МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Автоматики   
 (полное название кафедры)   
  
  
  
  
  
  
  
  
 **ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**  
Сизова Льва Сергеевича

#### Автор

\_\_\_\_\_\_\_\_Сизов Л.С.

(фамилия, И., О.)

АВТФ, АВТ-318

(факультет, группа)

  
 (подпись, дата)

#### Руководитель

\_\_\_\_\_\_Гунько А.В.

(фамилия, И., О.)

к.т.н., доцент

(уч. степень, уч. звание)

  
 (подпись, дата)

(фамилия, имя, отчество автора – студента-выпускника)   
  
Исследование возможностей библиотеки Vulcan для сложных математических задач

(тема работы)   
  
  
  
Направление подготовки 09.03.01. Информатика и вычислительная техника

(код и наименование направления подготовки бакалавра)

Новосибирск, 2017 г.

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Автоматики *\_*.

(полное название кафедры)

##### Утверждаю

**Зав. кафедрой** Жмудь В.А. .

(подпись, инициалы, фамилия)

. "« » 2017 г..

**ЗАДАНИЕ  
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

студенту Сизову Льву Сергеевичу *\_*

(фамилия, имя, отчество)

Направление подготовки: 09.03.01 Информатика и вычислительная техника *\_*

(код и наименование направления подготовки бакалавра)

*\_*

Факультет автоматики и вычислительной техники .

(полное название факультета)

Тема Исследование возможностей библиотеки Vulcan для сложных математических задач *\_*

(полное название темы выпускной квалификационной работы бакалавра)

.

Исходные данные (или цель работы): исследовать возможность, целесообразность и практическую ценность вычислений на графических процессорах с использованием технологии Vulkan *\_*

Структурные части работы:

Введение *\_*

1. Техническое задание *\_*

2. Обзор библиотеки Vulkan и её функций для вычислений на графических процессорах

3. Проектирование *\_*

4. Реализация *\_*.

5. Тестирование *\_*

Заключение *\_*

Список использованных источников \_ *\_*

Задание согласовано и принято к исполнению.

**Руководитель**

**от НГТУ**

. Гунько А.В. .

(фамилия, имя, отчество)

. к.т.н., доцент .

(ученая степень, ученое звание)

. *\_*

(подпись, дата)

**Автор выпускной квалификационной работы**

. \_\_Сизов Л.С. .

(фамилия, имя, отчество)

АВТФ, АВТ-318 .

(факультет, группа)

. *\_*

(подпись, дата)

Тема утверждена приказом по НГТУ № *\_\_\_\_\_\_\_\_\_* от «*\_\_\_\_*» *\_\_\_\_\_\_\_\_\_\_\_* 2017 г.

ВКР сдана в ГЭК № *\_\_\_\_\_\_\_*, тема сверена с данными приказа

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись секретаря государственной экзаменационной комиссии по защите ВКР, дата)

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(фамилия, имя, отчество секретаря государственной

экзаменационной комиссии по защите ВКР)

***Примечание: поля данного блока заполняются секретарем ГЭК при приеме ВКР к защите.***

Оглавление

[Введение 5](#_Toc484897813)

[1. Техническое задание 8](#_Toc484897814)

[1.1. Сведения и исходные данные для разработки программы 8](#_Toc484897815)

[1.2. Требования к программе 8](#_Toc484897816)

[2. Обзор библиотеки *Vulkan* и её функций для вычислений на графических процессорах 12](#_Toc484897817)

[2.1. Общие сведения 12](#_Toc484897818)

[2.2. Обзор функций, структур данных, утилит 22](#_Toc484897819)

[3. Проектирование 29](#_Toc484897820)

[3.1. Положение библиотеки Vulkalc в иерархии вызовов 29](#_Toc484897821)

[3.2. Сущности Vulkalc 29](#_Toc484897822)

[3.3. Описание процесса запуска задачи на исполнение 30](#_Toc484897823)

[4. Реализация 34](#_Toc484897824)

[4.1. Система сборки проекта 34](#_Toc484897825)

[4.2. Система контроля версий 36](#_Toc484897826)

[4.3. Документация проекта 37](#_Toc484897827)

[4.4. Реализация проекта 38](#_Toc484897828)

[5. Тестирование 49](#_Toc484897829)

[5.1. Модель исполнения тестов фреймворка Catch 49](#_Toc484897830)

[5.2. Модульные тесты библиотеки Vulkalc 49](#_Toc484897831)

[5.3 Тестирование функций Vulkalc 52](#_Toc484897832)

[Заключение 54](#_Toc484897833)

[Список использованных источников 55](#_Toc484897834)

## Введение

Последние поколения процессоров показывают хорошие результаты в параллельных вычислениях, однако до видеокарт им далеко. Причина в том, что видеокарты содержат в себе много графических процессоров, предназначенных для параллельной обработки.

К сожалению, ещё не существует единого стандарта для написания программ для графических процессоров. Основными средствами для написания программ для графических процессоров являются *CUDA* от *NVidia* и *OpenCL* от *AMD*. Однако 16 февраля 2016 года группой компаний *Khronos Group* был выпущен *Vulkan API* предназначенный для программирования графики и вычислений на графических процессорах, используемых в различных платформах – от персональных компьютеров до мобильных телефонов и игровых приставок. Технология *Vulkan* разработана с заделом на будущее, поэтому она будет очень активно развиваться и поддерживаться разработчиками драйверов и графических процессоров.

Для непосредственных вычислений на графических процессорах в библиотеках *Vulkan* и *OpenCL* используется язык программирования шейдеров *GLSL*. Однако для обеспечения кроссплатформенности шейдеры компилируются в промежуточный байт-код *SPIR-V*.

На текущий момент библиотека *Vulkan* используется только такими компаниями, как *Epic Games*, *Unity Technologies* и *Valve Corporation* в их движках и инструментах для разработки компьютерных игр. Также на момент написания этого документа не было найдено ни одной библиотеки для упрощения разработки программ с использованием вычислений на графических процессорах с использованием технологии *Vulkan*. Так как разработка приложений с использованием библиотеки *Vulkan* является не самой тривиальной задачей из-за необходимости гибкой настройки конвейеров и буферов данных, было решено разработать кроссплатформенную библиотеку для этих целей.

В данной работе под сложными математическими задачами подразумеваются вычислительно-сложные математические задачи, требующие большого количество операций над числами, такие как перемножение матриц большого размера и другие.

Новизна данной разработки заключается в том, что это будет первая библиотека для вычислений на графических процессорах, использующая вычислительные возможности видеокарт, работа с которыми происходит с использованием только вычислительных функций библиотеки *Vulkan*. С помощью данной разработки планируется изучить возможность и целесообразность использования самой библиотеки *Vulkan* для математических вычислений на графических процессорах.

Актуальность данной разработки заключается в том, что сейчас происходит лавинообразный рост объёмов информации, которые нужно обрабатывать, проводить вычисления на основе этих данных. Наличие подобной библиотеки ускорит разработку приложений для обработки этих данных. Существуют такие технологии, как *CUDA* и *OpenCL* для высокопроизводительных вычислений на графических процессорах. Однако у них есть свои минусы, которые могут отсутствовать в решениях на основе библиотеки *Vulkan*.

Так как разработчиком библиотеки *Vulkan* заявлены лучшая по сравнению с *OpenGL* поддержка многопоточности, уменьшенная нагрузка на центральный процессор и низкоуровневый контроль над графическим процессором, ожидается производительность вычислений на порядок больше, чем при вычислениях на центральном процессоре и схожая с производительностью вычислений на основе других технологий вычислений на графических процессорах.

Практическая ценность разработки будет выявлена по результатам использования разработанной библиотеки.

Цель работы – исследовать возможность, целесообразность и практическую ценность вычислений на графических процессорах с использованием технологии *Vulkan*.

Задачи:

* разработать библиотеку-обёртку над вызовами библиотеки *Vulkan* и предоставить в ней *API* для вычислений на графических процессорах;
* продемонстрировать пример использования разработанной библиотеки;

## Техническое задание

### Сведения и исходные данные для разработки программы

##### Наименованиепрограммы

Наименование программы: “Библиотека *Vulkalc* для вычислений на графических процессорах для решения вычислительно сложных математических задач”.

##### Сроки разработки библиотеки

Начало разработки: 19.02.2017

Окончание разработки: 10.06.2017

##### Цели создания программы

Упростить разработчикам процесс создания программ с использованием вычислений на графических процессорах с использованием технологии *Vulkan* за счёт предоставления “обёртки” над вызовами библиотеки *Vulkan* и набора функций для использования вычислительных возможностей видеокарт, запуска вычислительных шейдеров и получения результатов расчётов.

##### Назначение и область применения

Разработка предназначена для применения при написании программ для высокопроизводительных математических вычислений на графических процессорах.

### Требования к программе

##### Общие положения

Основные требования, предъявляемые к разрабатываемому продукту:

* Иметь открытый исходный код;
* Иметь открытую лицензию;
* Иметь инструкции по сборке;
* Иметь демонстрационный проект для демонстрации примеров использования;
* Иметь инструкцию по сборке демонстрационного проекта;
* Иметь документацию на английском языке по публичным классам, методам, полям, функциям;
* Компилироваться под операционные системы *Windows* и *Linux;*
* Иметь юнит-тесты;
* Проходить все существующие тесты;
* Использовать систему контроля версий;
* Иметь комментарии в исходном коде;
* Компилироваться и запускать тесты на целевых операционных системах при каждом изменении;
* Иметь как можно меньше зависимостей;

##### Требования к техническому обеспечению

Для использования библиотеки необходим персональный компьютер:

* ОС: 64 разрядные *Windows 7, 8, 8.1, 10* или *Ubuntu 14.04* и выше или аналогичные;
* Процессор: любой *x86-64* процессор;
* Оперативная память: не менее 8ГБ;
* Дисковое пространство: около 2МБ для *Vulkan Runtime*, около 150МБ для *Microsoft Visual C++ Redistributable*(*Windows*), около 150МБ для видеодрайвера *Linux* или около 350МБ для видеодрайвера *Windows*;
* Дискретная видеокарта с поддержкой технологии *Vulkan;*

Данные требования предъявляются библиотекой *Vulkan* [1].

##### Требования к функциональности

Разрабатываемая библиотека *Vulkalc* должна предоставлять следующие возможности:

* Статическое и динамическое связывание с программами на операционных системах *Windows* и *Linux;*
* Проверка наличия поддерживаемых графических процессоров и перечисление установленных в системе поддерживаемых графических процессоров;
* Проверка наличия в системе установленного *Vulkan Runtime;*
* Выбор графического процессора из списка доступных для запуска на нём вычислений;
* Запуск разработанных пользователем вычислительных шейдеров;
* Подготовка установленных поддерживаемых графических процессоров к вычислениям;
* Освобождение графического процессора;
* Запуск и остановка вычислений на поддерживаемых установленных графических процессорах;
* Копирование данных в видеопамять;
* Копирование данных из видеопамяти;
* Выделение видеопамяти для собственных нужд определённого размера;
* Очищение выделенной видеопамяти;
* Сообщение кода и информации об ошибке в случае её возникновения;
* Обнаружение, проверка шейдеров, компиляция шейдеров в промежуточный байт-код *SPIR-V;*

##### 1.2.4. Требования к организации входных данных

При написании собственных вычислительных шейдеров программист сам определяет формат и вид входных данных. Формат и вид входных данных для предоставляемых шейдеров описан в документации проекта.

##### Требования к организации выходных данных

При написании собственных вычислительных шейдеров программист сам определяет формат и вид выходных данных. Формат и вид выходных данных для предоставляемых шейдеров описан в документации проекта.

##### Требования к временным характеристикам

Требования к временным характеристикам не предъявляются

##### Требования к информационному обеспечению

Исходный код программы должен быть написан на языке *C++*. Должна использоваться стандартная библиотека языка *C++* и библиотека *Vulkan*. Классы, структуры, методы, функции, переменные, предназначенные для использования пользователем, должны быть экспортированы в соответствии со стандартом.

##### Требования к программным средствам, используемым программой

Программные средства, используемые библиотекой *Vulkalc*, которые должны быть предоставлены ОС семейства *Windows*:

* *Microsoft Visual Studio* 13 или 15, или *MinGW* с пакетами *mingw-g++* и *mingw-make*;
* *CMake* 2.8.7 и выше;
* *Vulkan Runtime* 1.0.39.1 и выше;
* Для видеокарт *NVidia* – драйвер версии 377.06 и выше, для видеокарт *AMD* – самая новая версия драйвера;
* *Microsoft Visual C++ Redistributable* 2013 и 2015;
* *Doxygen* для генерации документации;

Программные средства, используемые библиотекой *vulkalc*, которые должны быть предоставлены ОС семейства *Linux*:

* *CMake 2.8.7* и выше;
* *Vulkan Runtime 1.0.39.1* и выше;
* *Clang 3.4* и выше или *gcc 4.6.3* и выше;
* Мета-пакет *build-essentials;*
* Пакеты *bison*, *libx11-xcb-dev*, *libxkbcommon-dev*, *libmirclient-dev*, *libwayland-dev*, *libxrandr-dev* для *Ubuntu/Debian* и аналоги этих пакетов для других ОС;
* Стандартная библиотека *C++;*
* Для видеокарт *NVidia* – драйвер версии 377.06 и выше, для видеокарт *AMD* – самая новая версия драйвера;
* *Doxygen* для генерации документации;

Данные требования, кроме *CMake* и *Doxygen*, предъявляются библиотекой *Vulkan* [2].

## Обзор библиотеки *Vulkan* и её функций для вычислений на графических процессорах

### Общие сведения

##### Терминология

В спецификации *Vulkan API* используются понятия, описанные ниже:

* *ABI* (рус. Двоичный интерфейс приложения) – Набор соглашений для доступа к операционной системе и другим низкоуровневым сервисам, спроектированный для переносимости исполняемого кода между машинами, имеющими совместимые ABI
* *API* (рус. Интерфейс программирования приложений) – Набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) или операционной системой для использования во внешних программных продуктах
* *Assembler* (рус. Сборщик) – Один из этапов каждой стадии конвейеров Vulkan, который преобразует входящие данные к нужному виду
* *Command* (рус. Команда) – Непосредственные команды для графического процессора
* *Command Buffer* (рус. Командный буфер) – Структура данных, содержащая команды для графического процессора
* *Compute Shader* (рус. Вычислительный шейдер) – Программа на специальном языке для графического процессора, предназначенная для вычислений
* *CPU* (рус. ЦП) – Центральный процессор
* *Device* (рус. Устройство) – Логическая сущность, представляющая собой физическое устройство. Содержит в себе информацию об устройстве
* *Extension* (рус. Расширение) – Наборы расширения базового функционала Vulkan с помощью обратных вызовов
* *Fragment Shader* (рус. Фрагментный (пиксельный) шейдер) – Один из этапов графического конвейера
* *Geometry Shader* (рус. Геометрический шейдер) – Один из этапов графического конвейера
* *GPU* (рус. ГП) – Графический процессор
* *Host* (рус. Хост) – Программа, отправляющая команды на графический процессор
* *Physical Device* (рус. Физическое устройство) – Графический процессор, выполняющий команды, отправленные с хоста
* *Queue* (рус. Очередь) – Структура данных, в которую поступают команды для устройства
* *Queue Family* (рус. Семейство очередей) – Набор очередей, предоставляющих определённые возможности
* *SDK* – Комплект средств разработчика для создания программ для определённой платформы
* *Shader* (рус. Шейдер) – Программа на специальном языке, предназначенная для исполнения на ГП. В Vulkan представляет собой программируемую стадию графического конвейера
* *SPIR-V* – Промежуточный кроссплатформенный шейдерный байт-код
* *Tessellation* (рус. Тесселяция) – Автоматизированный процесс добавления новых выпуклых многоугольников в полигональную сетку с целью повышения детализации сетки.
* *Tessellation Shader* (рус. Шейдер тесселяции(замощения)) – Один из этапов графического конвейера
* *Validation Layer* (рус. Слой валидации) – Прослойка между драйвером и хостом для перехвата ошибок и обработок ошибок, записи логов и трассировки вызовов
* *Vertex Shader* (рус.Вершинный шейдер) – Один из этапов графического конвейера

##### Архитектурная модель

Согласно спецификации, библиотека *Vulkan* спроектирована и написана для ЦП, ГП и других аппаратных ускорителей со следующими свойствами:

* Поддержка 8, 16, 32 и 64 битных знаковых и беззнаковых целых с обратным кодом для представления отрицательных чисел
* Поддержка 32 и 64 битных чисел с плавающей запятой с точностью в 6 знаков
* Порядок байтов в двоичном представлении чисел должен быть одинаковый для хоста и физического устройства

Библиотека *Vulkan* представляет собой интерфейс программирования приложений со свободными точками входа в функции. Большая часть параметров и возвращаемых типов представляет собой структуры языка *C,* обязательные для заполнения. Все указатели по умолчанию имеют размер 64 бита, если не указан явный размер в 32 бита или на платформе *x86.*

Двоичный интерфейс приложения предоставляется производителем драйвера для конкретной платформы. Например, поставщиком *ABI* для *Android* является компания *Google.*

В исходном виде, драйвер *Vulkan* не возвращает никаких ошибок, кроме критических, и не проверяет поступающие данные. Для проверки данных используются слои, которые играют роль прослойки между хостом и драйвером. В этих слоях происходит проверка входящих данных, но никакой информации о ошибках или успехе проверок они не возвращают, а просто предотвращают критические сбои в работе программы [3, с. 6].

Рисунки 2.1 и 2.2 демонстрируют механизм работы слоёв в случае успешного прохождения всех проверок (Рисунок 2.1) и в случае провала одной из проверок (Рисунок 2.2).

|  |  |
| --- | --- |
| C:\Users\whyami\Desktop\normal.jpg | C:\Users\whyami\Desktop\fail.jpg |
| Рис. 2.1. | Рис. 2.2. |

Базовый функционал *Vulkan* можно расширить с помощью расширений, основная часть которых поставляется вместе с *SDK*; другие расширения поддерживаются поставщиками драйверов и *ABI*. Примером расширения является базовое расширение *VK\_EXT\_debug\_report* позволяет получать детальную информацию о поведении программы и ошибках при её исполнении. Примером стороннего расширения является расширение *VK\_NV\_external\_memory\_capabilities* от *NVidia*, которое позволяет использовать память в приложении *Vulkan* память, выделенную средствами *Direct3D*.

При включенном расширении *VK\_EXT\_debug\_report* у слоя появляется возможность передавать информацию об ошибках на предыдущий слой. Тогда механизм работы слоёв меняется (Рисунок 2.3) и появляется возможность передавать отладочную информацию или информацию об ошибках при проверках в слоях назад.

##### Конвейеры *Vulkan*

Согласно спецификации, библиотека *Vulkan* предоставляет два конвейера для использования: графический и вычислительный. Они изображены на рисунке 2.4. Структуру конвейеров и порядок стадий изменить нельзя (можно только пропустить некоторые стадии). Жёлтым цветов на рисунке 2 выделены программируемые стадии конвейера, называемые шейдерами.

В шейдер передаётся необходимая информация через *Buffer* или *Image*. Где первый может содержать любые данные, а второй только информацию о пикселях в определённом изображении.

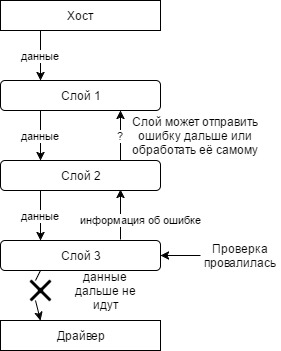


Рис. 2.3. Механизм работы слоёв со включенным расширением обработки ошибок

Перед тем как запустить шейдер на выполнение, необходимо передать ему полученную из приложения информацию. Это происходит на этапе *Assembler,* который присутствует у каждой стадии конвейера. *Assembler* занимается тем, что преобразует читает и преобразует в нужный формат полученные данные и передают дальше в шейдер.

В графическом конвейере существуют следующие стадии:

* Вершинный шейдер – производит преобразования для переноса позиций вершин из пространства моделей в пространство экрана. Он так же передает данные по вершинам далее по конвейеру;
* Шейдер тесселяции (опциональный) – позволяет разбивать трёхмерную сетку модели на основе определённых правил для повышения качества сетки;
* Геометрический шейдер – позволяет добавлять, изменять и удалять трёхмерные примитивы
* Фрагментный(пиксельный) шейдер – производит выборку из текстур и/или математические операции над цветом и значением глубины пикселей для каждого отображаемого на экране пикселя



Рис. 2.4. Графический и вычислительный конвейеры *Vulkan*

Вычислительный конвейер состоит из всего одной стадии – вычислительного шейдера. Он позволяет производить произвольные вычисления для, например, симуляции системы частиц или расчёта отражения лучей [3, с. 167].

##### Модель выполнения

Согласно спецификации, библиотека *Vulkan* даёт доступ к физическому устройству через одну или несколько логических сущностей Устройство. Оно в своё очередь предоставляет доступ к очередям выполнения команд. Наборы поддерживаемых ГП очередей разбиты на семейства. Каждое семейство поддерживает функциональность определённого типа и может содержать несколько очередей со схожими параметрами. Существует 4 типа очередей:

* Для работы с графикой
* Для работы с вычислениями
* Для перемещения данных
* Для работы с разреженной памятью

Память на физическом устройстве может быть выделена только явным образом с хоста. Каждое устройство может создать несколько блоков памяти типа куча, которые могут размещаться как на физическом устройстве, так и на хосте, причём вне зависимости от типа кучи, эта память всегда видна хосту.

Виды памяти типа куча:

* *DLHV heap* (*device local, host visible*) – память физически размещена на физическом устройстве, видна хосту
* *HLHV heap* (*host local, host visible*) – память размещена на хосте, видна и хосту и устройству

Приложение на *Vulkan* отправляет устройствам командный буфер, содержащий записанные команды для физического устройства. После того, как создан командный буфер, он может быть поставлен в очередь на исполнение неограниченное количество раз. Спецификация не определяет, что произойдёт, если отправить несколько командных буферов в несколько разных очередей – командные буферы могут исполниться параллельно или не в том порядке, в каком они были отправлены в очереди. Но в пределах одной очереди командные буферы выполняются в том порядке, в котором они были отправлены [3, с. 7].

Выполнение командных буферов по отношению к приложению-хосту происходит асинхронно. О синхронизации между очередями, устройством и хостом должен заботиться хост.

Командные буферы могут находиться в нескольких состояниях:

* Первичное – в этом состоянии буфер находится, когда он только был создан
* Режим записи – буфер записывает все команды для исполнения устройством, которые были вызваны функциями вида *vkCmd\**
* Исполнение – буфер записал все нужные команды и готов быть переданным на исполнение в очередь
* В ожидании – буфер команд передан в очередь и исполняется устройством. В этом состоянии нельзя изменять буфер
* Ошибка – буфер в этом состоянии может быть только освобожден, сброшен на исходное состояние или включен в режим записи

##### Операции в очередях

Очереди в *Vulkan* предоставляют возможность запуска команд на устройстве. Команды записываются в командный буфер вместе с командой *Queue submission* (англ. - отправка в очередь) и запускаются на исполнение специальными движками исполнения. После того, как команды были отправлены в очередь, хост не может никак повлиять на работу этих команд. Вместе с командным буфером можно передать набор семафоров для того, чтобы сигнализировать устройству о том, что можно начать выполнение, или для того, чтобы устройство сообщило об окончании определённой операции.

Спецификация *Vulkan* не определяет порядок выполнения команд в разных очередях – если передать два буфера команд в две разные очереди, то нельзя заранее сказать, какой из них выполнится первым.

Порядок выполнения командных буферов можно контролировать с помощью примитивов синхронизации, которые подробнее описаны в пункте 2.1.7. Не смотря на асинхронную природу выполнения команд устройством, гарантируется, что все задачи, стоящие до примитива синхронизации, будут выполнены к тому моменту, когда примитив синхронизации отправит сигнал о синхронизации устройству.

Команды в очередях могут выполнять действия (рисовать, копировать память, очищать, начинать/останавливать подпроходы отрисовки), изменять состояние (состояние подпрохода отрисовки, отправка констант, привязка буферов к очередям, конвейерам) или производить операции синхронизации.

Так как команды для устройства выполняются асинхронно, то управление может сразу же вернуться хосту после отправки командного буфера в очередь.

##### Объектная модель

Согласно спецификации, устройства, очереди и сущности *Vulkan* представлены в виде объектов. Все объекты на уровне *API* управляются через их дескрипторы. Всего существуют дескрипторы двух видов: управляемые и неуправляемые. Управляемые дескрипторы — это указатели на непрозрачные типы данных (типы данных, не объявленные в интерфейсах, скрытые от глаз). Обычно такие дескрипторы используются для перехвата вызовов *API* драйвером и чаще всего передаются в функции первым параметром. Спецификация *Vulkan* определяет, что каждый управляемый объект должен иметь один уникальный управляемый дескриптор в течение всего времени жизни объекта.

Неуправляемые дескрипторы – целочисленные 64 битные типа данных, применение которых может меняться в зависимости от реализации. Неуправляемые объекты могут иметь несколько неуправляемых дескрипторов, причём гарантируется, что уничтожение одного дескриптора не уничтожит другие дескрипторы.

Все объекты, созданные с помощью сущности логического устройства *VkDevice* видны и могут быть использованы только этим логическим устройством.

Объекты создаются с помощью функций вида *vkCreate\**, другие объекты создаются выделением памяти функцией вида *vkAllocate\**. Как только объект был создан, его структура становится иммутабельной (англ. *Immuatble* - неизменяемый), но данные в структуре изменять не запрещено.

Предполагается, что создание и уничтожение объектов происходит нечасто, в отличии от выделения и освобождения памяти под объекты. Поэтому объектов, требующих явного выделения памяти рекомендуется использоваться пулы и кучи. Уничтожение объектов осуществляется функциями вида *vkDestroy\**, освобождение памяти – *vkFree\**.

Драйвер никак не контролирует время жизни объектов, перекладывая эту задачу на плечи хоста. Хост сам должен контролировать, чтобы объекты не уничтожались во время использования.

Существует также набор объектов, которые нельзя уничтожать при любых или определённых условиях.

Никакой командой ни при каких условиях нельзя уничтожать объекты *VkShaderModule* и *VkPipelineCache*

Нельзя уничтожать объект *VkPipelineLayout*, если какой-либо буфер команд находится в режиме записи

Если командный буфер находится в режиме ожидания, то никакой из следующих объектов, используемых командным буфером, не должен быть уничтожен: *VkEvent, VkQueryPool, VkBuffer, VkBufferView, VkImage, VkImageView, VkPipeline, VkSampler, VkDescriptorPool, VkFramebuffer, VkRenderPass, VkCommandBuffer, VkCommandPool, VkDeviceMemory, VkDescriptorSet.*

Если очередь выполняет команды, то следующие объекты, используемые в буферах команд, не должны быть уничтожены: *VkFence, VkSemaphore, VkCommandBuffer, VkCommandPool.*

Объекты типа *VkQueue* не могут быть уничтожены явно – они уничтожаются автоматически вместе с объектом устройства *VkDevice*, с помощью которого они были созданы

При уничтожении объекта пула, все объекты из этого пула уничтожаются неявным образом

Объект логического устройства *VkDevice* может быть уничтожен только тогда, когда все связанные с ним очереди находятся в режиме ожидания, т.е. не имеют команд в очереди, и уничтожены следующие объекты, связанные с очередями: *VkFence, VkSemaphore, VkEvent, VkQueryPool, VkBuffer, VkBufferView, VkImage, VkImageView, VkShaderModule, VkPipelineCache, VkPipeline, VkPipelineLayout, VkSampler, VkDescriptorSetLayout, VkDescriptorPool, VkFramebuffer, VkRenderPass, VkCommandPool, VkCommandBuffer, VkDeviceMemory.*

Объект физического устройства *VkPhysicalDevice* не должен уничтожаться явным образом, он уничтожается вместе с объектом *VkInstance* в конце цикла жизни программы [3, с. 9].

##### Механизмы синхронизации

Согласно спецификации, синхронизация доступа к ресурсам является первичной задачей приложения-хоста. Существует несколько неявных гарантий порядка запуска команд, однако в большинстве случаев это нужно делать самому, как и управление кэшем и другими оптимизациями.

*Vulkan* предоставляет 4 примитива синхронизации:

* Барьер/Забор (англ. *Fence*) – используются для сообщения хосту о завершении команды
* Семафор – используется для контроля доступа к ресурсам из нескольких очередей
* События – используются для сообщения о наступлении какого-либо события из хоста или командного буфера.
* Конвейерный барьер (англ. *Pipeline Barrier*) – используются для синхронизации внутри командных буферов, но только в одном месте за раз [3, с. 77].

### Обзор функций, структур данных, утилит

В этой главе будет осуществлён обзор утилит, поставляемых вместе с *SDK Vulkan*, а также функции и структуры данных, необходимых для реализации вычислений на ГП с помощью технологии *Vulkan*. Так как основное предназначение библиотеки *Vulkan* это отрисовка графики и связанных с ней вычислений, очень многие функции будут не нужны.

##### Обзор утилит

В комплекте с *SDK Vulkan* поставляются следующие утилиты:

* *cube* и *cube\_32* – демонстрационная программа с текстурированным вращающимся кубом. Версии *x86-64* и *x86* соответственно;
* *glslangValidator –* компилятор и валидатор. Проверяет валидность кода на *GLSL* и компилирует его в *SPIR-V*-совместимый байт-код;
* *glslc –* компилятор *GLSL* в *SPIR-V*-совместимый байт-код;
* *spirv-as* – компилирует удобочитаемый *SPIR-V* код в байт-код *SPIR-V*;
* *spirv-cfg –* строит граф потока управления шейдера *SPIR-V*;
* *spirv-cross –* транслирует байт-код *SPIR-V* в удобочитаемые файлы на шейдерных языках *GLSL*, *MSL* (экспериментальная версия), *C++* (экспериментальная версия);
* *spirv-dis –* дизассемблирует байт-код *SPIR-V* в удобочитаемый *SPIR-V* код;
* *spirv-opt* – оптимизирует байт-код *SPIR-V*;
* *spirv-remap* – сжимает и оптимизирует байт-код *SPIR-V*, удаляя неиспользуемые функции и отладочные символы;
* *spirv-val –* валидатор байт-кода *SPIR-V*;
* *via –* проверяет наличие и валидность установленного драйвера *Vulkan* и наличие и состояние компонентов *Vulkan*;
* *vkreplay* – воспроизводит файл трассировки без необходимости в приложении, с которого была записана трассировка;
* *vktrace* – трассировщик вызовов *API Vulkan*  [4]*.*

##### Обзор структур данных

Так как библиотека *Vulkan* предоставляет *C API,* в ней нет классов и интерфейсов, только структуры. Структуры языка *С* используются как передаваемые в функции параметры и как возвращаемые типы. Названия некоторых структур, а также описание их полей и предназначения описаны ниже.

* *VkInstanceCreateInfo* – Хранение и изменение информации о приложении:
  + *VkStructureType* *sType* – Тип структуры;
  + *const* *void\** *pNext* – 0 или указатель на следующую структуру;
  + *const char\* pApplicationName* – Указатель на строку с названием приложения;
  + *uint32\_t applicationVersion* – Номер версии приложения;
  + *const char\* pEngineName* – Имя движка;
  + *uint32\_t engineVersion* – Номер версии движка;
  + *uint32\_t apiVersion* – Версия *API Vulkan*;
* *VkDeviceCreateInfo* – Параметры для создаваемого логического устройства:
  + *VkStructureType sType* – Тип структуры;
  + *const void\* pNext* – 0 или указатель на следующую структуру;
  + *VkDeviceCreateFlags flags* – Зарезервировано;
  + *uint32\_t queueCreateInfoCount* – Размер массива структур с информацией для создания очередей;
  + *const VkDeviceQueueCreateInfo\* pQueueCreateInfos* – Массив структур с информацией для создания очередей;
  + *uint32\_t enabledLayerCount* – Не используется;
  + *const char\* const\* ppEnabledLayerNames* – Не используется;
  + *uint32\_t enabledExtensionCount* – Количество расширений, которое нужно включить;
  + *const char\* const\* ppEnabledExtensionNames* – Указатель на массив строк с именами расширений, которые нужно включить;
  + *const VkPhysicalDeviceFeatures\* pEnabledFeatures* – 0 или указатель на структуру с описанием включенных свойств графического процессора;
* *VkDeviceQueueCreateInfo* - Информация о создаваемой очереди:
  + *VkStructureType sType* – Тип структуры;
  + *const void\* pNext* – 0 или указатель на следующую структуру;
  + *VkDeviceQueueCreateFlags flags* – Зарезервировано;
  + *uint32\_t queueFamilyIndex* – Индекс семейства очередей у устройства;
  + *uint32\_t queueCount* – Количество очередей для создания;
  + *const float\* pQueuePriorities* – Массив чисел с плавающей запятой с приоритетами для каждой создаваемой очереди;
* *VkPhysicalDeviceFeatures* – Поддерживаемые ГП особенности;
* *VkPhysicalDeviceProperties* – Свойства физического устройства ГП:
  + *uint32\_t apiVersion* – Версия API Vulkan;
  + *uint32\_t driverVersion* – Номер драйвера;
  + *uint32\_t vendorID* – Идентификатор производителя ГП;
  + *uint32\_t deviceID* – Идентификатор устройства;
  + *VkPhysicalDeviceType deviceType* – Тип устройства;
  + *char deviceName[VK\_MAX\_PHYSICAL\_DEVICE\_NAME\_SIZE]* – Название устройства;
  + *uint8\_t pipelineCacheUUID[VK\_UUID\_SIZE]* – Уникальный идентификатор устройства;
  + *VkPhysicalDeviceLimits limits* – Физические ограничения устройства;
  + *VkPhysicalDeviceSparseProperties sparseProperties* – Свойства разреженной памяти устройства;
* *VkPhysicalDeviceLimits* – Физические ограничения размеры различных параметров физического устройства ГП;
* *VkPhysicalDeviceMemoryProperties* – Свойства памяти физического устройства ГП:
  + *uint32\_t memoryTypeCount* – Количество типов памяти;
  + *VkMemoryType memoryTypes[VK\_MAX\_MEMORY\_TYPES]* – Массив структур, описывающих доступные типы памяти;
  + *uint32\_t memoryHeapCount* – Количество куч;
  + *VkMemoryHeap memoryHeaps[VK\_MAX\_MEMORY\_HEAPS]* – Массив структур, описывающих кучи памяти;
* *VkMemoryHeap* – Информация о куче:
  + *VkDeviceSize size* – Размер области памяти типа куча;
  + *VkMemoryHeapFlags flags* – Флаги памяти в виде битовой маски;
* *VkMemoryType* – Информация о типе памяти:
  + *VkMemoryPropertyFlags propertyFlags* - Битовая маска с параметрами;
  + *uint32\_t heapIndex* – Куче с каким индексом соответствует эта структура;

##### Обзор функций и команд

Для задач вычислений на ГП, основной интерес представляют следующие команды и функции:

* *VkResult vkAllocateMemory(VkDevice, const VkMemoryAllocateInfo\*, const VkAllocationCallbacks\*, VkDeviceMemory\*)* – выделяет память заданного размера на заданном устройстве
* *VkResult vkBeginCommandBuffer(VkCommandBuffer, const VkCommandBufferBeginInfo\*)* – переключает заданный командный буфер в режим записи команд для устройства
* *void vkCmdCopyBuffer(VkCommandBuffer commandBuffer, VkBuffer srcBuffer, VkBuffer dstBuffer, uint32\_t regionCount, const VkBufferCopy\*)* – записывает в командный буфер команду копирования содержимого одного буфера в другой
* *void vkCmdFillBuffer(VkCommandBuffer commandBuffer, VkBuffer dstBuffer, VkDeviceSize dstOffset, VkDeviceSize size, uint32\_t data)* – записывает в командный буфер команду заполнения заданного буфера фиксированными данными
* *VkResult vkCreateBuffer(VkDevice device, const VkBufferCreateInfo\* pCreateInfo, const VkAllocationCallbacks\* pAllocator, VkBuffer\* pBuffer)* – создаёт новый объект буфера
* *VkResult vkCreateDevice(VkPhysicalDevice physicalDevice, const VkDeviceCreateInfo\* pCreateInfo, const VkAllocationCallbacks\* pAllocator, VkDevice\* pDevice)* – создаёт новый объект логического устройства
* *VkResult vkCreateFence(VkDevice device, const VkFenceCreateInfo\* pCreateInfo, const VkAllocationCallbacks\* pAllocator, VkFence\* pFence)* – создаёт новый примитив синхронизации типа барьер
* *VkResult vkCreateInstance(const VkInstanceCreateInfo\* pCreateInfo, const VkAllocationCallbacks\* pAllocator, VkInstance\* pInstance)* – создаёт объект экземпляра приложения *Vulkan*
* *void vkDestroyBuffer(VkDevice device, VkBuffer buffer, const VkAllocationCallbacks\* pAllocator)* – уничтожает буфер и все данные в нём
* *void vkDestroyDevice(VkDevice device, const VkAllocationCallbacks\* pAllocator)* – уничтожает объект логического устройства
* *void vkDestroyFence(VkDevice device, VkFence fence, const VkAllocationCallbacks\* pAllocator)* – уничтожает примитив синхронизации типа барьер
* *void vkDestroyInstance(VkInstance instance, const VkAllocationCallbacks\* pAllocator)* – уничтожает объект экземпляра программы *Vulkan*
* *VkResult vkEnumeratePhysicalDevices(VkInstance instance, uint32\_t\* pPhysicalDeviceCount, VkPhysicalDevice\* pPhysicalDevices)* – перечисляет все существующие в системе физические устройства
* *void vkFreeMemory(VkDevice device, VkDeviceMemory memory, const VkAllocationCallbacks\* pAllocator)* – освобождает память, выделенную на устройстве
* *VkResult vkResetCommandBuffer(VkCommandBuffer commandBuffer, VkCommandBufferResetFlags flags)* – сбрасывает командный буфер в исходное состояние
* *VkResult vkCreateShaderModule(VkDevice device, const VkShaderModuleCreateInfo\* pCreateInfo, const VkAllocationCallbacks\* pAllocator, VkShaderModule\* pShaderModule)* – создаёт объект модуля шейдера
* *VkResult vkCreateComputePipelines(VkDevice device, VkPipelineCache pipelineCache, uint32\_t createInfoCount, const VkComputePipelineCreateInfo\* pCreateInfos, const VkAllocationCallbacks\* pAllocator, VkPipeline\* pPipelines)* – создаёт объект вычислительного конвейера для запуска вычислительных шейдеров

Общие шаги для создания, инициализации приложения *Vulkan* и запуска вычислительного шейдера:

1. Создать объект *VkInstance* с нужными параметрами с помощью *vkCreateInstance()*;
2. Просмотреть существующие устройства ГП и получить идентификатор нужного с устройства *VkPhysicalDevice*;
3. Создать объект логического устройства *VkDevice* с помощью метода *vkCreateDevice()*;
4. Получить объект очереди VkQueue от устройства с помощью функции *vkGetDeviceQueue()*;
5. Создать модуль шейдера *VkShaderModule* с помощью функции *vkCreateShaderModule(),* передав в него шейдерный код;
6. Создать вычислительный конвейер *VkPipeline* с помощью метода *vkCreateComputePipelines*;
7. Выделить память, записать туда данные;
8. Создать командный буфер, привязать к конвейеру с помощью *vkCmdBindPipeline()*;
9. Записать команду на запуск шейдера с помощью *vkCmdDispatch()*;
10. Запустить командный буфер на исполнение, отправив в очередь с помощью *vkQueueSubmit().*

## Проектирование

### Положение библиотеки Vulkalc в иерархии вызовов

Библиотека Vulkalc является промежуточным звеном между пользовательским приложением, которое занимается вычислениями и средой исполнения Vulkan. Важно отметить, что библиотека не является высокоуровневой обёрткой над вызовами Vulkan, а предоставляет собственный интерфейс для удобной работы с вычислениями, скрывая непосредственную работу со средой выполнения Vulkan внутри себя.

Рисунок 3.1 демонстрирует положение библиотеки Vulkalc в иерархии вызовов функций Vulkan и работы с графическим процессором.

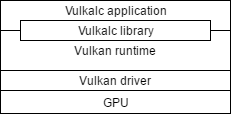


Рис. 3.1. Иерархия вызовов функций Vulkan

### Сущности Vulkalc

Библиотека Vulkalc экспортирует одно пространство имён Vulkalc для предотвращения пересечения имён. В этом пространстве имён содержатся классы, предоставляющие доступ ко всем функциям библиотеки. На рисунке 3.2 изображена структурная диаграмма основных сущностей.

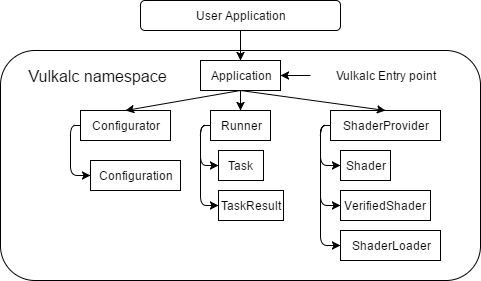


Рис 3.2. Структурная диаграмма сущностей

Точкой входа в библиотеку и предоставляемые библиотекой классы и функции является класс *Application*. Далее описаны основные сущности *Vulkalc* и их назначение:

1. *Configurator* – конфигурирует приложение информацией о слоях валидации *Vulkan*, используемом графическом процессоре и другой информацией;
   1. *Configuration* – содержит информацию, используемую Configurator для конфигурации приложения;
2. *Runner* – сущность, контролирующая запуск задач на вычисление и получение результатов;
   1. *Task* – сущность, выражающая собой задачу на вычисление, содержащая буфер входных данных и выбранный вычислительный шейдер;
   2. *TaskResult* – результат вычисления последней задачи, содержит буфер выходных данных;
3. *ShaderProvider* – предоставляет доступ к пользовательским шейдерам и шейдерам, идущим в комплекте в библиотекой, предоставляет логическую сущность шейдера;
   1. *Shader* – вычислительный шейдер, загруженный с диска;
   2. *VerifiedShader* – вычислительный шейдер, проверенный и скомпилированный с помощью утилиты *glslangValidator*.

### Описание процесса запуска задачи на исполнение

Библиотека Vulkalc предоставляет единый процесс загрузки и запуска шейдеров и для пользовательских шейдеров, и для предоставляемых библиотекой шейдеров. Процесс загрузки шейдеров продемонстрирован на рисунке 3.3.

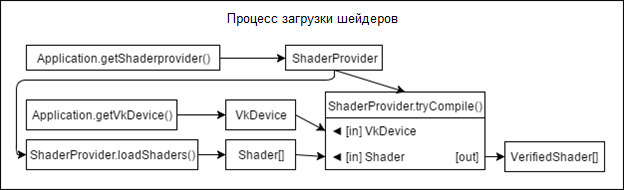


Рис. 3.3. Процесс загрузки шейдеров

Загрузка шейдеров состоит из следующих этапов:

1. Получаем экземпляр загрузчика шейдеров *ShaderProvider* с помощью метода *getShaderProvider* объекта класса *Application*;
2. Вызываем у экземпляра класса *ShaderLoader* метод *loadShaders*, который возвращает массив сущностей шейдера *Shader*;
3. Вызываем метод *tryCompile* у объекта класса *ShaderProvider*, передаём в него *VkDevice*, полученный с помощью метода *getVkDevice* объекта класса *Application*, и объект шейдера или массив объектов шейдера;
4. Метод *tryCompile* возвращает массив сущностей скомпилированных шейдеров *VerifiedShader*. Стоит отметить, что шейдеры, которые не удалось скомпилировать, не попадают в этот массив. Компиляция выполняется инструментом *glslangValidator*.

После того, как шейдеры были скомпилированы и загружены, их можно использовать для запуска задач вычисления. Процесс запуска шейдеров на выполнение продемонстрирован на рисунке 3.4 и описан ниже.

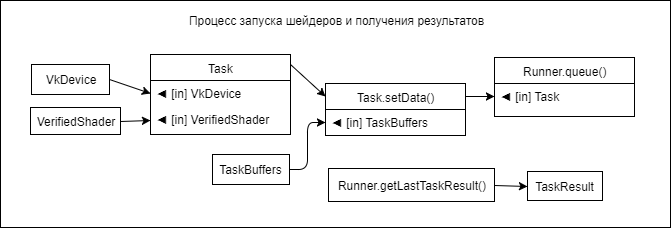


Рис. 3.4. Процесс запуска шейдера на исполнение

Этапы запуска шейдера на вычисление:

1. Создать задачу вычисления *Task*;
2. В задачу *Task* передать выбранный верифицированный и скомпилированный шейдер *VerifiedShader* и *VkDevice*, полученный от *Application*. При необходимости передать буфер со входными данными*;*
3. Передать задачу *Task* в метод *queue* класса *Runner*,полученного из *Application* с помощью метода *getRunner;*
4. Получить из класса *Runner* результат вычисления *TaskResult* с помощью метода класса *Runner* *getLastTaskResult*. Стоит отметить, что если к моменту вызову *getLastTaskResult* результат ещё не будет готов, то процесс будет ожидать получения результата.

Библиотека *Vulkalc* использует исключения для уведомления об ошибках, как это рекомендуется в стандартах [александреску, 72]. Диаграмма классов исключений представлена ниже на рисунке 3.5.

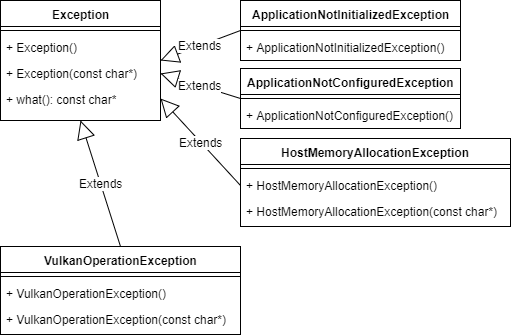


Рис. 3.5. Диаграмма классов исключений

На рисунке 3.6 представлена диаграмма всех остальных классов, экспортируемых библиотекой *Vulkalc*.

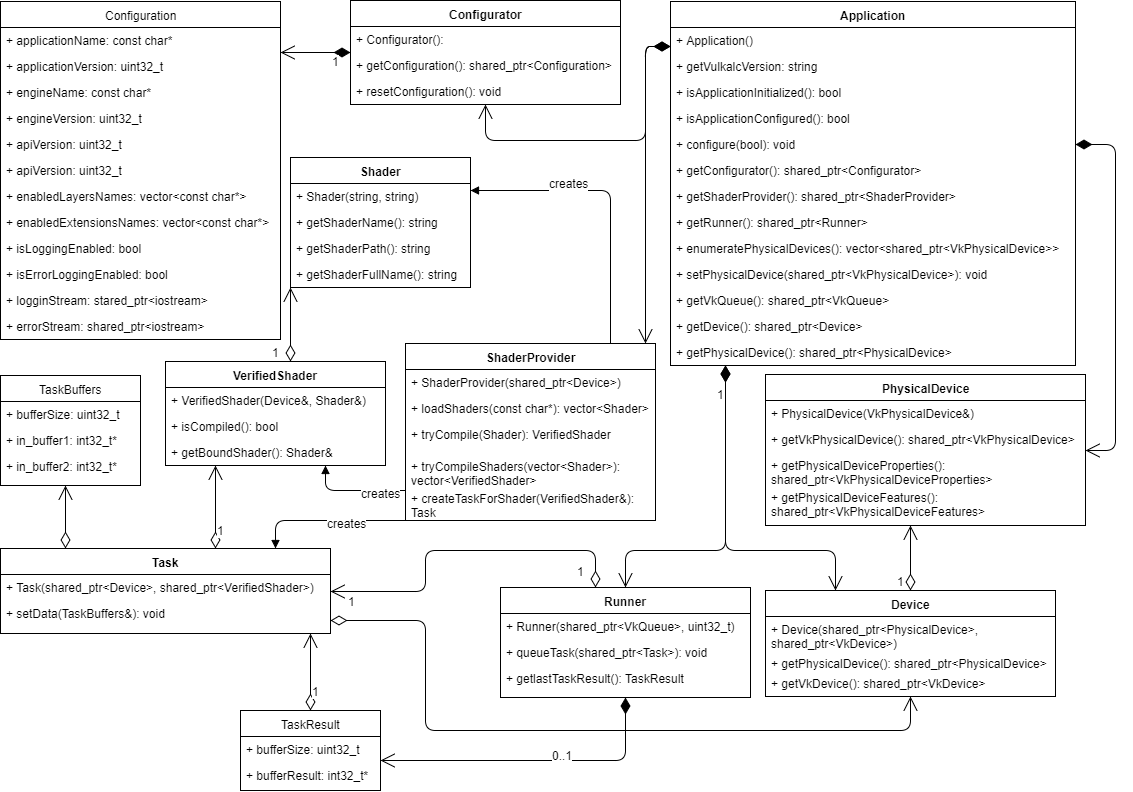


Рис. 3.6. Диаграмма классов библиотеки *Vulkalc*

## Реализация

### Система сборки проекта

Стандарты программирования на *C++* рекомендуют использовать автоматические системы сборки [5, c. 19]. Так как к проекту предъявлено требование кроссплатформенности, то и система сборки должна быть кроссплатформенной. Была выбрана система сборки *CMake*, доступная для операционных систем *Windows* и *Linux.* С помощью *CMake* производится конфигурация сборки и генерация проекта. Таким образом, для *Windows* можно создать проект *Visual Studio*, а для *Linux* создать проект на основе *Makefiles*. Конфигурация проекта производится с помощью задания значений существующим переменным *CMake*. Например, установив переменную *VULKALC\_BUILD\_STATIC=0*, при компиляции будет создана динамическая библиотека, а не статическая. Все существующие переменные *CMake*, определяемые *Vulkalc*, описаны в таблице 2.

Таблица 1 – Переменные *CMake*

|  |  |  |
| --- | --- | --- |
| Название переменной | Назначение переменной | Значение по умолчанию |
| *VULKALC\_BUILD\_VULKALC* | Собирать библиотеку (1) или нет (0) | 1 |
| *VULKALC\_BUILD\_STATIC* | Создаётся динамическая (0) или статическая (1) библиотека | 1 |
| *VULKALC\_INCLUDE\_TESTS* | Включать в сборку модульные тесты (1) или нет (0) | 1 |
| *VULKALC\_INCLUDE\_SAMPLE* | Включать в сборку демонстрационный проект (1) или нет (0) | 1 |
| *VULKALC\_GENERATE\_DOC* | Добавлять в сборку цель для генерации документации (1) или нет (0) | 1 |
| *VULKALC\_GENERATE\_HTML* | Генерировать документацию в формате HTML (1) или нет (0) | 1 |
| *VULKALC\_GENERATE\_PDF* | Генерировать документацию в формате PDF (1) или нет (0) | 0 |
| *VULKALC\_GENERATE\_LATEX* | Генерировать документацию в формате LaTeX (1) или нет (0) | 0 |
| *VULKALC\_GENERATE\_MAN* | Генерировать документацию в виде Linux manual (1) или нет (0) | 0 |
| *VULKALC\_GENERATE\_RTF* | Генерировать документацию в формате RTF (1) или нет (0) | 0 |

Так как библиотека *Vulkalc* напрямую зависит от *Vulkan SDK*, в сборочных скриптах *CMake* проверяется наличие переменной среды *VULKAN\_SDK*, которая должна задаваться при установке *Vulkan SDK*.

Листинг 1. Проверка существования переменной среды *VULKAN\_SDK*

if (NOT EXISTS "$ENV{VULKAN\_SDK}")

message(FATAL\_ERROR "Vulkalc requires Vulkan SDK installed.")

endif (NOT EXISTS "$ENV{VULKAN\_SDK}")

Так как *Vulkan SDK* содержит библиотеки, скомпилированные под *x86* и *x86-64*, то нужно определить разрядность целевой платформы, чтобы включать библиотеку нужной разрядности. Для этого в *CMake* производится попытка компиляции файла *detect\_arch.c*, который с помощью директивы препроцессора C *#error* выводит в консоль разрядность целевой системы. Далее в скрипте *CMake* считывается это сообщение, и на его основе включаются библиотеки *Vulkan* нужной разрядности в сборку.

Листинг 2. Определение архитектуры целевой операционной системы

#if defined(i386) || defined(\_\_i386) || defined(\_\_i386\_\_) || defined(\_M\_IX86) || defined(\_X86\_) || defined(\_\_386) || defined(\_\_X86\_\_) || defined(\_\_I86\_\_)

#error cmake\_ARCH i386

#elif defined(\_\_x86\_64) || defined(\_\_x86\_64\_\_) || defined(\_\_amd64) || defined(\_\_amd64\_\_) || defined(\_M\_X64) || defined(\_M\_AMD64)

#error cmake\_ARCH amd64

#else

#error cmake\_ARCH unknown

#endif

Библиотека Vulkalc содержит в себе модульные тесты для тестирования некоторых отдельных частей функциональности, среди которых есть тесты, напрямую зависящие от наличия графического процессора. Также используются сервисы непрерывной интеграции *Travis CI* и *AppVeyor CI* для автоматической сборки проекта и запуска тестов при каждом изменении кодовой базы репозитория проекта. Однако сервера, на которых производится сборка могут не иметь графических процессоров, поэтому на этапе конфигурации проекта производится проверка наличия графических процессоров в системе.

Листинг 3. Проверка наличия графических процессоров в системе

if (WIN32)

execute\_process(COMMAND wmic path win32\_VideoController get name OUTPUT\_VARIABLE GPUS)

if (${GPUS} MATCHES "Name(.+\n)+")

message("At least one GPU exists, enabling GPU-dependent tests")

set(GPU\_EXISTS 1)

endif ()

else ()

execute\_process(COMMAND lspci OUTPUT\_VARIABLE GPUS)

if (${GPUS} MATCHES ".\*(VGA|3D|2D)+.\*")

message("At least one GPU exists, enabling GPU-dependent tests")

set(GPU\_EXISTS 1)

endif ()

endif (WIN32)

if (${GPU\_EXISTS} EQUAL 1)

add\_definitions("-DGPU\_EXISTS")

endif ()

Компилятору добавляется флаг *GPU\_EXISTS* с результатами проверки. В файле модульного теста проверяется наличие этого флага с помощью директивы препроцессора #*ifdef.*

### Система контроля версий

По стандартам программирования на *C++* рекомендуется использовать систему контроля версий [5, с. 20]. В проекте используется система контроля версий *Git*, разработанная Линусом Торвальдсом. При разработке используется модифицированная модель ветвления *Git* (*Git branching model*) [6]. Вся основная работа ведётся в ветке *dev*. Время от времени изменения с ветки *dev*, сливаются с веткой *testing*, на которой системы непрерывной интеграции проводят сборку и запуск тестов, чтобы убедиться, что проект всё ещё компилируется, добавленная функциональность работает на обеих целевых платформах и все тесты успешно завершаются. В случае успеха, изменения попадают на ветку *stable*, где им присваивается семантический номер версии. Таким образом на ветке *stable* всегда находится последняя стабильная функциональность проекта, которую можно использовать.

### Документация проекта

Для генерации документации по проекту используется система *Doxygen*, поддерживаемая большинством операционных систем. Как утверждается на веб-сайте проекта, *Doxygen* фактически является стандартом для создания документации в проектах, написанных на языках *C*, *C++* и *Fortran* [7].

Для генерации документации по исходному коду используются комментарии специального вида. Пример такого комментария приведён ниже в листинге 4.

Листинг 4. Часть комментария для генерации документации класса *Application*.

/\*!

\* \namespace Vulkalc

\* \brief Vulkalc namespace that contains all Vulkalc classes

\*

\* Vulkalc namespace that contains all Vulkalc classes.

\*/

namespace Vulkalc

{

/\*!

\* \class Application

\* \brief Vulkalc entry point class

\*

\* Application class is the entry point for Vulkalc library. It contains and gives access to all Vulkalc

\* classes and functions.

\*/

class VULKALC\_API Application

Специальные команды *Doxygen* позволяют форматировать документацию и выделять отдельные секции и блоки. Например, команда *\brief* задаёт короткое однострочное описание документируемой сущности, команда *\return* документирует возвращаемый из функции тип и его назначение. По умолчанию документация генерируется в формате *HTML*, однако можно включить генерацию документации в виде документов *PDF*, *RTF*, *LaTeX* и других.

С помощью *CMake* добавляется дополнительная цель сборки *doc*, выполнение которой вызывает *Doxygen* и генерирует документацию проекта.

Листинг 5. Фрагмент скрипта сборки документации.

add\_custom\_target(doc COMMAND ${DOXYGEN\_EXECUTABLE} ${DOC\_DIR}/GeneratedDoxyfile WORKING\_DIRECTORY ${ROOT\_DIR})

### Реализация проекта

* + 1. Экспорт объектов

Библиотека *Vulkalc* определяет пространство имён *Vulkalc*, в котором находятся все классы, экспортируемые библиотекой. Так как к библиотеке предъявлено требование кроссплатформенности, то необходимо было определить формат экспорта классов. У компилятора *GCC* модификатор доступа по умолчанию экспортирует все классы, структуры и функции. У компилятора *MSVC* нужно явно задавать экспортируемые объекты. В листинге 5 приведён фрагмент файла *Export.hpp*, в котором определён макрос для кроссплатформенного экспорта объектов.

Листинг 5. *Export.hpp*

#if defined(\_WIN32) || defined(\_\_CYGWIN\_\_)

#ifdef \_\_GNUC\_\_

#define VULKALC\_API \_\_attribute\_\_ ((dllexport))

#else

#define VULKALC\_API \_\_declspec( dllexport )

#endif

#define VULKALC\_LOCAL

#else

#if \_\_GNUC\_\_ >= 4

#define VULKALC\_API \_\_attribute\_\_ ((visibility ("default")))

#define VULKALC\_LOCAL \_\_attribute\_\_ ((visibility ("hidden")))

#else

#define VULKALC\_API

#define VULKALC\_LOCAL

#endif

#endif

* + 1. Инициализация и настройка *Application*

Согласно парадигме *RAII,* получение ресурса есть его инициализация. В соответствии с этой парадигмой, первичная инициализация полей класса происходит, когда создаётся объект класса *Application*, который является точкой входа во все функции библиотеки *Vulkalc*.

Листинг 6. Конструктор класса *Application*.

Application::Application() throw(HostMemoryAllocationException)

{

if (m\_isInitialized)

return;

m\_isInitialized = true;

m\_isLoggingEnabled = false;

m\_isErrorLoggingEnabled = false;

m\_spConfigurator = std::make\_shared<Configurator>();

m\_spShaderProvider = std::make\_shared<ShaderProvider>();

}

Далее нужно сконфигурировать объект класса *Application* с помощью объекта *Configuration*, полученного из объекта класса *Configurator*. Ссылка на объект класса *Configurator* хранится в объекте класса *Application* и доступна через вызов метода *getConfigurator*. После записи всех желаемых настроек в объект класса *Configuration*, нужно вызвать метод *configure* у объекта класса *Application*. В методе *configure* происходит первичная инициализация *Vulkan*.

Листинг 7. Конфигурация *Application*.

Application\* application = new Application();  
SharedConfiguration configuration = application->getConfigurator()->getConfiguration();  
configuration->isErrorLoggingEnabled = false;  
configuration->applicationName = "Vulkalc sample";  
configuration->logStream = cout;  
application->configure(false);

Основой приложения *Vulkan* является объект типа *VkInstance*. Так как *Vulkan* предоставляет *C API*, то все объекты являются структурами, ссылками на них или указателями, перечислениями и примитивными типами. Поэтому во все функции для создания объектов нужно передавать определённую структуру, содержащую какую-либо информацию о создаваемом объекте. Это также позволяет избежать перечисления большого количества параметров функций.

Для создания объекта *VkInstance* нужно заполнить структуру *VkApplicationInfo,* структуру *VkInstanceCreateInfo* со ссылкой на *VkApplicationInfo* и передать ссылку на неё в функцию *vkCreateInstance* вместе со ссылкой на сам объект. В листинге 8 структура заполняется информацией, полученной из объекта конфигурации Configuration.

Листинг 8. Создание объекта *VkInstance*.

auto configuration = m\_spConfigurator->getConfiguration();

m\_spLogStream = configuration->logStream;

m\_spErrorStream = configuration->errorStream;

m\_isLoggingEnabled = configuration->isLoggingEnabled;

m\_isErrorLoggingEnabled = configuration->isErrorLoggingEnabled;

//filling in VkApplicationInfo

\_prepareVulkanApplicationInfo();

if(m\_spVkApplicationInfo == nullptr)

throw HostMemoryAllocationException("Failed to allocate memory for VkApplicationInfo");

\_checkAvailableExtensions();

//filling in VkInstanceCreateInfo

\_prepareVulkanInstanceInfo();

if(m\_spVkInstanceCreateInfo == nullptr)

throw HostMemoryAllocationException("Failed to allocate memory for VkInstanceCreateInfo");

//creating VkInstance

VkResult result;

result = vkCreateInstance(m\_spVkInstanceCreateInfo.get(), nullptr, &m\_VkInstance);

if(result != VK\_SUCCESS)

throw VulkanOperationException("Failed to create VkInstance");

В листинге 8 вызываются приватные методы *\_prepareVulkanInstanceInfo* и *\_prepareVulkanApplicationInfo*, в которых и происходит заполнение структур данными.

Для всех операций с устройствами, такими как запуск шейдера, выделение памяти, используется объект логического устройства *VkDevice*, чтобы создать который нужно сначала выбрать физическое устройство, которое будет использоваться. Для этого в Vulkan API есть метод *vkEnumeratePhysicalDevices*. Так как интегрированные графические процессоры не подходят для вычислений, необходимо проверить тип найденных физических устройств. Для этого нужно получить *VkPhysicalDeviceProperties* с помощью функции *vkGetPhysicalDeviceProperties* и проверить поле *deviceType*. Если физическое устройство является дискретным, то это поле примет значение *VK\_PHYSICAL\_DEVICE\_TYPE\_DISCRETE\_GPU.* В листинге 9 продемонстрирован фрагмент метода *enumeratePhysicalDevices*, ответственный за получение списка физических устройств.

Листинг 9. Фрагмент метода *enumeratePhysicalDevices*.

std::vector<SharedPhysicalDevice> physicalDevices;

uint32\_t deviceCount = 0;

vkEnumeratePhysicalDevices(m\_VkInstance, &deviceCount, nullptr);

if(deviceCount == 0)

return physicalDevices;

Выбранное физическое устройство передаётся в метод *setPhysicalDevice,* в котором продолжается конфигурирование объекта Application и создаётся объект VkDevice. Чтобы создать объект VkDevice, нужно выбрать одно из семейств очередей, поддерживаемых устройством. Некоторые семейства поддерживают очереди вычислительных конвейеров, другие нет. Однако, как минимум одна очередь с поддержкой очередей для вычислительных конвейеров гарантированно существует, так как устройства без поддержки вычислительных шейдеров не поддерживаются технологией *Vulkan* [6, c. 42].

* + 1. Компиляция и загрузка шейдеров

После того, как объект *Application* был сконфигурирован, а физическое устройство для использования было выбрано, настаёт этап подготовки шейдеров. В библиотеке *Vulkalc* доступ к шейдерам предоставляет класс *ShaderProvider*, ссылка на объект которого хранится в объекте *Application*. Сначала шейдеры нужно обнаружить. Для этого используется метод *loadShaders* класса *ShaderProvider*, которому передаётся путь, в котором искать шейдеры. Этот метод требует, чтобы расширение файла было “*.comp*”, что соответствует расширению файла вычислительного шейдера *GLSL*. Так как к библиотеке предъявлено требование кроссплатформенности, то некоторые функции необходимо было реализовывать отдельно для платформ *Windows* и *Linux*. Такой функцией является функция *loadShaders*, которой нужно работать с директорией и читать её содержимое, однако в *C++* нет кроссплатформенного объекта для работы с директориями. Листинг 10 содержит определение функции *loadShaders*.

Листинг 10. Определение функции *loadShaders.*

std::vector<std::string> shaderNames;  
#ifdef \_MSC\_VER  
 std::stringstream ss;  
 *//listing file names in directory to file* ss << "dir \"" << directory << "\" /b > files";  
 std::string command;  
 getline(ss, command);  
 std::system(command.c\_str());  
 *//reading files* std::ifstream file = std::ifstream("files");  
 std::string buf;  
 while (!file.eof())  
 {  
 std::getline(file, buf);  
 if(checkFileNameExtension(buf, ".comp"))  
 shaderNames.push\_back(buf);  
 }  
 file.close();  
 std::system("del test");  
#else  
 DIR\* d;  
 struct dirent\* dir;  
 d = opendir(directory);  
 if(d)  
 {  
 while((dir = readdir(d)) != NULL)  
 {  
 std::string fileName = dir->d\_name;  
 if(checkFileNameExtension(fileName, ".comp"))  
 shaderNames.push\_back(fileName);  
 }  
  
 closedir(d);  
 }  
#endif  
 return shaderNames;

В зависимости от операционной системы, компилятор будет включать в тело метода различные реализации, проверяя препроцессорные константы.

Метод *loadShaders* возвращает вектор объектов класса *Shader*, которые дальше нужно передать в метод *tryCompileShaders*. Этот метод полагается на то, что путь до директории с утилитами *Vulkan* добавлен в системную переменную *PATH*. Он вызывает утилиту *glslangValidator* с ключом *-V*, чтобы скомпилировать *Vulkan*-совместимый шейдерный байт-код. В случае успешной компиляции, файл считывается в буфер, а затем создаётся объект *VkShaderModule*, которому передаётся размер загруженного шейдера и ссылка на начало буфера с шейдерным байт-кодом. Листинг 11 содержит фрагмент кода метода *tryCompile*, ответственного за компиляцию шейдера и создание модуля шейдера *Vulkan*.

Листинг 11. Фрагмент кода метода *tryCompile*.

std::string compileCommand =  
 "glslangValidator -V " + m\_shader.getShaderFullName() + " " + m\_shader.getShaderFullName() + ".spv";  
std::system(compileCommand.c\_str());  
std::ifstream shaderFile = std::ifstream(m\_shader.getShaderFullName(), std::ios::binary | std::ios::in | std::ios::ate);  
if(!shaderFile.is\_open())  
{  
 m\_isCompiled = false;  
 return m\_isCompiled;  
}  
size\_t size = shaderFile.tellg();  
shaderFile.seekg(0, std::ios::beg);  
char\* code = new char[size];  
shaderFile.read(code, size);  
shaderFile.close();  
VkShaderModuleCreateInfo createInfo;  
createInfo.sType = *VK\_STRUCTURE\_TYPE\_SHADER\_MODULE\_CREATE\_INFO*;  
createInfo.pNext = nullptr;  
createInfo.flags = 0;  
createInfo.codeSize = size;  
createInfo.pCode = (uint32\_t\*) code;  
VkShaderModule module;  
VkResult result = vkCreateShaderModule(\*(device->getVkDevice()), &createInfo, nullptr, &module);

Вызов метода *tryCompile* возвращает вектор объектов скомпилированных шейдеров *VerifiedShader*. Однако вектор может быть пустым, если не удалось скомпилировать шейдеры.

* + 1. Создание и запуск задачи на вычисление

Задача на вычисление представлена классом Task. Чтобы создать объект класса Task и запустить его на исполнение, нужно:

1. Получить объект класса *Runner* с помощью метода *getRunner* у объекта класса *Application*;
2. Вызвать у объекта класса *Runner* метод *createTaskForShader* и передать в него нужный объект скомпилированного шейдера *VerifiedShader*;
3. Задать буфер данных с помощью метода *setData*, в который передаётся структура, содержащая буфера входных данных;
4. Отправить задачу в очередь с помощью метода *queueTask* объекта класса Runner, передав в него объект задачи *Task*.

При вызове метода *queueTask* происходит основная работа по созданию объектов *Vulkan* и их настройке. Для того, чтобы запустить шейдер на исполнение, требуется выполнить шаги ниже в указанном порядке:

1. Создать объект *VkDescriptorPool* – пул дескрипторов;
2. Создать объект *VkDescriptorSet* – набор дескрипторов, задаёт вид и форму разметок соответствий между буферами в приложении и буферами в шейдере на устройстве;
3. Определить наличие необходимого размера памяти на устройстве с помощью структуры *VkDeviceMemoryProperties*;
4. Создать необходимые объекты *VkDeviceMemory* – выделенная память на устройстве, ещё ни к чему не привязанная;
5. Создать необходимые буферы *VkBuffer* и связать их с выделенной памятью методом *vkBindBufferMemory*;
6. Создать необходимое количество объектов *VkWriteDescriptorSet* – описывает вид дескриптора и связывает буферы на устройстве с переменными в шейдере;
7. Обновить информацию о разметке буферов на устройстве вызовом *vkUpdateDescriptorSets*;
8. Создать объект *VkCommandPool* – пул команд устройству;
9. Создать буфер команд *VkCommandBuffer*;
10. Переключить командный буфер в режим записи вызовом *vkBeginCommandBuffer*;
11. Привязать ранее созданный вычислительный конвейер к созданному буферу команд;
12. Связать командный буфер, дескрипторы и разметку конвейера вызовом *vkCmdBindDescriptorSets*;
13. Переключить командный буфер в режим готовности к отправке вызовом *vkCmdDispatch*;
14. Отправить буфер в очередь вызовом *vkQueueSubmit*.

В целях демонстрации класс Task был реализован с возможностью запуска только одного шейдера – сложение матриц.

Объекты *VkDescriptorSet* и *VkWriteDescriptorSet* задают соответствие между куском памяти на устройстве и памятью, используемой переменной в шейдере, как показано на рисунке 4.1.

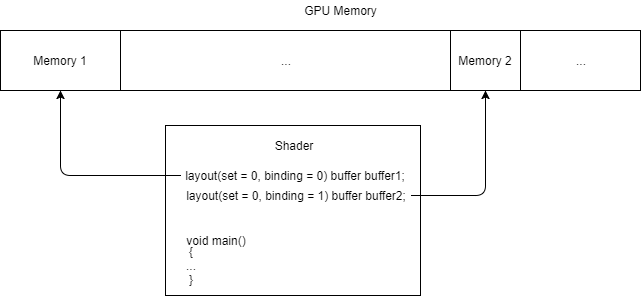


Рис. 4.1. Соответствие между памятью устройства и переменной шейдера

В листинге 12 содержится фрагмент кода метода queue, в котором задаётся это соответствие.

Листинг 12. Фрагмент кода метода queue.

VkWriteDescriptorSet writeDescriptorSets[3] = {

{VK\_STRUCTURE\_TYPE\_WRITE\_DESCRIPTOR\_SET, 0, descriptorSet, 0, 0, 1,

VK\_DESCRIPTOR\_TYPE\_STORAGE\_BUFFER, 0, &input1BufferInfo, 0},

{VK\_STRUCTURE\_TYPE\_WRITE\_DESCRIPTOR\_SET, 0, descriptorSet, 1, 0, 1,

VK\_DESCRIPTOR\_TYPE\_STORAGE\_BUFFER, 0, &input2BufferInfo, 0},

{VK\_STRUCTURE\_TYPE\_WRITE\_DESCRIPTOR\_SET, 0, descriptorSet, 2, 0, 1,

VK\_DESCRIPTOR\_TYPE\_STORAGE\_BUFFER, 0, &outputBufferInfo, 0}

};

vkUpdateDescriptorSets(\*(task->\_getDevice()->getVkDevice()), 3, writeDescriptorSets, 0, 0);

Запуск командного буфера с записанными командами на исполнение продемонстрирован в листинге 13.

Листинг 13. Запуск командного буфера на исполнение

result = vkBeginCommandBuffer(commandBuffer, &commandBufferBeginInfo);

if(result != VK\_SUCCESS)

throw Exception("Failed to begin command buffer");

//pushing size of array to shader

vkCmdPushConstants(commandBuffer, \*(task->\_getPipelineLayout()), VK\_SHADER\_STAGE\_COMPUTE\_BIT, 0, sizeof(uint32\_t), &bufferLength);

vkCmdBindPipeline(commandBuffer, VK\_PIPELINE\_BIND\_POINT\_COMPUTE, \*(task->\_getPipeline()));

vkCmdBindDescriptorSets(commandBuffer, VK\_PIPELINE\_BIND\_POINT\_COMPUTE, \*(task->\_getPipelineLayout()), 0, 1, &descriptorSet, 0, 0);

vkCmdDispatch(commandBuffer, 1, 1, 1);

VkSubmitInfo submitInfo = {VK\_STRUCTURE\_TYPE\_SUBMIT\_INFO, 0, 0, 0, 0, 1, &commandBuffer, 0, 0};

vkQueueSubmit(\*m\_spQueue, 1, &submitInfo, 0);

Стоит обратить внимание на вызов *vkCmdDispatch*. Согласно документации, 2, 3 и 4 параметры задают количество локальных рабочих групп по осям X, Y и Z, таким образом создавая абстрактную трёхмерную сеть из локальных рабочих групп. Шейдер будет выполняться потенциально параллельно для каждой из этих рабочих групп [3, c. 165].

* + 1. Получение результатов вычисления

Получить результат можно вызвав метод *getLastTaskResult* у объекта класса *Runner*. Стоит отметить, что этот метод будет ждать, пока устройство не вернёт результат, поэтому вызов этого метода может заблокировать управление, если в очереди много задач. Вызов этого метода скопирует буфер с результатами из памяти устройствами и вернёт его в объекте *TaskResult*. В листинге 14 представлено тело метода *getLastTaskResult*, который считывает данные с устройства.

Листинг 14. Фрагмент кода метода *getLastTaskResult*.

vkQueueWaitIdle(\*m\_spQueue);

TaskResult taskResult;

int32\_t \*payload;

uint32\_t memorySize = m\_spLastTask->\_getTaskBuffers().buffer\_size \* sizeof(int32\_t);

VkResult result = vkMapMemory(\*(m\_spLastTask->\_getDevice()->getVkDevice()), \*m\_spLastTaskOutMemory, 0, memorySize / 3, 0, (void \*\*) &payload);

if(result != VK\_SUCCESS)

throw Exception("Failed to map output device memory");

memcpy(taskResult.out\_buffer, payload, memorySize);

taskResult.buffer\_size = m\_spLastTask->\_getTaskBuffers().buffer\_size;

return taskResult;

## Тестирование

Для тестирования функционала библиотеки *Vukalc* применяются модульные тесты. Для языка *С++* существует множество фреймворков и библиотек модульного тестирования, однако выбор остановился на фреймворке *Catch*, так как он поставляется в виде одного заголовочного файла, что удовлетворяет требованию как можно меньшего числа зависимостей.

### 5.1. Модель исполнения тестов фреймворка Catch

Каждый сценарий тестирования задаётся макросом *TEST\_CASE* с описанием. Внутри одного сценария тестирования может содержаться 0 и более секций *SECTION*. Каждая из этих секций выполняется независимо от других секций и соответственно имеет своё виртуальное адресное пространство и стек. Таким образом внутри одного сценария тестирования можно протестировать несколько различных случаев поведения или различных наборов входных данных [8].

Для тестирования различных условий используются следующие макросы:

* *REQUIRE* – требование верности выражения в скобках;
* *REQUIRE\_FALSE* – требование неверности выражения в скобках;
* *REQUIRE\_NOTHROW* – требование выполнения выражения в скобках без исключений;
* *REQUIRE\_THROW\_AS* – требует, чтобы выражение в скобках при выполнении бросало исключение указанного типа

Существуют аналоги вышеуказанных макросам вида *CHECK* и т.д. Если макросы *REQUIRE* не выполняются, то выполнение тестов сразу останавливается, в отличии от макросов *CHECK*, которые не прерывают выполнения тестов.

### 5.2. Модульные тесты библиотеки Vulkalc

При разработке необходимо тестировать реализуемый в библиотеке функционал. Так как в случае с библиотекой это сделать напрямую невозможно, очень удобно тестировать с помощью модульных тестов. Библиотека Vulkalc содержит в себе тесты для основного функционала класса Application и его частей, а также исключений и вспомогательных функций. Листинг 15 содержит в себе сценарий тестирования “Создание устройства” модульного теста класса *Application.*

Листинг 15. Сценарий тестирования “Создание устройства”.

TEST\_CASE("Creating Device")

{

Application\* application = new Application();

SharedConfiguration configuration = application->getConfigurator()->getConfiguration();

shared\_ptr<stringstream> ss = make\_shared<stringstream>();

configuration->logStream = ss;

configuration->errorStream = ss;

application->configure(false);

std::vector<SharedPhysicalDevice> devices = application->enumeratePhysicalDevices();

SECTION("At least one GPU exists")

{

REQUIRE(devices.size() != 0);

}

SECTION("Setting PhysicalDevice continues configuring")

{

REQUIRE\_NOTHROW(application->setPhysicalDevice(devices[0]));

REQUIRE(application->getDevice() != nullptr);

}

SECTION("PhysicalDevice cannot be changed once set")

{

REQUIRE\_NOTHROW(application->setPhysicalDevice(devices[0]));

REQUIRE(application->getDevice() != nullptr);

REQUIRE\_THROWS\_AS(application->setPhysicalDevice(0), Exception);

REQUIRE\_THROWS\_AS(application->setPhysicalDevice(devices[0]), Exception);

}

SECTION("Runner and ShaderProvider are null before setting PhysicalDevice")

{

REQUIRE(application->getRunner() == nullptr);

REQUIRE(application->getShaderProvider() == nullptr);

}

SECTION("Runner is created after PhysicalDevice is set")

{

application->setPhysicalDevice(devices[0]);

REQUIRE(application->getRunner() != nullptr);

REQUIRE(application->getShaderProvider() != nullptr);

}

SECTION("Device is created after setting VkPhysicalDevice")

{

application->setPhysicalDevice(devices[0]);

REQUIRE(application->getDevice() != nullptr);

REQUIRE(application->getDevice()->getVkDevice() != nullptr);

}

devices.clear();

ss.reset();

configuration.reset();

delete application;

}

На рисунке 5.1 продемонстрирован результат запуска модульных тестов из среды разработки *CLion*, скомпилированных с помощью *MinGW g++*.



Рис. 5.1. Результат запуска тестов, скомпилированных *MinGW g++*.

На рисунке продемонстрирован результат запуска модульных тестов, скомпилированных с помощью *Microsoft Visual Studio 2015 x64*.

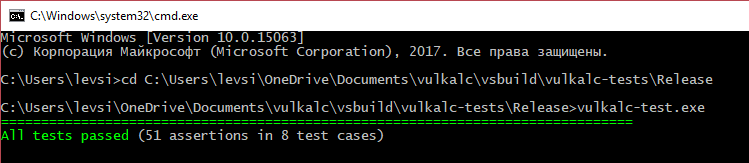


Рис. 5.2. Результат запуска тестов, скомпилированных *MS Visual Studio 2015 x64.*

### 5.3 Тестирование функций Vulkalc

Листинг 16 содержит исходный код демонстрационного проекта. На рисунке 5.3 продемонстрирован результат запуска демонстрационного проекта в среде *CLion*.

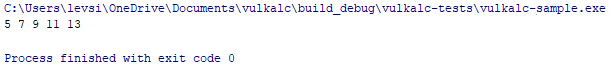


Рис 5.3. Результаты запуска демонстрационного проекта

Листинг 16. Исходный код демонстрационного проекта.

#include <Application.hpp>

using namespace Vulkalc;

using namespace std;

int main()

{

//creating and configuring Application

Application\* application = new Application();

SharedConfiguration configuration = application->getConfigurator()->getConfiguration();

configuration->isErrorLoggingEnabled = false;

configuration->applicationName = "Vulkalc sample";

shared\_ptr<stringstream> ss = make\_shared<stringstream>();

configuration->logStream = ss;

application->configure(false);

//choosing device

auto devices = application->enumeratePhysicalDevices();

application->setPhysicalDevice(devices[0]);

//compiling shaders

SharedShaderProvider shaderProvider = application->getShaderProvider();

vector<Shader> shaders = shaderProvider->loadShaders("../shaders/");

vector<VerifiedShader> verifiedShaders = shaderProvider->tryCompileShaders(shaders);

SharedTask task = make\_shared<Task>(application->getRunner()->createTaskForShader(verifiedShaders[0]));

//settings data

Task::TaskBuffers buffer;

buffer.buffer\_size = 5;

int32\_t array1[5] = {0, 1, 2, 3, 4};

int32\_t array2[5] = {5, 6, 7, 8, 9};

buffer.in\_buffer1 = array1;

buffer.in\_buffer2 = array2;

task->setData(buffer);

//running task

application->getRunner()->queueTask(task);

//getting task

TaskResult result = application->getRunner()->getLastTaskResult();

for(uint32\_t i = 0; i < result.buffer\_size; ++i)

cout << result.out\_buffer[i] << " ";

cout << endl;

devices.clear();

configuration.reset();

delete application;

return 0;

}

## Заключение

Для быстрого прототипирования вычислительных программ библиотека Vulkan в исходном виде не подходит. Слишком много рутинных задач нужно сделать, чтобы вычислить что-либо на графическом устройстве. Однако если предоставить удобный интерфейс, который прятал бы все эти рутинные задачи, то библиотеку *Vulkan* вполне можно использовать без особых трудозатрат. Стоит учитывать, что не все алгоритмы хорошо работают на графических процессорах из-за невозможности их распараллеливания.

Разработанная библиотека *Vulkalc* показала, что можно спрятать большинство вызовов Vulkan API и предоставить относительно удобный интерфейс для запуска вычислительных задач на графическом процессоре. Так как библиотека Vulkalc представляет собой демонстрацию работы с Vulkan API и предоставления интерфейса к его вызовам, она нуждается в доработке для повседневного использования. Также разработанная библиотека позволяет запускать только один демонстрационный шейдер с соответствующим ему расположению буферов в памяти и данными.

Направления развития библиотеки Vulkalc:

* Добавить возможность запуска шейдеров с произвольными данными и их расположением;
* Реализовать распараллеливание шейдера для увеличения производительности;
* Оптимизировать алгоритм выставления задач в очередь – для каждой задачи создаётся свой конвейер, пул команд, дескриптор. Для запуска одинаковых шейдеров часть этих структур можно не уничтожать;
* Добавить возможность задавать функции обратного вызова для слоёв валидации;
* Провести сравнение производительности вычислений на основе Vulkan, CUDA и OpenCL.

Так как к проекту предъявляется требование возможности свободного доступа к исходному коду, весь проект размещён в сети Интернет в свободном доступе под лицензией MIT по адресу <https://github.com/ToxikCoder/vulkalc>.

## Список использованных источников

1. Getting Started with the Vulkan SDK [Электронный ресурс] // LunarXchange. – Режим доступа: <https://vulkan.lunarg.com/doc/sdk/1.0.39.1/windows/getting_started.html>, свободный. - Заглавие с экрана. – Яз. англ. – (Дата обращения 20.02.2017).
2. Development Environment [Электронный ресурс] // Vulkan Tutorial. – Режим доступа: <https://vulkan-tutorial.com/Development_environment>, свободный. - Заглавие с экрана. – Яз. англ. – (Дата обращения 10.04.2017).
3. Vulkan API Reference Pages [Электронный ресурс] // Khronos.org. Connecting Software to Silicon. – Режим доступа: <https://www.khronos.org/registry/vulkan/specs/1.0/apispec.html>, свободный. - Заглавие с экрана. – Яз. англ. – (Дата обращения 17.04.2017).
4. SPIR-V Toolchain [Электронный ресурс] // LunarXchange. – Режим доступа: <https://vulkan.lunarg.com/doc/sdk/1.0.39.1/windows/spirv_toolchain.html>, свободный. - Заглавие с экрана. – Яз. англ. – (Дата обращения 08.03.2017).
5. Саттер, Герб, Александреску, Андрей. Стандарты программирования на С++ [Текст] : Пер. с англ. – М. : ООО “И.Д. Вильямс”, 2008. – 224 с.: ил. – Парал. тит. англ.
6. A successful Git branching model [Электронный ресурс] // Nvie.com. Thoughts and writings by Vincent Driessen. – Режим доступа: <http://nvie.com/posts/a-successful-git-branching-model/> , свободный. - Заглавие с экрана. – Яз. англ. – (Дата обращения 10.04.2017).
7. Doxygen [Электронный ресурс] // Doxygen.org. – Режим доступа: <http://www.doxygen.org/>, свободный. - Заглавие с экрана. – Яз. англ. – (Дата обращения 20.05.2017).
8. Getting Catch [Электронный ресурс] // Github.com. – Режим доступа: <https://github.com/philsquared/Catch/blob/master/docs/tutorial.md#test-cases-and-sections>, свободный. – Заглавие с экрана. – Яз. англ. – (Дата обращения 08.06.2017).