**Министерство образования и науки Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение**
**высшего образования «Московский государственный технический университет**
**имени Н.Э. Баумана (национальный исследовательский университет)»**
**(МГТУ им. Н. Э. Баумана)**

---

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Теоретическая информатика и компьютерные технологии» (ИУ9)

# РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ НА ТЕМУ:

## *«Оптимизирующие преобразования для формата SPIR-V»*

Студент группы ИУ9-82 _____ (Подпись, дата) А.А. Олохтонов

Руководитель ВКР _____ (Подпись, дата) А. В. Синявин

Нормоконтролер _____ (Подпись, дата) _____ (И. О. Фамилия)

2019 г.

# АННОТАЦИЯ

Объектом разработки является бинарный формат шейдеров SPIR-V.

Цель работы — разработка оптимизатора для формата SPIR-V с использованием библиотеки, реализованной в рамках курсовой работы.

Поставленная цель достигается за счет модернизации библиотеки и реализации на ее основе трех оптимизаций: подъема инвариантного кода из циклов, распространения и свертки констант и удаления мертвого кода. Для проверки корректности оптимизаций реализовано простейшее графическое приложение с использованием Vulkan API.

В работе показывается, что реализованный оптимизатор порождает более компактные бинарные файлы, чем утилита spirv-opt, разрабатываемая Khronos Group.

Объем данной дипломной работы составляет 77 страниц. Для ее написания было использовано 9 источников. В работе содержатся 75 листингов и 3 таблицы.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

На сегодняшний день существует множество конкурирующих программных интерфейсов для графического программирования: DirectX, OpenGL, Metal, Mantle, предоставляющих не только различные наборы функций для обращения к драйверу графического ускорителя, но и использующих собственные языки и промежуточные представления шейдеров [1, 2]. Так, только перечисленные спецификации используют языки GLSL, HLSL, C++ и AMD IL. Более того, некоторые языки обязывают каждого поставщика драйвера реализовывать собственный компилятор.

Графический интерфейс Vulkan создавался Khronos Group как «новое поколение OpenGL», и при его разработке преследовались следующие цели:

1. уменьшение накладных расходов (англ: overhead);
2. поддержка графических вызовов, совершаемых с нескольких потоков;
3. снижение нагрузки на центральный процессор [3].

Еще одной ключевой особенностью Vulkan является использование промежуточного двоичного формата шейдеров SPIR-V, в который компилируются все популярные языки шейдеров высокого уровня.

Официальный репозиторий Khronos Group содержит ряд утилит для работы с форматом SPIR-V, таких как компилятор из GLSL в SPIR-V, валидатор, дизассемблер, линкер, визуализатор потока управления и оптимизатор [4]. Именно на последнем сосредоточено внимание настоящей дипломной работы. Как оказалось, функции предоставляемой утилиты spirv-opt весьма ограничены: трудности составило даже написание такого шейдера, чтобы входные и выходные данные оптимизатора не были идентичны.

В ходе выполнения курсовой работы в седьмом семестре была разработана библиотека, предоставляющая функции для упрощения написания оптими-

зирующих преобразований формата SPIR-V. Эта библиотека повсеместно использовалась при написании данной дипломной работы.

Целью настоящей работы является разработка оптимизирующего преобразователя для формата SPIR-V, а также простейшего графического приложения, использующего Vulkan API, необходимого для проверки корректности произведенных преобразований. Задачи, таким образом, включают: изучение предметной области оптимизирующих преобразований, выбор конкретных оптимизаций и их реализация, изучение графического API Vulkan и реализация простейшего графического приложения, проверку корректности реализованных оптимизаций, модификацию и доработку разработанной в рамках курсовой работы библиотеки.

Практической значимостью данная работа обладает по целому ряду причин. Так, встраивание оптимизатора в рабочий процесс позволяет не только увеличить потенциальную производительность конечного продукта, но и сокращает занимаемое на диске место, что положительно влияет на длительность цикла «внесение изменений — компиляция — тестирование». Также стоит заметить, что если реализованный оптимизатор и не будет широко применяться, само его существование может повлечь за собой более активную разработку официального оптимизатора Khronos Group.

# 1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

## 1.1 Формат SPIR-V

SPIR-V — это простой двоичный промежуточный язык для графических шейдеров и вычислительных ядер [5]. Модуль SPIR-V содержит, кроме прочего, набор функций, каждая из которых состоит из списка базовых блоков, а каждый базовый блок содержит упорядоченный список инструкций.

Для доступа к объявленным переменным используются инструкции чтения и записи, а промежуточные результаты хранятся с использованием SSA формы [6]. Объекты представляются логически, с использованием иерархии типов. Выбор модели адресации устанавливает, могут ли использоваться операции с указателями или доступ к памяти является чисто логическим.

Инструкции в формате SPIR-V представляются линейной последовательностью четырехбайтовых слов, за исключением первых пяти слов, которые содержат заголовок модуля, формат которого представлен в таблице 1.

Таблица 1: формат заголовка SPIR-V модуля

| Номер слова | Содержание |
|---|---|
| 0 | Магическое число |
| 1 | Номер версии спецификации SPIR-V |
| 2 | Магическое число генератора. Используется для идентификации компилятора SPIR-V |
| 3 | Верхняя граница — число, гарантирующее, что все идентификаторы в этом модуле меньше этого числа |
| 4 | Нулевой байт |

Каждая инструкция содержит в нулевом слове два числа: старшие 16 бит хранят длину данной инструкции в словах (с учетом нулевого слова), а младшие 16 бит — число, идентифицирующее тип данной инструкции — так называе-

мый опкод (англ: opcode). Содержание остальных слов инструкции (как и их количество) варьируется от одного опкода к другому, однако для инструкций, пораждающих некоторый результат, чаще всего первое и второе слова содержат уникальный идентификатор результата (англ: result id) и тип этого результата, соответственно. Модуль SPIR-V находится в SSA форме в том смысле, что любой result id порождается лишь одной инструкцией.

Порядок инструкций внутри корректного модуля SPIR-V обязан быть следующим:

1. все инструкции OpCapability;
2. опциональные инструкции OpExtension;
3. опциональные инструкции OpExtInstImport;
4. одна обязательная инструкция OpMemoryModel;
5. все объявления точек входа OpEntryPoint;
6. все объявления режимов выполнения OpExecutionMode или OpExecutionModeId;
7. отладочные инструкции OpString, OpSourceExtension, OpSource, OpSourceContinued, OpName и др.;
8. инструкции аннотаций (декорирущие инструкции OpDecorate, OpMemberDecorate и др.);
9. все объявления типов OpTypeXXX, объявления констант OpConstantXXX, объявления глобальных переменных OpVariable с классом хранения *не* OpFunction);
10. все прототипы функций: OpFunction, параметры OpFunctionParameter окончания функций OpFunctionEnd;

11. все функции: OpFunction, параметры, все базовые блоки функции, OpFunctionEnd.

Тела функций же удовлетворяют следующим условиям:

1. базовый блок всегда начинается с инструкции OpLabel;
2. базовый блок всегда заканчивается финальной инструкцией (инструкцией ветвления OpBranch или OpBrandConditional, инструкцией OpReturn или OpReturnValue, либо специальной инструкцией для завершения базового блока);
3. все переменные, объявленные в функции, должны иметь класс хранения Function в инструкции OpVariable, а сами инструкции располагаться до всех остальных инструкций функции. Таким образом, все переменные внутри функции являются глобальными.

## 1.2 Оптимизирующие преобразования

Оптимизирующими преобразованиями называют алгоритмы, принимающие на вход некоторое промежуточное представление программы и производящие семантически инвариантные преобразования, дающие на выходе более эффективную по некоторым критериям программу. Такими критериями могут быть время выполнения программы, размер программного кода, количество используемой программой памяти и т.д. Программы, выполняющие оптимизирующие преобразования, называют оптимизаторами.

Написание такого оптимизатора, который для любой программы породит самый оптимальный ее вариант, аналогично решению проблемы останова, и потому невозможно [7]. По этой причине, при написании оптимизаторов обычно выбирают некоторый набор оптимизирующих преобразований, и ограничиваются реализацией лишь этого набора.

## 1.3 SSA форма

SSA формой промежуточного представления называется такое его состояние, в котором каждой переменной значение присваивается лишь единожды. SSA форма широко используется при разработке оптимизирующих компиляторов, так как промежуточное предствление в SSA форме всегда имеет цепь «определение-использование» (DU-цепь, англ: definition-use) из одного элемента, что значительно упрощает многие алгоритмы. Среди компиляторов, использующих SSA форму в своем промежуточном представлении: LLVM, GCC, V8, PyPy, GoLang и многие другие.

При построении SSA формы переменным обычно присваиваются версии, так что присвоение значения переменной превращается в объявление новой версии данной переменной. Так, фрагмент программы представленный на листинге 1 будет в SSA форме иметь вид, представленный на листинге 2.

Листинг 1: фрагмент программы до построения SSA формы

```
y = 0
y = 1
x = y
```

Листинг 2: фрагмент программы после построения SSA формы

```
y_1 = 0
y_2 = 1
x = y_2
```

В случае, если в данной точке программы значение переменной зависит того, откуда было передано управление (то есть присваивание одной и той же переменной происходит в двух после ветвления потока управления), вводят так

называемую «фи-функцию», выбирающую нужное значение переменной в зависимости от того, откуда управление пришло в данную точку программы:

Листинг 3: фи-функция SSA формы

```
x_1 = 0
if (...) {
    x_2 = 1
}
y = phi(x_1, x_2)
```

Описанные далее оптимизации предполагают, что для входные данные уже находятся в SSA форме.

### 1.4 Подъем инвариантного кода из цикла

Такие инструкции, которые можно вынести из тела цикла, не изменив семантику программы, называют *инвариантными относительно цикла*. Подъем инвариантного кода из цикла (англ: loop-invatiant code motion, LICM) — оптимизирующее преобразование, нацеленное на автоматический поиск инвариантных инструкций и их вынос из тела цикла.

К примеру, рассмотрим следующий фрагмент кода на языке C:

Листинг 4: фрагмент программы до выноса инвариантных инструкций

```
i = 0;
do {
    a = 1;
    b = a * a;
    i = i + 1;
} while (i < 100);
```

Можно заметить, что присваивания первые две строки в теле цикла содержат инвариантные инструкции, а потому могут быть вынесены за пределы цикла, что позволит избежать их повторного выполнения.

Листинг 5: фрагмент программы после выноса инвариантных инструкций

```
i = 0;
a = 1;
b = a * a;
do {
    i = i + 1;
} while (i < 100);
```

Так как циклы часто являются критическими с точки зрения производительности фрагментами кода, описанное преобразование крайне желательно к реализации в любом оптимизаторе (стоит заметить, что на момент написания данной ВКР утилита spirv-opt не реализует данную оптимизацию).

Как и многие другие оптимизирующие преобразования, подъем инвариантного кода из цикла ценен не только сам по себе, но и за счет того, что упрощает работу других оптимизаций.

### 1.5 Распространение и свертка констант

В случае, если все операнды некоторого выражения константы, то возможно вычислить значение данного выражения в момент компиляции и заменить выражение на его значение. Оптимизация, находящая все такие вычислимые выражения называется *сверткой констант*.

*Распространением констант* называют оптимизирующие преобразования, осуществляющие поиск всех использований константных выражений и замену их использования на значение соответствующей константы.

Особенно ценны поочередные применения распространения и свертки

констант, так как на этапе распространения возможно появление новых вычислимых константных выражений, подлежащих свертке. После свертки же, возможно появление новых констант, подлежащих распространению.

Листинг 6: фрагмент программы до распространения и свертки констант

```
int x = 4;
int y = 5 - x / 2;
return y * (24 / x + 2);
```

Этап распространения заменяет использование переменной x на значение 4, после чего возникает новое выражение, подлежащее свертке. Этап свертки заменит выражение 5 - 4 / 2 на значение 3, после чего переменная уже переменная y подлежит распространению. Еще одно повторение этапов распространения и свертки, в итоге, преобразует код следующему виду:

Листинг 7: фрагмент программы после распространения и свертки констант

```
int x = 4;
int y = 3;
return 24;
```

Последующие оптимизации могут заметить, что инструкции присваивания не имеют эффекта (являются мертвым кодом) и удалить их.

При реализации данного преобразования также важно помнить, что фактически производится замена арифметических операций целевой платформы на операции той платформы, на которой будет запущен оптимизатор. Таким образом, особенного внимания заслуживают крайние случаи, такие как переполнение целочисленных типов, деление на ноль и т.п.

## 1.6 Исключение мертвого кода

Инструкции, никак не влияющие на результат работы программы, называются *мертвым кодом*. Существует два типа мертвого кода: недостижимый код — инструкции, которые никогда не выполнятся в ходе работы программы, и код, изменяющий только значения *мертвых переменных*. Мертвой называется переменная, в которую происходит запись, но не чтение. Здесь существенно требование о нахождении кода в SSA форме, так как до ее построения переменная может быть использована и на чтение, а на запись, однако две или более записей, не разделенных чтением, очевидно, пораждают мертвый код. В то же время, после построения SSA формы такие идущие подряд записи преобразуются в две или более версии SSA переменной, и лишь значение последней из них может быть использовано, предыдущие же версии являются мертвыми:

Листинг 8: фрагмент программы до удаления мертвого кода

```
a = 0
a = 1
b = a
```

До построения SSA формы может быть неочевидно, что первое присваивание переменной a является мертвым.

Листинг 9: фрагмент программы до удаления мертвого кода (SSA форма)

```
a_1 = 0
a_2 = 1
b = a_2
```

Здесь же явно видно, что переменная первая версия переменной a не используется, и может быть удалена.

Листинг 10: фрагмент программы после удаления мертвого кода (SSA форма)

```
a_2 = 1
b = a_2
```

В приведенном примере удалена всего одна инструкция, однако возможна ситуация, в которой значение, присвоенное мертвой переменной, требует ресурсоемких вычислений (например, в случае присваивания результата вызова некоторой функции). Особенно важную роль играет данное преобразование в случае его применения *после* остальных оптимизаций. Более того, некоторые оптимизации реализуются с расчетом на то, что позже будет произведена «чистка» при помощи удаления мертвого кода.

# 2 РАЗРАБОТКА

## 2.1 Структура программного комплекса

Разработку программного комплекса, составляющего настоящую дипломную работу, было решено разделить на три части:

1. библиотека для написания оптимизирующих преобразований. Реализованная в рамках курсовой работы и нуждается в доработке;
2. консольное приложение, осуществляющее оптимизирующие преобразования, и использующее библиотеку;
3. графическое приложение, использующее Vulkan API, позволяющее проверить корректность произведенных оптимизаций.

Таким образом, реализованный программный комплекс должен функционировать согласно следующим шагам:

1. составление кода шейдера на одном из языков высокого уровня;
2. компиляция шейдера в формат SPIR-V;
3. запуск графического приложения и наблюдение полученного изображения;
4. запуск оптимизатора, порождающий обновленный SPIR-V модуль;
5. наблюдение за графическим приложением на предмет изменений порождаемого изображения.

## 2.2 Библиотека

Библиотека для написания оптимизаций формата SPIR-V была реализована в рамках курсовой работы и предоставляет пользователю функции по конструированию промежуточного представления, модификации этого представ-

ления (в том числе графа потока управления) и сериализации промежуточного представления обратно в бинарный формат.

Так как полная спецификация формата SPIR-V содержит более 250 инструкций, структура библиотеки подразумевает поддержку лишь подмножества всего формата. Однако, с тем чтобы упростить доработку библиотеки, была предусмотрена возможность простого расширения множества поддерживаемых инструкций.

Более того, набор предоставляемых пользователю библиотеки функций составлялся практически «вслепую», так как разработка кода, использующего библиотеку, не входила в план курсовой работы.

Таким образом, доработка библиотеки в рамках дипломной работы включает:

1. расширение набора поддерживаемых инструкций, в том числе инструкций сравнения, инструкций объявления констант, инструкций объявления типов, а также более детализированное представление классов хранения и типов переменных;

2. добавление в интерфейс библиотеки функций получения result id и списка операндов инструкции, а также функции создания новой константы;

3. минимизацию зависимостей библиотеки, в том числе избавление от функций выделения памяти и записи в файл;

4. добавление возможности поставки в виде заголовочного и бинарного файлов для статической или динамической линковки.

Отдельно стоит отметить важность минимизации зависимостей библиотеки. Минимальный набор зависимостей позволяет достичь максимальной переносимости как между различными компиляторами, так и между версиями операционной системы или даже платформами.

Избавление от вызовов функции выделения памяти и записи в файл также крайне положительно сказывается на удобстве пользования библиотекой, так как оставляет пользователю возможность использования собственной нестандартной функции выделения памяти или записи в файл.

Для дальнейшего упрощения работы библиотеки была использована схема управления памятью, основанная на регионах (англ: memory arena), осуществляющая всего два вызова функций управления памятью, переданных пользователем: один на выделение региона памяти в начале работы программы, и второй на ее освобождение при завершении работы с библиотекой. Такая схема, при правильном ее использовании, дает пользователю гарантию, что библиотека либо не инициализируется вовсе, либо отработает без ошибок (работы с памятью).

## 2.3 Оптимизатор

Задачей работы оптимизатора является применение оптимизирующих преобразований к промежуточному представлению, составленному из валидного модуля SPIR-V. Существенным является требование валидности (корректности) принимаемого на вход бинарного файла: в соответствии с идеологией SPIR-V и Vulkan API в целом, библиотека не берет на себя ответственности по проверке действий пользователя на правильность. Гарантируется лишь корректность пораждаемого бинарного файла при выполнении условий корректности входного файла и манипуляций над промежуточным представлением, произведенных пользователем.

В рамках настоящей дипломной работы было решено реализовать три оптимизирующих преобразования: подъем инвариантного кода из циклов (англ: loop-invariant code motion), распространение и свертку констант (англ: constant propogation and folding), а также исключение (удаление) мертвого кода (англ: dead code elimination).

### 2.3.1 Loop-invariant code motion

Подъем инвариантного кода из цикла можно разбить на четыре этапа:

1. поиск циклов;
2. нахождение инвариантных инструкций;
3. проверка условий допустимости выноса инвариантных инструкций за пределы тела цикла;
4. подъем инструкций (англ: code motion).

Поиск циклов в общем случае использует дерево доминаторов и ищет так называемые *натуральные циклы*, однако в бинарный формат SPIR-V значительно упрощает эту задачу, так как содержит декларативные инструкции структурированного потока управления. Другими словами, любое ветвление потока управления программы объявляется в явном виде с указанием, в том числе, блоков слияния (англ: merge block), в которые поток управления сходится после цикла или условного ветвления. Таким образом, поиск циклов упрощается до поиска и разбора всех таких декларативных инструкций.

Нахождение инвариантных инструкций основывается на определении: инструкция инвариантна относительно цикла, если для каждого операнда этой инструкции выполняется хотя бы одно из условий:

1. операнд является константным;
2. все достигающие этого операнда определения находятся вне цикла;
3. существует единственное достигающее этого операнда определение, находящееся внутри цикла и являющееся, в свою очередь, инвариантным.

Алгоритм поиска инвариантных инструкций относительно цикла, таким образом, принимает следующий вид:

1. помечаем все инструкции в теле цикла как *не* инвариантные;
2. обходим все базовые блоки в цикле в порядке обхода в ширину, для каждого блока выполняем:
   (a) проходим по всем инструкциям блока, для каждого операнда инструкции проверяем, является ли он инвариантным;
   (b) если все операнды инструкции инвариантны, помечаем инструкцию как инвариантную;
3. повторяем обход до тех пор, пока не будет помечена как инвариантная ни одна новая инструкция.

Прежде чем вынести инвариантную инструкции из цикла необходимо проверить, не нарушает ли эта операция семантику программы. Возможно ситуации, в которых инструкция является инвариантной, однако вынесена из цикла быть не может.

Листинг 11: пример инвариантной инструкции, которую нельзя выносить из цикла

```
while (...) {
    if (...) {
        a = 2
    }
    b = a
}
```

Инструкция a = 2 является инвариантной, так как единственный ее операнд — константа. Однако вынос такой инструкции из тела цикла нарушит логику программы, так как в таком случае инструкция будет выполнена вне зависимости от результата работы оператора if. Сформулировать это условие можно

следующим образом: «вынос инструкции из цикла является корректным, только если определенная в инвариантной инструкции переменная доминирует над всеми своими использованиями». По той же причине нельзя выносить инструкции из тела цикла с предпроверкой условия, так как, в общем случае, условие выхода из цикла может не выполниться ни разу, а потому любые присваивания, произведенные внутри цикла не доминируют над соответствующими использованиями, расположенными после цикла:

Листинг 12: тело цикла с предпроверкой условия может не выполниться

```
a = 1
while (some_condition) {
    a = 2
    ...
}
b = a
```

Корректным, в общем случае, является следующий подход: цикл заменяется на цикл с постпроверкой условия и помещается внутрь условной конструкции, с условием совпадающим с первоначальным условием выхода из цикла. Таким образом, заведомо невыполнимые тела циклов не выполнятся даже после выноса инструкций из тела цикла, а количество вычислений условия остановки не изменится (что важно, если проверка условия выхода из цикла имеет побочные эффекты.

Листинг 13: преобразованная конструкция цикла с предпроверкой условия

```
a = 1
if (some_condition) {
    a = 2
    do {
```

```
        ...
    } while (some_condition)
}
b = a
```

В данной дипломной работы, для простоты, рассматриваются циклы с постпроверкой условия, не требующие никаких дополнительных действий.

Вторым, схожим условием допустимости выноса инструкции из тела цикла является условие доминирование вершины, содержащей инвариантную инструкцию, над всеми вершинами, содержащими выход из цикла. Ранний выход из цикла, возможен, например, при использовании оператора break в теле цикла.

Выполнение этих двух условий является достаточным, и инструкция может быть вынесена из цикла без нарушения семантики программы.

### 2.3.2 Constant propogation and folding

Распространение и свертка констант, вообще говоря, являются двумя разными оптимизациями, однако специфика устройства SPIR-V модуля в SSA форме позволяет просто их объединить. Общая версия алгоритма распространения выполняет абстрактную интерпретацию кода в SSA форме и использует решетку для хранения значений констант, отображая переменные SSA формы на элементы решетки [8].

В рассматриваемом же случае оптимизация значительно упрощается, так как все DU-цепи явно заданы при помощи result id, и большинство действий могут быть произведены непосредственно. Так, в начале рассчитывается для каждого result id множество инструкций, использующих этот данных result id. Распространение констант, таким образом, упрощается продвижения значений вдоль цепочек копирования (англ: copy propogation), а свертка констант требует

лишь объявления новой константы с последующей заменой вычисленного выражения на копирование значения константы (которое, в свою очередь, будет удалено на этапе распространения).

Множество типов и возможных их комбинаций в арифметических операциях формата SPIR-V объемно, а потому для простоты рассматривались и обрабатывались лишь 32-битные целочисленные числа и 32-битные числа с плавающей точкой. Правила арифметики с переполнением типов, строго говоря, являются в стандарте языка C неопределенным поведением, а потому в соответствие со спецификацией SPIR-V не приводились.

Оптимизации применяются в порядке «распространение-свертка» до тех пор, пока оба шага не стабилизируются.

### 2.3.3 Dead code elimination

Удаление мертвого кода в рассматриваемом случае также сильно отличается от канонической реализации. В стандартном алгоритме ищутся так называемые *критические* инструкции, которыми чаще всего являются инструкции ввода-вывода, а также инструкции возвращения значения из функции и подобные. Все инструкции, кроме критических, помечают как мертвые. После этого, все достигающие определения всех операндов живых инструкций помечаются как живые. Этот шаг повторяют до стабилизации множества живых инструкций.

В настоящей дипломной работе алгоритм реализован иначе: изначально все инструкции помечаются как живые. После этого все некритические инструкции, result id которых не используется, удаляются, а таблица использования result id обновляется. Этот шаг повторяется до тех пор, по множество живых инструкций не стабилизируется.

Наконец, все описанные оптимизирующие преобразования применяются по очереди (причем удаление мертвого кода в последнюю очередь), и, если хотя бы одна оптимизация совершила какие-то преобразования, то все три оптими-

зации запускаются заново.

## 2.4 Графическое приложение

С тем, чтобы убедиться в корректности пораждаемого оптимизатором кода, необходимо графическое приложение, которое позволит наблюдать в реальном времени изображение, получаемое применением переданного шейдера. Более конкретно, требования к приложению можно поставить следующим образом:

1. приложение должно использовать шейдер в формате SPIR-V;
2. приложение должно позволять по желанию пользователя без перезагрузки обновить один из шейдеров и пересобрать графический конвеер;
3. приложение должно выводить время, затраченное на отрисовку кадра.

Последнее требование установлено для того, чтобы оценить эффект от оптимизирующих преобразований.

# 3 РЕАЛИЗАЦИЯ

## 3.1 Модификация библиотеки

Как и библиотека, так и оптимизатор и графическое приложение были реализованы на языке C99. Использование минималистичного набора системных библиотек позволило добиться сборки как на ОС GNU/Linux с компиляторами gcc 8.3.0 и tcc 0.9.27, так и на ОС Windows 10 с компилятором cl версии 2017 года. Разработка велась поочередно на операционных системах Debian 10 и Windows 10.

Для добавления поддержки инструкций бинарных предикатов в перечисление opcode_t были добавлены опкоды инструкций сравнения.

Листинг 14: фрагмент перечисления opcode_t

```
enum opcode_t {
    ...
    ...
    OpIEqual = 170,
    OpINotEqual = 171,
    OpUGreaterThan = 172,
    OpSGreaterThan = 173,
    OpUGreaterThanEqual = 174,
    OpSGreaterThanEqual = 175,
    ...
}
```

Так как все инструкции сравнения в формате SPIR-V содержат одинаковые операнды, по аналогии с бинарными арифметическими операциями была создана обобщенная структура binary_comparison_layout, имеющая своими операндами четыре слова.

Листинг 15: обобщенная структура инструкций сравнения

```
struct binary_comparison_layout {
    u32 result_type;
    u32 result_id;
    u32 operand_1;
    u32 operand_2;
};
```

Полученная структура была добавлена в общую структуру инструкции SPIR-V/

Листинг 16: структура инструкции SPIR-V

```
struct instruction_t {
    enum opcode_t opcode;
    u32 *unparsed_words;
    u32 wordcount;
    union {
        ...
        struct opcopyobject_t OpCopyObject;
        struct unary_arithmetics_layout
            unary_arithmetics;
        struct binary_arithmetics_layout
            binary_arithmetics;
        struct binary_comparison_layout
            binary_comparison;
    };
};
```

Аналогично, были объявлены и добавлены инструкции объявления константы OpConstant и объявления целочисленного типа OpTypeInt и типа с плавающей точкой OpTypeFloat. Стоит заметить, что благодаря соответствию типов в спецификациях языка C и формата SPIR-V, значения OpConstant возможно сохранить в переменной нужного типа, что значительно упрощает организацию арифметических операций над константами.

Листинг 17: структура инструкции объявления константы

```
struct opconstant_t {
    u32 result_type;
    u32 result_id;
    union {
        s8 value_s8;
        u8 value_u8;
        s16 value_s16;
        u16 value_u16;
        f32 value_f32;
        s32 value_s32;
        u32 value_u32;
        f64 value_f64;
        s64 value_s64;
        u64 value_u64;
    };
};
```

Также, для упрощения определения типов в структуру промежуточного представления было добавлено поле types, предназначенное для установления соответствия между данным result id и его типом. Для отражения типа была создана структура datatype.

Листинг 18: структура, хранящая тип данных

```
struct datatype {
    enum supertype_t supertype;
    union {
        enum int_type_t int_type;
        enum float_type_t float_type;
    };
};
```

Здесь поле supertype определяет является ли число целым, либо с плавающей точкой. Перечисление int_type либо float_type указывает на конкретный тип, включая разрядность.

Для получения result id была реализована функция get_result_id, принимающая структуру инструкции, и возвращающая, в зависимости от опкода, либо значение операнда, содержащего result id, либо ноль, если данная инструкция не возвращает никакого значения. Для реализации использовалась конструкция switch.

Листинг 19: фрагмент функции get_result_id

```
switch (instruction->opcode) {
    case OpVariable:
        return(instruction->OpVariable.result_id);
    case OpLoad:
        return(instruction->OpLoad.result_id);
    ...
    default: return(0);
}
```

Аналогично, была реализована функция get_operands, возвращающая вектор операндов инструкции. В случае, если у инструкции нет операндов, возвращается пустой вектор. Важно заметить, что операндами здесь называются только result id других инструкций. То есть, например, строковые и численые литералы не возвращаются.

Листинг 20: фрагмент функции get_operands

```
switch(instruction->opcode) {
    case OpName: {
        vector_push(&result,
            instruction->OpName.target_id);
    } break;
    ...
    case OpPhi: {
        u32 operand_count =
            (instruction->wordcount - 3) / 2;
        vector_push(&result,
            instruction->OpPhi.result_type);
        for (u32 i = 0; i < operand_count; ++i) {
            vector_push(&result,
                instruction->OpPhi.variables[i]);
            vector_push(&result,
                instruction->OpPhi.parents[i]);
        }
    } break;
    ...
    default: {};
}
```

Наконец, была реализована функция ir_add_constant, добавляющая новую константу в SPIR-V модуль. Так как все константы, согласно спецификации, объявляются непосредственно перед функциями, функция ir_add_constant производит обход связного списка инструкций до тех пор, пока не будет найдена инструкция OpFunction, и вставляет новую инструкцию прямо перед объявлением функции.

Листинг 21: функция добавления новой константы

```
ir_add_constant(struct ir *module,
                struct instruction_t instruction)
{
    struct instruction_list *inst = module->pre_cfg;
    struct instruction_list *newinst
        = arena_push(sizeof(struct instruction_list));
    while (inst) {
        enum opcode_t opcode = inst->data.opcode;
        if (opcode == OpFunction) {
            newinst->data
                = ir_clone_instruction(instruction);
            newinst->next = inst;
            newinst->prev = inst->prev;
            inst->prev->next = newinst;
            inst->prev = newinst;
            break;
        }
        inst = inst->next;
    }
    return(newinst);
}
```

Прототипы описанных функций были добавлены в заголовочный файл библиотеки, таким образом делая их доступными для пользователя.

Для упрощения работы с памятью внутри библиотеки была реализована схема, основанная на регионах. А именно, был реализован простейший вариант с непрерывным фрагментом памяти фиксированного размера. Было реализовано три функции, осуществляющие инициализацию фрагмента, выделение памяти, и освобождение фрагмента. Чтобы добиться независимости от реализации функции выделения памяти, инициализирующая функция arena_init принимает указатель на функцию выделения памяти, к которой предъявляются следующие требования:

1. функция должна принимать один аргумент беззнакового целочисленного типа;
2. функция должна выделить непрерывный блок памяти длины, равной значению переданного аргумента, выровненный по размеру указателя в системе. Выделенная память также должна быть инициализирована нулями;
3. в случае успеха функция должна возвращать указатель на начало выделенного фрагмента;
4. в случае ошибки функция должна возвращать нулевой указатель.

Поставленных требований достаточно для реализации функций инициализации и выделения памяти arena_init и arena_push.

Для освобождения памяти также принимается указатель на функцию, поведение которой должно соответствовать поведению функции free.

Для избавления избавления от зависимости от функций манипуляции с памятью, таких как memset, memcmp, memcpy и strlen эти функции также были реализованы вручную. Таким образом, зависимости библиотеки удалось сократить до четырех заголовочных файлов:

1. stddef.h — для константы NULL (можно использовать собственную константу);

2. stdio.h — для вывода сообщений об ошибках (можно использовать аналог переменной ERRNO);

3. stdint.h — для типов фиксированных размеров;

4. stdbool.h — для типа bool и констант true и false (можно использовать собственный тип).

Листинг 22: инициализация простейшего менеджера памяти

```
void
arena_init(void *(*calloc) (u64), u64 size)
{
    void *memory = calloc(size);
    ASSERT(memory);
    arena.base = (u8 *) memory;
    arena.size = size;
    arena.used = 0;
}
```

Стоит заметить, что для обеспечения выравнивания адресов всех используемых в библиотеке сегментов памяти, необходимо вручную проверять размер выделяемого сегмента на предмет его кратности размеру указателя, и в случае отрицательности условия увеличивать размер выделяемого сегмента.

Превышение общего допустимого объема выделенной памяти считается за критическую ошибку и аварийно завершает работу библиотеки и использующей ее программы. В случае же успеха, функция вовзвращает указатель, который может быть использовать как и любой другой (за тем исключением, что

для него нельзя вызывать функцию free). За счет требований, накладываемых на изначально переданную функцию выделения памяти, любой выделенный фрагмент инициализирован нулями.

Листинг 23: выделение памяти в простейшем менеджере памяти

```
void *
arena_push(u64 size)
{
    if (size & 0x3L) {
        size = 8 * (size / 8 + 1);
    }
    ASSERT(arena.used + size <= arena.size);
    arena.used += size;
    return(arena.base + arena.used - size);
}
```

Одним из достоинств такого простого менеджера памяти является тот факт, что очистка памяти производится один раз в конце работы программы, и заведомо не оставляет утечек.

Листинг 24: очистка памяти в простейшем менеджере памяти

```
void
arena_free(void (*free) (void *))
{
    free(arena.base);
    arena.size = 0;
    arena.used = 0;
    arena.base = NULL;
}
```

Важные модификации библиотеки также касаются обработки входных и выходных параметров шейдера, предназначенных для получения и передачи данных от одного этапа графического конвеера к другому. Дело в том, что такие переменные доступны лишь по указателю, а потому не могут быть преобразованы в SSA форму в том виде, в котором это делается в библиотеке.

Для решения этой проблемы необходимо обрабатывать такие переменные отдельно, и восстанавливать соответствующие инструкции после построения SSA формы. Для всех переменных с классом хранения Input необходимо вставить инструкцию чтения данных по указателю OpLoad во входной блок функции, а для переменных класса Output — добавить инструкции OpStore записи последнего актуального значения по указателю во все терминальные блоки функции.

Входной блок для функции находить отдельно не нужно, так как он имеет в промежуточном представлении библиотеки нулевой индекс

Листинг 25: обработка переменных класса Input

```
if (original_variable->OpVariable.storage_class == Input
    && versions->size == 0) {
    ir_prepend_instruction(file->blocks + 0,
                           inst->data);
    ir_delete_instruction(file->blocks + block_index,
                          inst);
    deleted = true;
}
```

Для переменных класса Output поиск терминальных блоков также несложен: терминальными являются те, и только те базовые блоки, которые не содержат исходящих ребер.

Листинг 26: обработка переменных класса Output

```
if (orig_variable->OpVariable.storage_class == Output
    && is_termination_block) {
    struct instruction_t store;
    store.opcode = OpStore;
    store.wordcount = 3;
    store.unparsed_words = NULL;
    store.OpStore.pointer
        = orig_variable->OpVariable.result_id;
    store.OpStore.object
        = mapping->data[counter - 1];
    ir_append_instruction(file->blocks + block_index,
                          store);
}
```

Еще одной важной доработкой библиотеки стала обработка случая, в котором один базовый блок содержит несколько фи-функций SSA формы. Дело в том, что согласно спецификации SPIR-V все фи-функции должны быть расположены перед остальными инструкциями базового блока, однако в изначальной реализации прямо за фи-функцией вставлялась инструкция OpCopyObject. Для исправления этой ошибки вставка фи-функций была отложена до того момента, как будет найдены *все* фи-функции, которые необходимо вставить в данный базовый блок. После этого, с использованием функции ir_prepend_instruction сначала вставляются все инструкции OpCopyObject, а затем все инструкции OpPhi.

Наконец, последней модификацией библиотеки стало небольшое изменение кода функции ir_dump, осуществляющей сериализацию промежуточного представления. В изначальном варианте функции базовые блоки сериализовывались в порядке, в котором они были считаны и сохранены в промежуточном

представлении. Однако, спецификация SPIR-V требует, чтобы базовые блоки располагались в таком порядке, что для любого блока его непосредственный доминатор находится выше его самого.

Чтобы выполнить это условие, было решено использовать сериализовывать блоки в порядке обхода в ширину дерева доминаторов.

Листинг 27: сериализация базовых блоков

```
ir_dump(struct ir *file)
{
    ...
    struct uint_vector dom_bfs
        = cfg_bfs_order(&file->dominator_graph);
    ...
    for (u32 i = 0; i < file->cfg.labels.size; ++i) {
        u32 block_index = dom_bfs.data[i];
        ...
    }
    ...
}
```

### 3.2 Оптимизатор

Реализация оптимизатора состоит из вызовов библиотеки и поочередно применяемых оптимизирующих преобразований. Имена входных и выходных файлов считываются из аргументов командной строки. Входной файл будет загружен и преобразован в промежуточное представление, а в выходной будет записана оптимизированная версия шейдера. В первую выделяется память для работы библиотеки (так как ошибка на данном этапе подразумевает невозможность продолжения работы).

Листинг 28: инициализация региона памяти библиотеки

```
main(s32 argc, char **argv)
{
    if (argc != 3) {
        fprintf(stderr,
                "[ERROR] Usage: %s in out\n", argv[0]);
        return(1);
    }
    arena_init(&zero_alloc, 128 * 1024 * 1024);
    ...
}
```

Функция zero_alloc, передаваемая в вызов arena_init, удовлетворяет поставленным требованиям и использует директиву препроцессора #ifdef _MSC_VER для выбора между платформозависимыми функциями выделения памяти.

Листинг 29: функция выделения памяти

```
zero_alloc(u64 size)
{
    void *res;
#ifdef _MSC_VER
    res = _aligned_malloc(size, sizeof(void *));
#else
    posix_memalign(&res, sizeof(void *), size);
#endif
    memset(res, 0x00, size);
    return(res);
}
```

В случае, если выделение памяти прошло успешно, происходит загрузка бинарного файла с диска в память. Для этого используются вызовы fopen, fseek, rewind, fread и fclose.

Листинг 30: функция чтения бинарного файла с диска

```
static struct buffer { u32 *data; u32 words; }
get_binary(const char *filename)
{
    struct buffer result;
    FILE *file = fopen(filename, "rb");

    if (!file) {
        result.words = 0;
        result.data = NULL;
        return(result);
    }

    fseek(file, 0L, SEEK_END);
    result.words = ftell(file) / 4;
    rewind(file);

    result.data = arena_push(result.words * 4);
    fread((u8*) result.data, result.words * 4, 1, file);
    fclose(file);

    return(result);
}
```

Затем, вызываются библиотечные функции ir_eat и ssa_convert, осуществ-

ляющие конструирование промежуточного представления и преобразование в SSA форму, соответственно.

Листинг 31: конструирование IR и приведение его к SSA форме

```
struct ir module = ir_eat(source_file.data,
                          source_file.words);
ssa_convert(&module);
```

После этого последовательно вызываются функции, осуществляющие три описанных оптимизирующих преобразования.

Листинг 32: вызов оптимизирующих преобразований

```
loop_invariant_code_motion(&module);
constant_propogation_and_folding(&module);
dead_code_elimination(&module);
```

Наконец, преобразованная структура сериализуется и сохраняется на диск, а выделенная память освобождается.

Листинг 33: сброс на диск бинарного файла и освобождение памяти

```
{
    ...
    struct binary_buffer file = ir_dump(&module);
    FILE *output = fopen(argv[2], "wb");
    fwrite(file.data, file.words * 4, 1, output);
    fclose(output);
    arena_free(&free);
    return(0);
}
```

Функция loop_invariant_code_motion, осуществляющая подъем инвариантного кода из цикла, проходит по всем базовым блокам и ищет в них инструкцию OpLoopMerge. В случае, если такая инструкция найдена, ее операнды передаются в функцию process_loop. Передаваемые операнды — заголовочный блок цикла и базовый блок, в который перейдет управление после выхода из цикла (англ: merge block).

Листинг 34: поиск циклов

```
loop_invariant_code_motion(struct ir *file)
{
    for (u32 block_index = 0;
            block_index < file->cfg.labels.size;
            ++block_index) {
        struct instruction_list *ins
            = file->blocks[block_index].instructions;
        while (ins) {
            if (ins->data.opcode == OpLoopMerge) {
                u32 header_block = block_index;
                u32 merge_block
                    = ins->data.OpLoopMerge.merge_block;
                process_loop(file, header_block,
                             merge_block);
                break;
            }
            ins = ins->next;
        }
    }
}
```

Первое, что происходит в функции process_loop — это поиск внутри цикла и сохранение инструкций, производящих result id. Этот список необходим для того, чтобы определить, существует ли для некоторого операнда достигающее его определение внутри цикла. Для этого используется библиотечная функция get_result_id.

Затем, запускается основной цикл поиска, работающий до тех пор, пока множество инвариантных инструкций не стабилизируется. Для хранения этого множества используется отсортированный вектор result id.

Листинг 35: цикл поиска инвариантных инструкций

```
while (changes) {
        changes = false;
        for (u32 i = 0; i < bfs.size; ++i) {
            u32 block_index = bfs.data[i];
            struct basic_block *block
                = file->blocks + block_index;
            u32 last_invariant_size
                = invariant_operands.size;
            if (mark_block(invariant,
                           &invariant_operands,
                           &expressions, instructions,
                           block)) {
                ...
                changes = true;
            }
        }
    }
```

Функция mark_block проходит по всем инструкциям базового блока в по-

пытке найти инвариантную инструкцию, еще не отмеченную как таковую. Важно, что операнд может быть порожден ограниченным количеством инструкций:

1. операнд порожден инструкцией OpLoad, то есть является загрузкой значения входной переменной и является инвариантным;

2. операнд порожден инструкцией OpCopyObject или унарной арифметической, и необходима рекурсивная проверка;

3. операнд порожден инструкцией двух операндов — оба необходимо рекурсивно проверить на инвариантноть.

Возможно, что операнд вообще не порождается инструкцией внутри цикла, что сразу же делает его инвариантным.

Листинг 36: проверка операнда на инвариантность

```
s32 invariant_index = search_item_u32(invariant->data,
    invariant->size, operand);
s32 expr_index = search_item_u32(expressions->data,
    expressions->size, operand);
if (invariant_index != -1 || expr_index == -1) {
    return(true);
} else {
    struct instruction_t instruction
        = instructions[expr_index];
    switch (instruction.opcode) {
        case OpPhi: return(false);
        case OpLoad: return(true);
        ...
    }
}
```

После того, как все инвариантные инструкции найдены, проверяется допустимость подъема этих инструкций из цикла. Проверка условия доминирования над всеми использованиями аналогична отсутствию фи-функций с участием данной переменной. Проверка доминирования над терминальными блоками цикла производится непосредственно, с использованием дерева доминаторов, построенного библиотекой.

Инструкции, прошедшие обе проверки, помещаются в новый базовый блок, размещаемый перед заголовком цикла. Старые инструкции удаляются. Все дуги, направленные в заголовок цикла, перенаправляются в только что созданный базовый блок (кроме возвратных дуг цикла). Также, необходимо скорректировать все фи-функции в заголовке цикла, так как управление теперь приходит из другого блока. В последнюю очередь, добавляется безусловный переход из нового блока в заголовок цикла.

Листинг 37: корректировка фи-функций

```
while (inst) {
    if (inst->data.opcode == OpPhi) {
        u32 phi_parents_count
            = (inst->data.wordcount - 3) / 2;
        for (u32 i = 0; i < phi_parents_count; ++i) {
            u32 parent = inst->data.OpPhi.parents[i];
            if (parent == from_label) {
                inst->data.OpPhi.parents[i] = pre_label;
            }
        }
    } else break;
    inst = inst->next;
}
```

Функция constant_propogation_and_folding сохраняет значения всех объявленных констант, поиск которых ведется только в инструкциях, расположенных до графа потока управления. Далее, рассчитываются все достигающие определения для каждого result id, а также все инструкции, в которых используется данный result id. Для этой цели была реализована специальная функция compute_usages_of_result_id, проходящая по всем операндам всех инструкций. Важно также заметить, что result id могут иметь свое происхождение не только внутри графа потока управления, но и в инструкциях, расположенных до него, таких как, например, инструкции объявления типов, инструкции объявления констант и переменных и другие.

Этап распространения констант реализован следующим образом: для всех инструкций OpCopyObject проверяется их достигающее определение. Если порождающая инструкция это OpConstant, то инструкция операнд OpCopyObject подставляется на место использования result id OpCopyObject. Сама инструкция копирования не удаляется, так как это будет сделано позже — на этапе исключения мертвого кода.

Листинг 38: распространение констант

```
struct instruction_t *definition
    = reachdef[inst->data.OpCopyObject.operand];
if (definition->opcode == OpConstant) {
    u32 rid = inst->data.OpCopyObject.result_id;
    if (usage_counts[rid] > 0) {
        replace_usage(usage_counts, usages, rid,
                      definition->OpConstant.result_id);
        changes = true;
    }
}
```

Здесь, также, успешное распространение константы является поводом для повторения всего этапа заново после его окончания.

Для реализации светки констант было решено ограничиться обработкой лишь 32-битных целочисленных знаковых констант (исключительно в целях компактизации кода). В ходе этой оптимизации происходит поиск арифметических операций с константными операндами. Для того, чтобы определить является ли операнд константым, используется поиск по составленному ранее списку констант. К примеру, в случае с инструкцией сложения двух целых чисел:

Листинг 39: проверка операндов инструкции сложения на константность

```
u32 operand_1 = inst->data.binary_arithmetics.operand_1;
u32 operand_2 = inst->data.binary_arithmetics.operand_2;
s32 const_index_1 = search_item_u32(constants.data,
    constants.size, operand_1);
s32 const_index_2 = search_item_u32(constants.data,
    constants.size, operand_2);
if (const_index_1 != -1 && const_index_2 != -1) {
    ...
}
```

В случае, если все операнды оказались константными, извлекается значение констант и производится соответствующая арифметическая операция, и результат оформляется как новая константа. Далее, новая константа добавляется в список констант, и обновляются структуры достигающих определений. Наконец, все использования result id арифметической операции заменяются на result id объявления новой константы. Для замены использований используется функция replace_usage, осуществляющая проход по инструкциям и замену в них одного конкретного значения операнда на другое.

Листинг 40: свертка констант при сложении

```
s32 operand_1_value
    = constant_values[const_index_1].value_u32;
s32 operand_2_value
    = constant_values[const_index_2].value_u32;
s32 sum = operand_1_value + operand_2_value;
...
struct instruction_list *new_inst
    = ir_add_constant(module, newconst);
u32 new_rid = newconst.OpConstant.result_id;
changes = true;

constant_values[constants.head] = newconst.OpConstant;
vector_push(&constants, new_rid);

usages[new_rid]
    = arena_push(sizeof(struct instruction_list *));
usage_counts[new_rid] = 1;
reachdef[new_rid] = &new_inst->data;
replace_usage(usage_counts, usages, rid, new_rid);
```

Удаление (исключение) мертвого кода реализовано в функции, вызываемой последней — dead_code_elimination. Сначала, так же как и в распространении констант, вызывается функция расчета достигающих определений. Однако, в этот раз используется лишь количество ссылок на данный result id. Затем, в цикле проходятся все инструкции (как вне, так и внутри графа потока управления), и в случае, если для result id данной инструкции не существует ни одного использования, *и инструкция не является критической*, инструкция удаляется,

а для каждого из ее операндов уменьшается на единицу счетчик использований. Количество удаленных инструкций возвращается из функции с тем, чтобы определить, необходимо ли продолжать итерации, или же процесс стабилизировался.

Листинг 41: основной цикл удаления мертвого кода

```
u32 deleted;
do {
    deleted = 0;
    deleted += walk_instruction_list(NULL,
        module->pre_cfg, usage_counts);
    for (u32 i = 0; i < module->cfg.labels.size; ++i) {
        struct basic_block *block = module->blocks + i;
        deleted += walk_instruction_list(block,
            block->instructions, usage_counts);
    }
} while (deleted > 0);
```

Для сборки библиотеки и использующего ее оптимизатора были написаны Makefile для компиляции в ОС GNU/Linux и файл build.bat для сборки на Windows [Приложение А]. Библиотека компилировалась в архив для статической линковки, что позволяет не пересобирать библиотеку, если в нее не было внесено изменений, однако при разработке такая возможность не использовалась.

### 3.3 Графическое приложение

Для валидации пораждаемых оптимизатором шейдеров было реализовано простейшее приложение с использованием Vulkan API. Инициализация приложения состоит из 14 этапов: инициализация VkInstance, инициализация по-

верхности VkSurface и устройства VkDevice, инициализация командного буфера и цепочки VkSwapchainKHR, инициализация буфера глубины и uniform буфера, инициализация схемы графического конвеера и прохода VkRenderPass, инициализация шейдеров, и кадрового буфера, инициализация буфера вершин, пула дескрипторов и графического конвеера.

Этап инициализации шейдеров необходимо многократно повторять, и для этого была реализована функция rebuild_fragment_shader, избавленная от избыточности повторного вызова полной инициализации шейдеров. Инициализацию же графического конвеера необходимо вызывать заново целиком.

Листинг 42: функция пересборки фрагментного шейдера

```
rebuild_fragment_shader(char *path)
{
    VkShaderModuleCreateInfo module_create_info;
    u32 *fs_words, fs_size;
    fs_words = get_binary(path, &fs_size);
    module_create_info.sType
        = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    module_create_info.pNext    = NULL;
    module_create_info.flags    = 0;
    module_create_info.codeSize = fs_size * sizeof(u32);
    module_create_info.pCode    = fs_words;

    ASSERT_VK(vkCreateShaderModule(data.device,
        &module_create_info, NULL,
        &data.shader_stages[1].module));
}
```

В качестве инициатора обновления шейдера была использована систем-

ная библиотека inotify [9]. Один из переданных через аргументы командной строки путей к шейдеру передается библиотеке с флагом IN_MODIFY. Таким образом, при модификации файла шейдера на диске срабатывает указанная процедура. В данном случае — обновление глобальной переменной. Вся функция мониторинга запускается в отдельном потоке, так как использует блокирующее чтение. Основной цикл отрисовки же проверяет на каждой итерации значение переменной, и в случае необходимости вызывает функции обновления шейдера и графического конвеера.

Листинг 43: проверка факта модификации файла

```
inotify_add_watch(fd, argv[2], IN_MODIFY);
while (true) {
    len = read(fd, buffer, EVENT_BUF_LEN);
    p = buffer;
    while (p < buffer + len) {
        event = (struct inotify_event *) p;
        if (event->mask & IN_MODIFY) {
            shader_update = true;
        }
        p += sizeof(struct inotify_event) + event->len;
    }
}
```

Для того, чтобы не использовать лишние ресурсы процессора, цикл отрисовки организован следующим образом: в начале кадра замеряется текущее время при помощи вызова clock_gettime с флагом CLOCK_MONOTONIC [10]. В конце итерации производится второй вызов, и полученные отсчеки сравниваются. Если разница по времени меньше заданной (16.667 миллисекунд), то используется системный вызов usleep на вычисленную дельту [11].

Также, вычисленное время, затраченное на отрисовку кадра, усредняется за последние 100 кадров, и с заданной регулярностью выводится в консоль (то есть выводится скользящее среднее).

# 4 ТЕСТИРОВАНИЕ

## 4.1 Инструментарий

Для тестирования реализованного программного комплекса было вручную написано несколько шейдеров на языке GLSL, а также использовались инструменты glslangValidator для компиляции шейдеров в SPIR-V, spirv-dis для дизассемблирования шейдеров и spirv-val для первичной валидации. Сравнивались быстродействие графического приложения и размер файла шейдера на диске до оптимизации, после оптимизации утилитой spirv-val и после применения оптимизатора реализованного в настоящей дипломной работе. Для итоговой проверки корректности оптимизирующих преобразований использовалось реализованное графическое приложение, наблюдения проводились вручную.

Для запуска всего программного комплекса был написал вспомогательный скрипт launch.sh, осуществляющий полную пересборку программного комплекса (библиотеки, оптимизатора и графического приложения) и запуск графического приложения.

Листинг 44: скрипт запуска программного комплекса

```
#!/bin/bash
pushd vopt >/dev/null
echo "Building the optimizer..."
make && popd >/dev/null
pushd demo >/dev/null
echo "Building the Vulkan demo..."
make && popd >/dev/null
echo "Launching the Vulkan demo. See vopt/update.sh!"
./demo/build/debug/demo demo/shaders/sample.vert.spv  \
vopt/data/sample.frag.spv.opt
```

Для запуска оптимизатора был написал второй скрипт — update.sh, осуществляющий компиляцию шейдера на GLSL и последующий запуск оптимизатора и валидатора.

Листинг 45: скрипт запуска оптимизатора

```
#!/bin/bash
glslangValidator data/sample.frag -V \
-o data/2sample.frag.spv.in &&
./build/debug/opt data/sample.frag.spv.in \
data/sample.frag.spv.opt &&
spirv-val data/sample.frag.spv.opt
```

## 4.2 Примеры

Так как полученная реализация поддерживает очень небольшое подмножество языка SPIR-V, шейдеры для тестирования составлялись вручную. С тем, чтобы затронуть сразу все реализованные оптимизации, был составлен шейдер включающий цикл с инвариантной переменной, мертвый код, и возможность распространения и свертки констант. Код шейдера на GLSL, а также дизассемблированные SPIR-V модули до и после оптимизации приведены в Приложении Б. Сравнение времени скользящего среднего времени отрисовки кадров приведено в таблице 2, размеров SPIR-V файлов — в таблице 3.

Таблица 2: сравнение результатов работы оптимизаторов (скорость)

| № Теста | Время (мкс, до) | Время (мкс, spirv-opt) | Время (мкс, диплом) |
|---|---|---|---|
| 1 | 601 | 610 | 630 |
| 2 | 590 | 559 | 592 |
| 3 | 655 | 666 | 654 |

Можно заметить, что от запуска к запуску время отрисовки колеблется не

меньше, чем при оптимизации шейдера. По этой причине проследить какую-либо динамику не получилось. Причиной таких результатов, возможно, являются собственные оптимизации Vulkan API, применяемые на уровне драйвера.

Таблица 3: сравнение результатов работы оптимизаторов (размер)

| № Теста | Размер (байт, до) | Размер (байт, spirv-opt) | Размер (байт, диплом) |
|---|---|---|---|
| 1 | 1220 | 1220 | 756 |
| 2 | 1812 | 1812 | 772 |

# 5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

## 5.1 Возможности программы

Реализованный в рамках дипломной работы программный комплекс состоит из трех частей:

1. библиотеки для написания оптимизаций формата SPIR-V;
2. консольного приложения, реализующего оптимизирующие преобразования с использованием этой библиотеки;
3. графического приложения, предназначенного для валидации реализованных оптимизаций.

Библиотека предоставляет ее пользователю программный интерфейс (англ: API), реализующий набор методов, упрощающих процесс написания оптимизирующий преобразований. Так, библиотека позволяет преобразовать бинарный файл в промежуточное представление и предоставляет функции для модификации этого промежуточного представления, такие как: вставка и удаление инструкций, вставка и удаление базовых блоков, модификация графа потока управления (вставка, удаление и перенаправление ребер), обход графа потока управления в ширину и глубину, расчет дерева доминаторов, построение SSA формы. Также, библиотека предоставляет метод для сериализации текущего состояния промежуточного представления в поток байт, который может быть как записан на диск, так и использован напрямую в Vulkan приложении.

Более того, так как библиотека обрабатывает лишь подмножество набора инструкций формата SPIR-V, пользователь имеет возможность расширить этот набор, выполнив несколько задокументированных шагов.

Консольное приложение, использующее описанную библиотеку, реализует набор оптимизаций для формата SPIR-V, а именно: подъем кода из цикла (англ: loop invariant code motion), распространение констант (англ: constant

propogation) и удаление мертвого кода (англ: dead code elimination). Для упрощения написания приведенных преобразований также используется функция библиотеки по переведению кода в SSA форму.

Графическое приложение отображает трехмерную сцену (вращающегося вокруг вертикальной оси куба) с использованием графического API Vulkan. Приложение загружает бинарный файл с указанным именем с диска и использует содержимое этого файла в качестве фрагментного шейдера в графическом конвеере. В случае изменения содержимого файла на диске, приложение загружает файл заново и повторно инициализирует графический конвеер с обновленным шейдером.

Также, графическое приложение выводит с указанным интервалом время, затраченное на отрисовку последнего кадра. Таким образом, разработчик может убедиться, что реализуемые оптимизации эффективны.

## 5.2 Установка программы

Библиотека накладывает лишь одно ограничение на ее пользователя: для ее использования необходим компилятор языка C, поддерживающий небольшое подмножество стандарта C99. Примерами таковых являются: gcc (начиная с версии 4.6), msvc (начиная с версии 2013 года), icc (компилятор от Intel), clang и др.

Библиотека поставляется в виде заголовочного файла и архива для статической линковки. Для использования библиотеки необходимо включить заголовочный файл vopt.h.

Использование консольного приложения, реализующего оптимизации, не требует никаких зависимостей, сверх тех, что необходимы для работы библиотеки.

Для использования графического приложения необходима видеокарта с поддержкой API Vulkan, а также драйвер и заголовочные файлы, которые обыч-

но поставляются производителем графического ускорителя, либо могут быть доступны из репозиториев ОС (при их наличии). На данный момент, для проверки факта изменения файла используется системный интерфейс inotify, поэтому компиляция и запуск приложения возможны только на ОС GNU/Linux. Также, для создания окна используется библиотека xcb. Таким образом, установка зависимостей (включая необходимые для модификации исходных кодов графического приложения) может быть выполнена следующей командой:

Листинг 46: установка зависимостей графического приложения

```
# apt install libxcb1-dev libinotifytools0-dev \
libvulkan-dev mesa-vulkan-drivers
```

### 5.3 Использование программного комплекса

При условии выполнения всех указанных требований, для запуска программного комплекса можно использовать приложенные скрипты launch.sh — для пересборки всего программного комплекса и запуска графического приложения и update.sh — для запуска оптимизатора. В случае успеха, выполнение скрипта launch.sh должно выглядеть следующим образом:

Листинг 47: успешный запуск скрипта launch.sh

```
$ ./launch.sh
Building the optimizer...
[TIME] [0:00.12] Built library build/debug/libspirvopt.a
[TIME] [0:00.08] Built executable build/debug/opt
Building the Vulkan demo...
[TIME] 0:00.17
Launching the Vulkan demo. See vopt/update.sh!
```

Запуск оптимизатора также производится с помощью вспомогательного скрипта update.sh.

Листинг 48: успешный запуск скрипта update.sh

```
$ ./update.sh
data/sample.frag
```

## 5.4 Описание программного интерфейса библиотеки

### 5.4.1 Расширение набора инструкций

Набор инструкций SPIR-V, поддерживаемых библиотекой ограничен, и может быть расширен. Для того, чтобы добавить поддержку новой инструкции, необходимо выполнить следующие шаги:

1. добавить в файл vopt.h соответствующее значение перечисления (англ: enum) вида OpXXX = K, где OpXXX — опкод добавляемой инструкции, а K — числовое значение этого опкода (может быть найдено в спецификации формата SPIR-V);

2. добавить в файл vopt.h структуру xxx_t, где xxx — опкод, а поля структуру отображают содержимое операндов выбранной инструкции;

3. добавить в файле vopt.h в структуру instruction_t новое поле с именем XXX (опкод) в анонимный union;

4. добавить в файл vopt.c новый case в switch в функции instruction_parse, в котором заполнить только что добавленное поле структуры instruction_t;

5. добавить в файл vopt.c новый case в switch в функции instruction_dump, в котором заполнить временный буфер данными из структуры в строгом соответствии со спецификацией SPIR-V;

6*. модифицировать любые другие функции, в которых необходимо обработать новый опкод.

### 5.4.2 Функции взаимодействия с промежуточным представлением

Функция ir_eat предоставляет интерфейс для конструирования промежуточного представления из потока четырехбайтовых слов. Входные данные:

1. data — указатель на массив четырехбайтовых слов, содержащих корректный SPIR-V модуль;
2. size — количество слов.

Выходные данные: структура, содержащая промежуточное представление (возвращается по значению).

Листинг 49: функция ir_eat

```
struct ir
ir_eat(u32 *data, u32 size);
```

Функция ir_dump предоставляет интерфейс для сериализации промежуточного представления в файл по указанному имени. Входные данные:

1. file — указатель на структуру, содержащую промежуточное представление;
2. filename — имя файла.

Листинг 50: функция ir_dump

```
void
ir_dump(struct ir *file, const char *filename);
```

Функция ir_delete_instruction предоставляет интерфейс удаления инструкции из базового блока. Входные данные:

1. block — указатель на базовый блок, из которого удаляется инструкция;
2. inst — указатель на элемент связного списка инструкций, который нужно удалить.

Листинг 51: функция ir_delete_instruction

```
void
ir_delete_instruction(struct basic_block *block,
    struct instruction_list *inst);
```

Функция ir_prepend_instruction предоставляет интерфейс добавления инструкции в начало базового блока. Входные данные:

1. block — указатель на базовый блок, в начало которого будет добавлена инструкция;
2. inst — инструкция (передается по значению), которую нужно вставить.

Листинг 52: функция ir_prepend_instruction

```
void
ir_prepend_instruction(struct basic_block *block,
    struct instruction_t instruction);
```

Функция ir_append_instruction предоставляет интерфейс добавления инструкции в конец базового блока. Входные данные:

1. block — указатель на базовый блок, в конец которого будет добавлена инструкция;

2. inst — инструкция (передается по значению), которую нужно вставить.

Листинг 53: функция ir_append_instruction

```
void
ir_append_instruction(struct basic_block *block,
    struct instruction_t instruction);
```

Функция ir_add_bb предоставляет интерфейс создания нового базового блока. Входные данные:

1. file — указатель на структуру, содержащую промежуточное представление.

Выходные данные: индекс базового блока в массиве file->blocks.

Листинг 54: функция ir_add_bb

```
u32
ir_add_bb(struct ir *file);
```

Функция ir_destroy предоставляет интерфейс для удаления промежуточного представления и очистки памяти. Входные данные:

1. file — указатель на структуру, содержащую промежуточное представление.

Листинг 55: функция ir_destroy

```
void
ir_destroy(struct ir *file);
```

Функция ssa_convert предоставляет интерфейс для преобразования промежуточного представления в SSA форму. Входные данные:

1. file — указатель на структуру, содержащую промежуточное представление.

Листинг 56: функция ssa_convert

```
void
ssa_convert(struct ir *file);
```

### 5.4.3 Функции взаимодействия с графом потока управления

Функция cfg_add_edge предоставляет интерфейс для добавления ребра в граф потока управления. Входные данные:

1. cfg — указатель на структуру, содержащую граф потока управления;
2. from — вершина, из которой выходит ребро;
3. from — вершина, в которую входит ребро.

Выходные данные: true, если операция прошла успешно, иначе false.

Листинг 57: функция cfg_add_edge

```
bool
cfg_add_edge(struct ir_cfg *cfg, u32 from, u32 to);
```

Функция cfg_remove_edge предоставляет интерфейс для удаления ребра из графа потока управления. Входные данные:

1. cfg — указатель на структуру, содержащую граф потока управления;
2. from — вершина, из которой выходит ребро;
3. from — вершина, в которую входит ребро.

Выходные данные: true, если операция прошла успешно, иначе false.

Листинг 58: функция cfg_remove_edge

```
bool
cfg_remove_edge(struct ir_cfg *cfg, u32 from, u32 to);
```

Функция cfg_redirect_edge предоставляет интерфейс для перенаправления ребра из одной вершины в другую. Входные данные:

1. cfg — указатель на структуру, содержащую граф потока управления;
2. from — вершина, из которой выходит ребро;
3. to_old — вершина, в которую входило ребро;
4. to_new — вершина, в которую нужно направить ребро.

Выходные данные: true, если операция прошла успешно, иначе false.

Листинг 59: функция cfg_redirect_edge

```
bool
cfg_redirect_edge(struct ir_cfg *cfg, u32 from,
    u32 to_old, u32 to_new);
```

Функция cfg_show предоставляет интерфейс отладочного вывода граф в стандартный поток вывода. Входные данные:

1. cfg — указатель на структуру, содержащую граф потока управления.

Листинг 60: функция cfg_show

```
void
cfg_show(struct ir_cfg *cfg);
```

Функция cfg_dfs предоставляет интерфейс для обхода графа потока управления в глубину и сбора информации о предпорядке, постпорядке и дереве обхода. Входные данные:

1. cfg — указатель на структуру, содержащую граф потока управления.

Выходные данные: структура cfg_dfs_result, содержащая следующие поля:

1. preorder — указатель на массив, содержащий для каждой вершины ее предпорядок;
2. postorer — указатель на массив, содержащий для каждой вершины ее постпорядок;
3. parent — указатель на массив, содержащий для каждой вершины ее родителя в дереве обхода;
4. sorted_preorder — указатель на массив, содержащий вершины в предпорядке;
5. sorted_postorder — указатель на массив, содержащий вершины в постпорядке;
6. size — количество вершин во всех перечисленных массивах.

Листинг 61: функция cfg_dfs

```
struct cfg_dfs_result
cfg_dfs(struct ir_cfg *cfg);
```

Функция cfg_bfs_order предоставляет интерфейс для получения порядка предпорядка обхода в ширину. Входные данные:

1. cfg — указатель на структуру, содержащую граф потока управления.

Выходные данные: вектор, содержащий порядок обхода.

Листинг 62: функция cfg_bfs_order

```
struct uint_vector
cfg_bfs_order(struct ir_cfg *cfg);
```

Функция cfg_bfs_order_r предоставляет интерфейс для получения порядка предпорядка ограниченного обхода в ширину. Ограниченным обходом назовем обход в ширину с указанной начальной вершиной и вершиной остановки (эта вершина не посещается). Входные данные:

1. cfg — указатель на структуру, содержащую граф потока управления;
2. root — индекс начальной вершины;
3. terminate — индекс терминальной вершины.

Выходные данные: вектор, содержащий порядок обхода.

Листинг 63: функция cfg_bfs_order_r

```
struct uint_vector
cfg_bfs_order_r(struct ir_cfg *cfg,
    u32 root, s32 terminate);
```

Функция cfg_dominators предоставляет интерфейс для получения дерева доминаторов. Входные данные:

1. cfg — указатель на структуру, содержащую граф потока управления;
2. dfs — указатель на структуру, содержащую актуальный результат обхода в глубину.

Выходные данные: указатель на массив, содержащий для каждой вершины индекс ее непосредственного доминатора.

Листинг 64: функция cfg_dominators

```
s32 *
cfg_dominators(struct ir_cfg *input,
    struct cfg_dfs_result *dfs);
```

Функция cfg_whichpred предоставляет интерфейс для получения порядкового номера прямого наследника вершины в списке всех ее наследников. Входные данные:

1. cfg — указатель на структуру, содержащую граф потока управления;
2. block_index — индекс наследника;
3. pred_index — индекс родителя.

Выходные данные: порядоквый номер.

Листинг 65: функция cfg_whichpred

```
u32
cfg_whichpred(struct ir_cfg *cfg,
    u32 block_index, u32 pred_index);
```

# ЗАКЛЮЧЕНИЕ

В процессе выполнения дипломной работы был реализован оптимизатор бинарного формата SPIR-V. Для достижения поставленной цели были выполнены задачи: изучена предметная область оптимизирующих преобразований; доработана библиотека для написания оптимизаций формата SPIR-V; реализованы оптимизации: подъем инвариантного кода из циклов, распространение и свертки констант и удаление мертвого кода; изучен графический API Vulkan и реализовано с его помощью простейшее графическое приложение, позволяющее проверить корректность реализованных оптимизаций.

Реализованный оптимизатор был сравнен с утилитой spirv-opt и показал идентичные результаты по производительности. На составленных вручную примерах была показана эффективность реализованных оптимизаций, приводящих к уменьшению размера обрабатываего файла. Результат разработки можно оценить как положительный. Основная цель дипломной работы достигнута.

Дальнейшее развитие проекта может быть направлено на расширение поддерживаемого подмножества формата SPIR-V и реализацию большего количества оптимизаций.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. OpenGL Overview [Электронный ресурс] // OpenGL - The Industry's Foundation for High Performance Graphics URL: https://www.opengl.org/about/ (дата обращения: 23.06.2019).

2. Metal. Accelerating graphics and much more [Электронный ресурс] // Apple Developer URL: https://developer.apple.com/metal/ (дата обращения: 23.06.2019).

3. *Vulkan. Graphics and Compute Belong Together* [Электронный ресурс] // khronos.org URL: https://www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf (дата обращения: 12.05.2019).

4. *SPIR-V Tools* [Электронный ресурс] // github.com URL: https://github.com/KhronosGroup/SPIRV-Tools (дата обращения 15.06.2019).

5. *SPIR-V, Extended Instruction Set, and Extension Specifications* [Электронный ресурс] // khronos.org URL: https://www.khronos.org/registry/spir-v (дата обращения: 12.05.2019).

6. Rosen B., Wegman M., Zadeck K. *Global value numbers and redundant computations* // Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Нью-Йорк: 1988.

7. Об эквивалентности задачи создания полностью оптимизирующего компилятора Проблеме останова [Электронный ресурс] // cis.upenn.edu URL: http://www.cis.upenn.edu/~cis570/slides/lecture01.pdf (дата обращения: 16.06.2019).

8. K. Cooper, L. Torczon *Engineering a Compiler.* 2 изд. Сан-Франциско: Morgan Kaufmann, 2005.

9. inotify - monitoring filesystem events [Электронный ресурс] // Linux Programmer's Manual URL: http://man7.org/linux/man-pages/man7/inotify.7.html (дата обращения: 23.06.19).

10. clock and time functions [Электронный ресурс] // Linux Programmer's Manual URL: https://linux.die.net/man/3/clock_gettime (дата обращения: 23.06.19).

11. usleep - suspend execution for microsecond intervals [Электронный ресурс] // Linux Programmer's Manual URL: http://man7.org/linux/man-pages/man3/usleep.3.html (дата обращения: 23.06.19).

# ПРИЛОЖЕНИЕ А

Листинг 66: Makefile для сборки библиотеки и оптимизатора

```
APP_NAME = opt
LIB_NAME = spirvopt

MODE = Debug
CC = gcc
CFLAGS = -g -Wall -Wextra -pedantic
BUILD_PATH = build/debug

ifeq ($(MODE), Release)
    CFLAGS = -O2
    BUILD_PATH = build/release
endif

all:
    @mkdir -p $(BUILD_PATH)
    @/usr/bin/time -f "[TIME] [%E] Built library \
$(BUILD_PATH)/lib$(LIB_NAME).a" $(CC) $(CFLAGS) \
-c include/vopt.c -o $(BUILD_PATH)/lib$(LIB_NAME).o
    @ar rcs $(BUILD_PATH)/lib$(LIB_NAME).a \
$(BUILD_PATH)/lib$(LIB_NAME).o
    @/usr/bin/time -f "[TIME] [%E] Built executable \
$(BUILD_PATH)/$(APP_NAME)" $(CC) $(CFLAGS) \
main.c -L./$(BUILD_PATH) -l$(LIB_NAME) -o \
$(BUILD_PATH)/$(APP_NAME)
```

Листинг 67: build.bat для сборки библиотеки и оптимизатора

```
@echo off

set CFLAGS=/Zi /W4 /FC /GR- /EHa- /fp:except /nologo /Zf
set BUILD_PATH=build\debug
set LNAME=vopt
set XNAME=opt

pushd %BUILD_PATH%
cl /c %CFLAGS% ..\..\include\vopt.c /Folib%LNAME%.obj
lib /nologo /out:%LNAME%.lib lib%LNAME%.obj
cl %CFLAGS% ..\..\main.c %LNAME%.lib /Fe%XNAME%.exe
popd
```

# ПРИЛОЖЕНИЕ Б

Листинг 68: код шейдера на GLSL

```
#version 400

#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable

layout (location = 0) in vec4 color;
layout (location = 0) out vec4 outColor;

void main() {
    float comp = 0.0f;
    int i = 1;
    int i1 = i + 4;
    int a = 1;
    a = 2;
    int b = 2;
    int c = a + b;
    int d = c + 3;
    do {
        comp = 0.1f;
        i = i + 1;
    } while (i < 1000);

    outColor = color;
}
```

Листинг 69: дизассемблированный код шейдера до оптимизации

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 7
; Bound: 46
; Schema: 0
                OpCapability Shader
         %1 = OpExtInstImport "GLSL.std.450"
                OpMemoryModel Logical GLSL450
                OpEntryPoint Fragment %main \
"main" %outColor %color
                OpExecutionMode %main OriginUpperLeft
                OpSource GLSL 400
                OpSourceExtension \
"GL_ARB_separate_shader_objects"
                OpSourceExtension \
"GL_ARB_shading_language_420pack"
                OpName %main "main"
                OpName %comp "comp"
                OpName %i "i"
                OpName %i1 "i1"
                OpName %a "a"
                OpName %b "b"
                OpName %c "c"
                OpName %d "d"
                OpName %outColor "outColor"
                OpName %color "color"
                OpDecorate %outColor Location 0
```

Листинг 70: дизассемблированный код шейдера до оптимизации (продолжение)

```
                OpDecorate %color Location 0
        %void = OpTypeVoid
           %3 = OpTypeFunction %void
       %float = OpTypeFloat 32
%_ptr_Function_float = OpTypePointer Function %float
    %float_0 = OpConstant %float 0
         %int = OpTypeInt 32 1
%_ptr_Function_int = OpTypePointer Function %int
       %int_1 = OpConstant %int 1
       %int_4 = OpConstant %int 4
       %int_2 = OpConstant %int 2
       %int_3 = OpConstant %int 3
%float_0_100000001 = OpConstant %float 0.100000001
   %int_1000 = OpConstant %int 1000
        %bool = OpTypeBool
    %v4float = OpTypeVector %float 4
%_ptr_Output_v4float = OpTypePointer Output %v4float
   %outColor = OpVariable %_ptr_Output_v4float Output
%_ptr_Input_v4float = OpTypePointer Input %v4float
       %color = OpVariable %_ptr_Input_v4float Input
        %main = OpFunction %void None %3
           %5 = OpLabel
        %comp = OpVariable %_ptr_Function_float Function
           %i = OpVariable %_ptr_Function_int Function
          %i1 = OpVariable %_ptr_Function_int Function
           %a = OpVariable %_ptr_Function_int Function
           %b = OpVariable %_ptr_Function_int Function
```

Листинг 71: дизассемблированный код шейдера до оптимизации (продолжение)

```
          %c = OpVariable %_ptr_Function_int Function
          %d = OpVariable %_ptr_Function_int Function
               OpStore %comp %float_0
               OpStore %i %int_1
         %15 = OpLoad %int %i
         %17 = OpIAdd %int %15 %int_4
               OpStore %i1 %17
               OpStore %a %int_1
               OpStore %a %int_2
               OpStore %b %int_2
         %22 = OpLoad %int %a
         %23 = OpLoad %int %b
         %24 = OpIAdd %int %22 %23
               OpStore %c %24
         %26 = OpLoad %int %c
         %28 = OpIAdd %int %26 %int_3
               OpStore %d %28
               OpBranch %29
         %29 = OpLabel
               OpLoopMerge %31 %32 None
               OpBranch %30
         %30 = OpLabel
               OpStore %comp %float_0_100000001
         %34 = OpLoad %int %i
         %35 = OpIAdd %int %34 %int_1
               OpStore %i %35
               OpBranch %32
```

Листинг 72: дизассемблированный код шейдера до оптимизации (продолжение)

```
        %32 = OpLabel
        %36 = OpLoad %int %i
        %39 = OpSLessThan %bool %36 %int_1000
              OpBranchConditional %39 %29 %31
        %31 = OpLabel
        %45 = OpLoad %v4float %color
              OpStore %outColor %45
              OpReturn
              OpFunctionEnd
```

Листинг 73: дизасемблированный код шейдера после оптимизации

```
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 7
; Bound: 65
; Schema: 0
                OpCapability Shader
         %1 = OpExtInstImport "GLSL.std.450"
              OpMemoryModel Logical GLSL450
              OpEntryPoint Fragment %main \
"main" %outColor %color
              OpExecutionMode %main OriginUpperLeft
              OpSource GLSL 400
              OpSourceExtension \
```

Листинг 74: дизассемблированный код шейдера после оптимизации (продолжение)

```
"GL_ARB_separate_shader_objects"
                OpSourceExtension \
"GL_ARB_shading_language_420pack"
                OpName %main "main"
                OpName %outColor "outColor"
                OpName %color "color"
                OpDecorate %outColor Location 0
                OpDecorate %color Location 0
        %void = OpTypeVoid
          %3 = OpTypeFunction %void
       %float = OpTypeFloat 32
         %int = OpTypeInt 32 1
       %int_1 = OpConstant %int 1
    %int_1000 = OpConstant %int 1000
        %bool = OpTypeBool
     %v4float = OpTypeVector %float 4
%_ptr_Output_v4float = OpTypePointer Output %v4float
    %outColor = OpVariable %_ptr_Output_v4float Output
%_ptr_Input_v4float = OpTypePointer Input %v4float
       %color = OpVariable %_ptr_Input_v4float Input
        %main = OpFunction %void None %3
          %5 = OpLabel
         %45 = OpLoad %v4float %color
                OpBranch %61
         %61 = OpLabel
         %48 = OpCopyObject %v4float %45
```

Листинг 75: дизассемблированный код шейдера после оптимизации (продолжение)

```
        OpBranch %29
%29 = OpLabel
%47 = OpPhi %int %int_1 %61 %54 %32
%53 = OpCopyObject %int %47
        OpLoopMerge %31 %32 None
        OpBranch %30
%30 = OpLabel
%34 = OpCopyObject %int %53
%35 = OpIAdd %int %34 %int_1
%54 = OpCopyObject %int %35
        OpBranch %32
%32 = OpLabel
%36 = OpCopyObject %int %54
%39 = OpSLessThan %bool %36 %int_1000
        OpBranchConditional %39 %29 %31
%31 = OpLabel
        OpStore %outColor %48
        OpReturn
        OpFunctionEnd
```