**1. Модуль субтитров**Разработанный модуль субтитров представляет собой центральный компонент системы, обеспечивающий полный цикл работы с текстовыми дорожками в видеоматериалах. В его основе лежит многоуровневый конвейер, включающий:

* **Загрузку и парсинг**. Поддерживаются три наиболее распространенных формата — SRT, ASS и VTT. Парсер автоматически определяет структуру файла, корректно обрабатывает нестандартные варианты разметки и учитывает возможные ошибки в кодировке.
* **Предварительную обработку**. На этой стадии выполняется удаление или нормализация управляющих тегов, очистка лишних пробелов и специфических символов. Благодаря предварительной фильтрации повышается надёжность последующих шагов.
* **Стилизацию**. Пользователь может тонко настраивать визуальное оформление текста: задавать цветовую палитру (включая градиенты и полупрозрачные оттенки), выбирать шрифты, устанавливать размер, межстрочный интервал, выравнивание (левое, правое, центрированное, по ширине), обводку, тень и другие декоративные эффекты, которые описываются в JSON-конфиге.
* **Встраивание (прожиг)**. Модуль интегрируется с ffmpeg и библиотекой libass, что позволяет «жестко» закрепить титры в видеопотоке. В результате получаются видео с постоянными субтитрами, которые гарантированно будут корректно отображаться на любых устройствах и в любых плеерах.

**2. Ключевые возможности  
2.1. Конвертация форматов**Модуль обеспечивает двунаправленное преобразование между SRT, ASS и VTT без потери семантики текста и временных меток. При конвертации может производиться автоматическая адаптация стилей: например, из простого SRT-цвета генерируются ASS-теги с заданным шрифтом, обводкой и позицией.

**2.2. Прожиг субтитров**Использование связки ffmpeg + libass даёт возможность вшивать субтитры непосредственно в поток видео с битрейт-контролем и поддержкой аппаратного ускорения кодирования (NVENC, QuickSync, VA-API). Модуль умеет обрабатывать видео любых разрешений и контейнеров, гарантируя синхронность звука и титров.

**2.3. Гибкая настройка стиля**Все параметры оформления субтитров задаются через настраиваемый JSON-файл, где можно определить несколько стилей, назначить их по диапазону временных меток или по типу текста (диалоги, вставки, примечания). Поддерживается наследование стилей и их динамическая смена во время проигрывания.

**2.4. Параллельная обработка текста**Удаление тегов, разбиение строк по длине, анализ и исправление кодировки, подсчёт статистических метрик и т.д. реализованы в виде OpenCL-ядер, что позволяет задействовать вычислительные блоки GPU. Это даёт выигрыш по сравнению с однопоточным CPU-вариантом и обеспечивает линейное масштабирование при добавлении новых устройств.

**3. Структура проекта  
 3.1. Последовательная часть**

* **Сборка и запуск**
  + Перейдите в корень репозитория.
  + Запустите скрипт:
    - Linux/macOS: ./run.sh
    - Windows: run.bat
  + После успешной сборки исполняемые файлы появятся в app/.
* **Основной функционал**
  + Демо-приложение VTL\_sub\_style\_demo демонстрирует весь цикл: от чтения файла до получения видео с прожженными субтитрами (файлы сохраняются в VTL/test/data/).
* **Рефакторинг и доработки**
  + Устранены проблемы с распознаванием временных меток в нестандартном формате.
  + Реализованы и унифицированы функции модуля парсинга.
  + Внесены правки для корректной сборки проекта на macOS.

**3.2. Модуль OpenCL**

* **Обзор технологии**OpenCL — кроссплатформенный стандарт, позволяющий запускать параллельные вычисления на GPU и CPU. В проекте он применяется для ускоренного выполнения ряда однотипных операций над большими массивами текстовых данных.

**3.2.1. Архитектура OpenCL в проекте**

- Все OpenCL-функции вынесены в отдельную папку VTL/media\_container/sub/opencl/

- Для каждой задачи реализовано:

- OpenCL-ядро (kernel) — мини-программа на С-подобном языке, которая выполняется параллельно на всех доступных вычислительных устройствах (обычно на GPU).

- C-обёртка — функция на C, которая подготавливает данные, запускает ядро и собирает результат.

* **Примеры реализаций**
  + VTL\_sub\_OpenclStripTags — удаление тегов и HTML-спецсимволов.
  + VTL\_sub\_OpenclSplitLongLines — перенос по словам для контроля длины строки.
  + VTL\_sub\_OpenclDetectEncoding — распознавание и конвертация кодировки.
  + VTL\_sub\_OpenclFormatNumbers — форматирование числовых значений с разделителями.
  + VTL\_sub\_OpenclApplyAssStyle — массовое применение/удаление ASS-тегов стиля.

**3.2.2. Алгоритм работы OpenCL методов:**

Платформа OpenCL представляет собой совокупность устройств от одного вендора. Основные компоненты:

- Platform: поставщик вычислительных ресурсов (например, NVIDIA, AMD, Intel)  
- Device: конкретный вычислительный блок (GPU, CPU, FPGA)  
- Context: объединяет одно или несколько устройств для совместного использования ресурсов  
- Command Queue: очередь команд, отправляемых от хоста на устройство

**Platform Layer API (вызывается с хоста)**

API уровня платформы обеспечивает абстракцию над разнородными вычислительными ресурсами и отвечает за начальную настройку окружения:

* Абстракция вычислительных ресурсов  
   Открывает информацию о доступных платформах и устройствах (CPU, GPU, FPGA и т. д.), скрывая детали реализации разных вендоров.
* Поиск и инициализация устройств  
   Позволяет выполнить запрос доступных платформ, выбрать платформу нужного вендора, затем перечислить и получить дескрипторы отдельных устройств.
* Создание контекстов и очередей команд  
   Контекст объединяет одно или несколько устройств, а очередь команд (cl\_command\_queue) служит местом постановки задач (копирование данных, запуск кернелов) от хоста к устройству.

**Runtime API (вызывается с хоста)**

Рантайм-API отвечает за управление жизненным циклом и выполнением кернелов:

* Запуск compute-kerne­лов  
   Постановка в очередь команд для выполнения конкретных кернелов на выбранных устройствах.
* Настройка параметров исполнения  
   Определение объёма работы (global и local work‐sizes), настройка аргументов кернела.
* Управление ресурсами  
   Планирование и оркестрация операций копирования, вычислений и работы с памятью, а также синхронизация через события и барьеры.

**Пространство индексов NDRange, Work-Item’ы и Work-Group’ы**

**Индексное пространство NDRange**

* NDRange (N-Dimensional Range) — это N-мерный диапазон индексов, задающий, сколько копий одного и того же кернела будет запущено одновременно.
* NDRange может быть одномерным, двумерным или трёхмерным, в зависимости от задачи:  
  + 1D: обработка одномерных массивов
  + 2D: обработка изображений, матриц
  + 3D: объёмные данные, воксельные решения

При вызове кернела хост-программа указывает:

1. Global Work Size — общее число work-item’ов по каждой из осей
2. Local Work Size — размер work-group’ов по каждой из осей (опционально)

**Work-Item**

* Work-Item — это единичный экземпляр выполнения кернела, «поток» в параллельном конвейере.
* Каждый work-item исполняет тот же вычислительный код, но оперирует своими данными, определяемыми уникальным global ID.

Получить глобальный индекс в кернеле можно функцией  
  
 size\_t gid = get\_global\_id(dim);

* где dim = 0, 1 или 2 для соответствующей оси NDRange.

**Work-Group**

* Work-Group — логическая группа соседних work-item’ов, объединённых для совместного использования локальной памяти (\_\_local) и точек синхронизации (barrier()).
* При запуске кернела NDRange разбивается на несколько work-group’ов одинакового размера (кроме «хвоста», если Global Size не кратен Local Size).

У каждой work-group есть свой Work Group ID, а внутри группы у каждого work-item есть Local ID:  
  
 size\_t lid = get\_local\_id(dim); // индекс внутри группы

size\_t group = get\_group\_id(dim); // номер группы

size\_t lsize = get\_local\_size(dim); // размер группы

Work-group’ы гарантированно могут синхронизировать свои work-item’ы через барьеры:  
  
 barrier(CLK\_LOCAL\_MEM\_FENCE);

**Пример ядра из проекта – VTL\_sub\_opencl\_strip\_tags**

**const char\* kernelSource =**

**"\_\_kernel void strip\_tags(\_\_global const char\* in\_data, \_\_global int\* offsets, \_\_global int\* lengths, \_\_global char\* out\_data, \_\_global int\* out\_offsets) {\n"**

**" int idx = get\_global\_id(0);\n" *// Индекс текущей строки***

**" int in\_offset = offsets[idx];\n" *// Смещение начала строки во входном буфере***

**" int in\_len = lengths[idx];\n" *// Длина строки***

**" int out\_offset = out\_offsets[idx];\n" *// Смещение для записи результата***

**" int in\_tag = 0;\n" *// Флаг: находимся ли внутри тега***

**" int j = 0;\n" *// Индекс для записи в выходной буфер***

**" for (int k = 0; k < in\_len; ++k) {\n"**

**" char c = in\_data[in\_offset + k];\n" *// Читаем символ***

**" if (c == '{') in\_tag = 1;\n" *// Начало тега***

**" else if (c == '}' && in\_tag) in\_tag = 0;\n" *// Конец тега***

**" else if (!in\_tag) out\_data[out\_offset + j++] = c;\n" *// Если не в теге — копируем символ***

**" }\n"**

**" out\_data[out\_offset + j] = '\\0';\n" *// Завершаем строку нулём***

**"}\n";**

**Запуска ядра**

clEnqueueNDRangeKernel  
  
 clEnqueueNDRangeKernel(

queue, kernel,

work\_dim, // число осей (1,2 или 3)

NULL, // смещение (обычно 0)

global\_work\_size, // массив из work\_dim размеров

local\_work\_size, // массив из work\_dim размеров (или NULL)

0, NULL, NULL

);

1. Драйвер разбивает NDRange на work-group’ы, распределяет их по вычислительным блокам GPU.
2. Внутри каждой группы все work-item’ы выполняются одновременно по принципу SIMT (SIMD-подобная модель).  
     
     
   Пример из кода проекта:
3. err = clGetPlatformIDs(1, &platform, **NULL**);
4. **if** (err != CL\_SUCCESS) **return** VTL\_res\_opencl\_kPlatformError;
5. err = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_DEFAULT, 1, &device, **NULL**);
6. **if** (err != CL\_SUCCESS) **return** VTL\_res\_opencl\_kDeviceError;
7. context = clCreateContext(**NULL**, 1, &device, **NULL**, **NULL**, &err);
8. **if** (!context || err != CL\_SUCCESS) **return** VTL\_res\_opencl\_kContextError;
9. queue = clCreateCommandQueue(context, device, 0, &err);
10. **if** (!queue || err != CL\_SUCCESS) { clReleaseContext(context); **return** VTL\_res\_opencl\_kQueueError; }
11. program = clCreateProgramWithSource(context, 1, &kernelSource, **NULL**, &err);
12. **if** (!program || err != CL\_SUCCESS) { clReleaseCommandQueue(queue); clReleaseContext(context); **return** VTL\_res\_opencl\_kProgramError; }
13. err = clBuildProgram(program, 1, &device, **NULL**, **NULL**, **NULL**);
14. **if** (err != CL\_SUCCESS) { clReleaseProgram(program); clReleaseCommandQueue(queue); clReleaseContext(context); **return** VTL\_res\_opencl\_kBuildError; }
15. kernel = clCreateKernel(program, "strip\_tags", &err);
16. **if** (!kernel || err != CL\_SUCCESS) { clReleaseProgram(program); clReleaseCommandQueue(queue); clReleaseContext(context); **return** VTL\_res\_opencl\_kKernelError; }

**Модель выполнения**

Модель выполнения включает следующие этапы:  
1. Хост (Host): основная программа, запускающая OpenCL-код  
2. Программа (cl\_program): содержит исходный код или бинарный байткод кернелов  
3. Кернел (cl\_kernel): функция, выполняемая параллельно на устройстве  
4. NDRange: многомерный диапазон для распределения work-item’ов  
5. Work-group: группа work-items для синхронизации и совместного доступа к локальной памяти

**Модель памяти**

OpenCL определяет четыре уровня памяти:  
 **\_\_global память**

* **Назначение:** основная глобальная память устройства (например, VRAM на GPU).
* **Объём и скорость:** самая большая по объёму, но относительно медленная по доступу.
* **Область видимости:** доступна всем work-item’ам во всех work-group’ах.
* **Использование:** хранение входных и выходных массивов данных, больших буферов, структур. Любые чтения и записи в этой памяти происходят через высокозадержечные операции; поэтому целесообразно минимизировать их количество и объём.

### **\_\_constant память**

* **Назначение:** область памяти только для чтения, оптимизированная для константных данных.
* **Объём и скорость:** обычно мала (несколько десятков килобайт), но может быть закэширована на аппаратном уровне.
* **Область видимости:** доступна всем work-item’ам и work-group’ам, однако значения нельзя изменять во время выполнения кернела.
* **Использование:** передача неизменяемых параметров (например, матрицы трансформации, коэффициентов фильтра), которые читаются многократно; при этом выгода достигается за счёт кэширования и сокращения трафика к глобальной памяти.

### **\_\_local память**

* **Назначение:** быстрая разделяемая память для обмена данными внутри одной work-group’ы.
* **Объём и скорость:** существенно меньше глобальной, но гораздо быстрее (обычно реализуется как SRAM или L1-кэш).
* **Область видимости:** доступна только work-item’ам внутри одной группы; каждая work-group получает свою область \_\_local памяти.
* **Использование:**
  + Совместное кэширование участков глобальной памяти, чтобы сократить повторные обращения.
  + Обмен промежуточными результатами между work-item’ами (например, при реализации редукций, сортировок, свёрток).
  + Синхронизация через barrier(CLK\_LOCAL\_MEM\_FENCE) для согласованного чтения/записи.

### **Приватная память**

* **Назначение:** самый быстрый тип памяти, обычно отображается в регистры вычислительных блоков (CU).
* **Объём и скорость:** крайне ограничена по размеру, но доступ к ней выполняется без задержек.
* **Область видимости:** каждая ячейка приватной памяти привязана к одному work-item’у; другие work-item’ы её не видят.
* **Использование:**
  + Локальные переменные кернела, не объявленные явно как \_\_global, \_\_constant или \_\_local.
  + Хранение промежуточных вычислений, индексов, счётчиков внутри одной инстанции кернела.

**Буферные объекты**

Буферный объект (cl\_mem) — абстракция линейного массива байт, доступного и хосту, и устройству. Основные флаги при создании через clCreateBuffer:  
 - CL\_MEM\_READ\_WRITE, CL\_MEM\_READ\_ONLY, CL\_MEM\_WRITE\_ONLY — права доступа  
 - CL\_MEM\_USE\_HOST\_PTR — прямое использование памяти хоста  
 - CL\_MEM\_ALLOC\_HOST\_PTR — выделение pageable-памяти для ускоренного DMA  
 - CL\_MEM\_COPY\_HOST\_PTR — копирование данных из указанного host\_ptr при создании

**Конвейер работы хост-программы**

1. Инициализация платформы и выбор устройства  
2. Создание контекста и очереди команд  
3. Компиляция программы и создание кернела  
4. Создание и заполнение буферов  
5. Установка аргументов кернела  
6. Запуск кернела через clEnqueueNDRangeKernel  
7. Считывание результатов и освобождение ресурсов