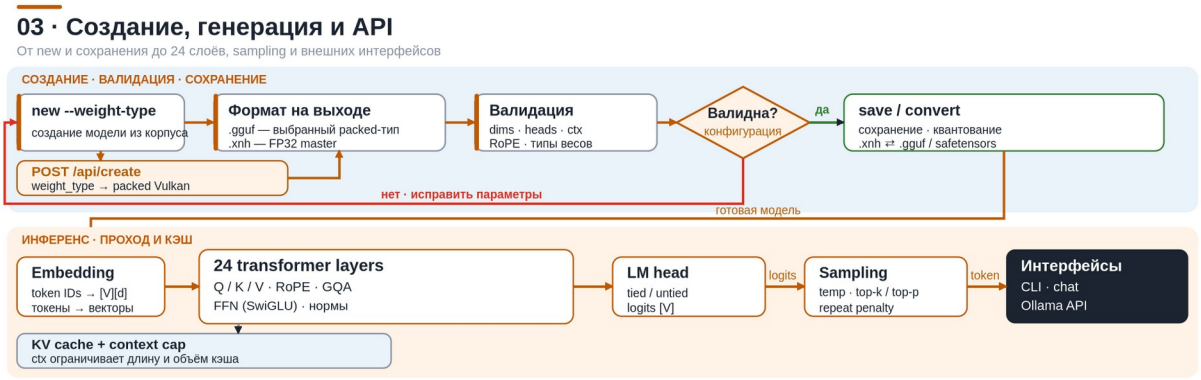



Рисунок 3 – Создание, генерация и API

new выбирает формат файла, а load выбирает формат в памяти. Генерация проходит embedding, 24 трансформерных слоя, LM head и sampling. KV-кэш ограничен контекстом; .xnh и обучение остаются FP32.