Министерство образования и науки Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»

(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«Библиотека для написания оптимизирующих преобразований для формата SPIR-V»

Студент ИУ9-72 _____ И.О. Фамилия
(Подпись, дата)

Руководитель курсового проекта _____ И.О. Фамилия
(Подпись, дата)

Москва, 2019 г.

Содержание

Введение

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

Графический интерфейс Vulkan изначально создавался Khronos Group как «новое поколение OpenGL», однако позже был переименован [1]. Цели Vulkan включали, но не ограничивались:

1. уменьшением накладных расходов (англ: overhead);
2. поддежкой графических вызовов, совершаемых с нескольких потоков;
3. снижением нагрузки на центральный процессор.

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

Целью настоящей курсовой работы является разработка кроссплатформенной библиотеки для написания оптимизирующих преобразований для подмножества формата SPIR-V. Задачи, таким образом, включают: изучение формата SPIR-V, разработку промежуточного представления бинарного формата в программе, проработку и реализацию интерфейса взаимодействия с библиотекой, а также тестирование и валидацию результатов.

1 Предметная область

1.1 Формат SPIR-V

SPIR-V — это простой двоичный промежуточный язык для графических шейдеров и вычислительных ядер. Модуль SPIR-V содержит несколько точек входа, которые при этом могут вызывать общие функции. Каждая функция содержит граф потока управления, состоящий из базовых блоков, которые в том числе содержаат инструкции для описания структурированного потока управления.

Инструкции чтения и записи используются для доступа к объявленным переменным, в том числе параметрам и возвращаемым значениям. Для хранения промежуточных результатов используется SSA форма [2]. Объекты данных представляются логически, с использованием иерархии типов. Выбираемые модели адресации устанавливают, могут ли использоваться операции с указателями или доступ к памяти является чисто логическим.

Модуль SPIR-V представляется линейной последовательностью четырехбайтовых слов. Первые пять слов содержат заголовок модуля [3].

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

Все остальные слова составляют линейную последовательность инструкций в определенном порядке:

1. все инструкции `OpCapability`;
2. опциональные инструкции `OpExtension`;
3. опциональные инструкции `OpExtInstImport`;
4. одна обязательная интструкция `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`.

Тело функции, в свою очередь, должно удовлетворять правилам:

1. базовый блок всегда начинается с инструкции `OpLabel`;

2. базовый блок всегда заканчивается терминирующей инструкцией (инструкция ветвления `OpBranch` или `OpBrandConditional`, инструкция `OpReturn` или `OpReturnValue`, либо специальные инструкции 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.

Многие инструкции содержат в одном из операндов `result<id>` — уникальный идентификатор, позволяющий другим инструкциям использовать результат выполнения данной инструкции. Модуль `SPIR-V` всегда находится в SSA форме в том смысле, что любой `result<id>` всегда порождается ровно одной инструкцией.

1.2 SSA форма

SSA формой промежуточного представления называется состояние, в котором каждой переменной значение присваивается лишь единожды. При построении SSA формы переменным обычно присваиваются версии, так что присвоение значения переменной превращается в объявление новой версии данной переменной.

```
1 y = 0 --> y_1 = 0;
2 y = 1 --> y_2 = 1;
3 x = y --> x = y_2;
```

Листинг 1: пример построения простейшей SSA формы

В случае, если присваивание переменной происходит в двух разных потоках управления, вводят так называемую «фи-функцию», выбирающую нужное значение переменной в зависимости от того, из какого базового блока пришел поток управления:

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

Листинг 2: фи-функция SSA формы

SSA форма широко используется при разработке оптимизирующих комплиляторов, так как промежуточное представаление 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`, `GoLang`, `V8`, `PyPy` и многие другие.

2 Разработка алгоритма

Ключевыми, с точки зрения разработки библиотеки, являются два вопроса:

1. вопрос выбора промежуточного представления, так как именно над ним (а не над бинарным файлом) проивзодятся оптимизации;
2. вопрос определения набора функций (интерфейса), доступных для пользователя библиотеки.

2.1 Промежуточное представление

В настоещей курсовой работе промежуточное представление разделено на следующие структурные элементы:

1. заголовок — информация о программе в целом;
2. набор базовых блоков, содержащих инструкции;
3. граф потока управления — информация об условных и безусловных переходах между блоками, а также мета-информация о графе;
4. список инструкций, расположенных до графа потока управления;
5. список инструкций, расположенных после графа потока управления.

2.1.1 Заголовок

Заголовок `SPIR-V` файла в выбранном представлении точно соответствует бинарному представлению в таблице 1. Заголовок сохраняется и его содержимое (за исключением верхней границы) никак не используется, однако, за счет проверки порядка байт в магическом числе, появлятся возможность определния порядка байт на платформе,

на которой запускался комплилятор. Также теоретически реализуем промежуточный слой, обеспечивающий совместимость с более старыми форматами, версию которых также можно прочитать из заголовка.

2.1.2 Инструкции

Каждый базовый блок содержит упорядоченный список инструкций, лежащих внутри этого базового блока.

Формат инструкций близок к их бинарному представлению, однако преобразован для более простой работы с операндами. Инструкция содержит идентифицирующий её код (так называемый *опкод*), а также словарь всех операндов, доступных как на чтение, так и на запись.

Так как полная спецификация `SPIR-V` содержит более трех сотен инструкций, в настоящей курсовой работе поддерживается лишь подмножество всех инструкций. Однако предусмотрена возможность расширения этого множетсва как разработчиком, поддерживающим библиотеку, так и конечным пользователем.

2.1.3 Базовые блоки

Базовый блок является упорядоченным набором инструкций, однако для автоматизации процесса изменения графа потока управления из всех базовых блоков во внутреннем представлении удаляются инструкции объявления базового блока и инструкции условного и безусловного ветвления. Эти инструкции добавляются обратно, когда пользователь вызывает процедуру сериализации.

Так как на данный момент не поддерживается инструкция ветвления `OpSwitch`, из любого базового блока возможнен либо безусловный переход, либо условный, либо никакого. По этой причине условие представляется необязательным атрибутом базового блока.

2.1.4 Граф потока управления

Граф потока управления — множество всех возможных путей исполнения программы, представленное в виде графа с отмеченной *точкой входа* [4]. Вершинами графа являются базовые блоки, ребра же обозначают условные и безусловные переходы между блоками.

Одной из гибких с точки зрения операций над графом структур представления является список инцидентности, который и используется в данной курсовой работе.

2.1.5 Дерево доминаторов

Помимо стандартной для графа информации, предметная область оптимизирующих преобразований часто требует данных о доминировании вершин друг над другом. По этой причине в качестве дополнительных данных в представление графа добавлено дерево доминаторов, в котором каждой вершине соответствует ее непосредственный доминатор.

2.2 Интерфейс библиотеки

2.2.1 Функции манипуляции промежуточным представлением

При изучении описаний алгоритмов на псевдокоде были выделены и зафиксированы операции, необходимые для реализации этих алгоритмов, а именно:

1. конструирование внутреннего представления из потока байт, причем чтение бинарного файла с диска производится пользователем самостоятельно;

2. обратное преобразование внутреннего представления в поток байт (сериализация). По схожей логике, пользователю предоставляется лишь сам поток байт, запись же содержимого в файл производится пользователем самостоятельно, с использованием адекватных в конкретной ситуации средств;

3. удаление инструкции из базового блока — необходимо для реализации оптимизаций, передвигающих или удаляющих инструкции (к примеру, свертка констант);

4. вставка инструкции в конец или начало базового блока. Вставка инструкций необходима, например, для реализации подъема инвариантного кода из циклов;

5. создание нового базового блока. Также необходимо для подъема инвариантного кода, так как вставляется т.н. «пред-заголовок», который является новым базовым блоком, предшествующим заголовку цикла;

6. преобразование внутреннего представления в SSA форму. Ценность SSA формы для оптимизирующий преобрзований сложно переоценить: тривиализируется операция поиска достигающего определения, упрощается поиск мертвых выражений, распространение констант и многие другие [5].

2.2.2 Функции манипуляции графом потока управления

В отличие от манипуляций с внутренним представлением, манипуляции с графом потока управления практически не связаны с предметной областью оптимизаций, а лишь представляют набор операций, достаточных для преобразований графа. Таковыми являются:

1. добавление ребра между двумя вершинами;

2. удаление ребра из одной вершины в другую;

3. перенаправление ребра — операция, соединяющая удаление и добавление ребра, добавленная для удобства;

4. удаление вершины из графа (вершина удаляется вместе со всеми входящими и исходящими ребрами).

Помимо перечисленных операций, были также реализованы общие алгоритмы теории графов, результаты работы которых оказываются необходимы при написании оптимизаций:

1. обход в глубину, возвращающий предпорядок, постпорядок и дерево обхода;
2. обход в ширину, возвращающий порядок обхода;
3. ограниченный обход в ширину, позволяющий задать начальную вершину и «терминальную вершину», в которую выходить запрещено;
4. расчет дерева доминаторов;
5. функция, возвращающая для вершины ее порядковый номер в списке детей ее родителя.

3 Реализация

Библиотека была реализована на языке `C` (с использованием стандарта `C99`), так как кроссплатформенность требуется постановкой задачи. Использование минимально возможного набора системных библиотек позволило в конечной реализации добиться компиляции кода без каких-либо изменений как `gcc` так и `cl`.

3.1 Инструкции

Все поддерживаемые типы инструкций представлены как перечисление (англ: enum), с тем упростить процесс расширения набора обрабатываемых инструкций.

```
enum opcode_t {
    OpVariable = 59,
    OpLoad = 61,
    OpStore = 62,
    ...
}
```

Листинг 3: перечисление поддерживаемых типов инструкций

Для каждого элемента перечисления объявлена структура, которая содержит именованные операнды в соответствующих полях.

```
struct opvariable_t {
    u32 result_type;
    u32 result_id;
    u32 storage_class;
    u32 initializer; // optional
};
```

Листинг 4: пример структуры операндов инструкции

Все унарные и бинарные арифметические инструкции объединены в структуры `unary_arithmetics_layout` и `binary_arithmetics_layout`.

```
struct unary_arithmetics_layout {
    u32 result_type;
    u32 result_id;
    u32 operand;
};

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

Листинг 5: структуры арифметических инструкций

Итоговая структура инструкции использует объединение (англ: union) для создания псевдо-полиморфизма, а также содержит указатель `unparsed_words` на массив слов для хранения инструкций, которые не поддерживаются в текущей версии библиотеки.

```
struct instruction_t {
    enum opcode_t opcode;
    u32 wordcount;
    u32 *unparsed_words;
    union {
        struct opvariable_t OpVariable;
        ...
        struct unary_arithmetics_layout unary_arithmetics;
        struct binary_arithmetics_layout binary_arithmetics;
    }
};
```

Листинг 6: обобщенная стуктура инструкции

Таким образом, функции для работы с инструкциями могут обрабатывать все поддерживаемые типы инструкций при помощи, к примеру, оператора `switch`.

В случае, если пользователь или разработчик библиотеки захочет расширить множество поддерживаемых инструкций, выбранный способ реализации упрощает процедуру до следующих шагов:

1. добавление соотствествующего значение в перечисление `opcode_t`;
2. объявления соответствующей структуры с необходимым операндами;
3. обработки нового значения перечисления в функциях `instruction_parse` и `instruction_dump`, обеспечив, таким образом, доступность инструкции во внутреннем представлении;
4. обработки нового значения перечисления в остальных функциях, подразумевающих различное поведение для различных типов инструкций.

Так как наименьшей единицей данных в формате `SPIR-V` является четырхбайтовое слово, функция `instruction_parse` получает на вход указатель на первое слово инструкции, которую необходимо прочитать.

```
1 static struct instruction_t
2 instruction_parse(u32 *word);
```

Листинг 7: прототип функции разбора инструкции

Далее, необходимо заполнить данные инструкции, не зависящие от её типа. В соответствии со спецификацией первое слово инструкции содержит ее тип (опкод) и количество слов, которые занимает инструкция (с учетом первого слова).

```
1 static const u32 WORDCOUNT_MASK     = 0xFFFF0000;
2 static const u32 OPCODE_MASK        = 0x0000FFFF;
```

```
...
struct instruction_t instruction;
instruction.opcode = *word & OPCODE_MASK;
instruction.wordcount = (*word & WORDCOUNT_MASK) >> 16;
instruction.unparsed_words = memdup(word, instruction.wordcount * 4);

// NOTE: move the pointer to the first word of the first instruction
++word;
```

Листинг 8: разбор общей части инструкции

После сдвига указателя на следующее слово начинается разбор инструкции, зависящий от её типа. Так как большинство аргументов состоят из одного слова, разбор инструкции упрощается до сдвига и разыменования указателя. К примеру разбор инструкции типа `opvariable_t`, включающий обработку необязательного аргумента, выглядит следующим образом:

```
case OpVariable: {
    instruction.OpVariable.result_type = *(word++);
    instruction.OpVariable.result_id = *(word++);
    instruction.OpVariable.storage_class = *(word++);
    if (instruction.wordcount == 5) {
        instruction.OpVariable.result_type = *(word++);
    }
} break;
```

Листинг 9: пример разбора вариативной части инструкции

Функция `instruction_dump` работает практически точно наоборот по сравнению с функцией `instruction_parse`, однако имеет одно важное отличие: формат бинарного файла строго фиксирован спецификацией `SPIR-V`, поэтому при расширении набора поддерживаемых инструкций необходимо строго соблюдать этот формат.

Аргументом функции является инструкция и указатель на массив слов, который

используется без очистки.

```
static u32 *
instruction_dump(struct instruction_t *inst, u32 *buffer);
```

Листинг 10: прототип функции сериализации инструкции

В первый элемент буфера записываются код инструкции и количество слов, скомбинированные при помощи бинарного сдвига:

```
buffer[0] = inst->opcode | (inst->wordcount << 16);
```

Листинг 11: сериализация общей части инструкции

Далее, в зависимости от типа инструкции, остальные `inst->wordcount - 1` слов буфера заполняются операндами в порядке, фиксированном спецификацией SPIR-V. Пример для инструкции типа `opvariable_t`:

```
case OpVariable: {
    buffer[1] = inst->OpVariable.result_type;
    buffer[2] = inst->OpVariable.result_id;
    buffer[3] = inst->OpVariable.storage_class;
    if (inst->wordcount == 5) {
        buffer[4] = inst->OpVariable.result_type;
    }
} break;
```

Листинг 12: пример сериализации вариативной части инструкции

3.2 Конструирование промежуточного представления

Функция разбора бинарного файла `eat_ir` принимает массив слов, а возвращает промежуточное представление, заданное структурой ir.

```
struct ir {
    struct ir_header header;
    struct ir_cfg cfg;
    struct basic_block *blocks;
    struct instruction_list *pre_cfg;
    struct instruction_list *post_cfg;
};
```

Листинг 13: структура промежуточного представления

Заголовок считывается из входящих слов с помощью разыменования указателя на структуру (стоит заметить, что структура `ir_header` имеет поля, расположенные в строгом порядке, и выровнена по 4-х байтовым границам).

```
struct ir
ir_eat(u32 *data, u32 size)
{
    struct ir file;
    file.header = *((struct ir_header *) data);
    ...
}
```

Листинг 14: функция конструирования промежуточного представления

Дальнейший разбор файла разделен на следующие этапы:

1. разбор всех инструкций и подсчет базовых блоков;
2. отделение инструкций, лежащих до первого базового блока;
3. заполнение графа потока управления;
4. отделение инструкций, лежащих после последнего базового блока.

Так как списки инструкций представляются в виде двунаправленного списка (для

упрощения реализации операций вставки и удаления инструкций), все инструкции заворачиваются в структуру `instruction_list`.

```
struct instruction_list {
    struct instruction_t data;
    struct instruction_list *next;
    struct instruction_list *prev;
};
```

Листинг 15: элемент двунаправленного списка инструкций

Таким образом, разбор потока инструкций начинается со свдига указателя на размер заголовка, после чего все остальные инструкции разбираются в цикле вызовом функции `instruction_parse` и присоединяются к концу растущего двунаправленного списка инструкций.

```
...
u32 *word = data + offset;

// NOTE: get all instructions in a list, count basic blocks
do {
    inst = malloc(sizeof(struct instruction_list));
    inst->data = instruction_parse(word);
    inst->prev = last;

    if (inst->data.opcode == OpLabel) {
        labels[bb_count++] = inst->data.OpLabel.result_id;
    }

    offset += inst->data.wordcount;

    if (last) {
        last->next = inst;
    } else {
        all_instructions = inst;
```

```
20      }
21
22      last = inst;
23      word = data + offset;
24  } while (offset != size);
25
26  last->next = NULL;
```

Листинг 16: составление полного списка инструкций

Затем, находится первая инструкция первого базового блока, и на этом месте список разрывается. Первую из получившихся частей содержит все инструкции, расположенные до базовых блоков.

```
1  // NOTE: all pre-cfg instructions
2  file.pre_cfg = all_instructions;
3  inst = all_instructions;
4  do {
5      inst = inst->next;
6  } while (inst->data.opcode != OpLabel);
7
8  inst->prev->next = NULL;
9  inst->prev = NULL;
10 // =======================
```

Листинг 17: отделение инструкций, расположенных до базовых блоков

Цикл разбора инструкций, содержащихся внутри базовых блоков, работает по следующей схеме:

1. обработка начинается с заголовочной инструкции `OpLabel`;
2. указатель сдвигается на следующую инструкцию, так как заголовочные и терминальные интрукции, а также инструкции ветвления, хранятся неявно;

3. расчитывается размер базового блока (количество инструкций) простым проходом по до первой инструкции ветвления или терминальной инструкции;

4. в случае, если блок завершается инструкцией безусловного перехода `OpBranch`, добавляется ребро в граф потока управления;

5. в случае, если блок завершается инструкцией условного ветвления `OpBranchConditional`, добавляется две ребра в граф потока управления, а также сохраняется идентификатор (`result<id>`) условия в артибуты текущего базового блока;

6. общий список инструкций разрывается.

```
while (inst->data.opcode == OpLabel) {
    block.count = 0;
    struct instruction_list *start = inst->next;
    inst = inst->next; // NOTE: skip OpLabel

    while (!terminal(inst->data.opcode)) {
        inst = inst->next;
        ++block.count;
    }

    block.instructions = (block.count > 0 ? start : NULL);

    if (inst->data.opcode == OpBranch) {
        u32 edge_id = inst->data.OpBranch.target_label;
        u32 edge_index = search_item_u32(labels, bb_count, edge_id);
        cfg_add_edge(&file.cfg, block_number, edge_index);
    } else if (inst->data.opcode == OpBranchConditional) {
        file.cfg.conditions[block_number] = inst->data.OpBranchConditional.condition;
        u32 true_edge = search_item_u32(labels,
                                        bb_count,
```

```
                                        inst->data.OpBranchConditional.true_label);
        u32 false_edge = search_item_u32(labels,
                                         bb_count,
                                         inst->data.OpBranchConditional.false_label);
        cfg_add_edge(&file.cfg, block_number, true_edge);
        cfg_add_edge(&file.cfg, block_number, false_edge);
    }

    if (!supported_in_cfg(inst->data.opcode)) {
        fprintf(stderr,
                "[ERROR] Unsupported instruction (opcode %d) in CFG\n",
                inst->data.opcode);
        exit(1);
    }

    struct instruction_list *save = inst;
    inst->prev->next = NULL;
    inst->next->prev = NULL;
    inst = inst->next;

    file.blocks[block_number++] = block;
}
```

Листинг 18: составление графа потока управления

Наконец, инструкции, расположенные после последнего базового блока, сохраняются в соответствующее поле структуры промежуточного представления. Никакая дополнительная работа на этом этапе уже не требуется.

```
// NOTE: all post-cfg instructions
file.post_cfg = inst;
// =======================
```

Листинг 19: отделение инструкций, расположенных после базовых блоков

3.3 Построение SSA формы

Для построения SSA формы был применен алгоритм, использующий фронт доминаторов [6], и состоящий из следующих шагов:

1. поиск переменных и присваиваний;
2. вставка фи-функций;
3. переименование переменных и замена инструкций записи `OpStore` и чтения `OpLoad` на инструкции копирования `OpCopyObject`;
4. удаление старых переменных и сопутствующих им декорирующих инструкций.

Расчет дерева доминаторов производился с помощью алгоритма, предпложенного Ленгауэром и Тарьяном в 1979 году [7]. Для реализации алгоритмов обхода в ширину и глубину были использованы стек и очередь вершин, соответственно.

Для повышения читаемости полученного бинарного файла с помощью диззасембилрующих утилит были также добавлены декорирующие инструкции `OpName` для всех SSA имён.

Для расчета итерированного фронта доминаторов были реализованы функции проверки наличия элемента в массиве `search_item_u32` и функция `vector_push_maybe`, использующие линейный поиск для гарантии уникальности элементов.

```
1 static inline s32
2 search_item_u32(u32 *array, u32 count, u32 item)
3 {
4     for (u32 i = 0; i < count; ++i)
5         if (array[i] == item) return(i);
6     return(-1);
7 }
```

Листинг 20: функция линейного поиска

```
// NOTE: push only if not in vector already
static bool
vector_push_maybe(struct uint_vector *v, u32 item)
{
    if (search_item_u32(v->data, v->size, item) == -1) {
        vector_push(v, item);
        return(true);
    }
    return(false);
}
```

Листинг 21: функция добавления уникального элемента в вектор

3.4 Граф потока управления

Граф потока управления программы был представлен в виде списка инцидентности, дополненного информацией о входящих ребрах (так как они нужны для поиска доминаторов).

```
struct ir_cfg {
    struct uint_vector labels; // NOTE: zero means 'deleted'
    u32 *conditions;
    s32 *dominators;
    struct edge_list **out;
    struct edge_list **in;
};
```

Листинг 22: структура графа потока управления

Каждый элемент массива `out` или `in` является однонаправленным списком ребер, содержащий как номер вершины, в которую уходит (или из которой приходит) данное ребро, так и указатель на следующий элемент списка (либо `NULL`, если этот элемент является последним).

```
struct edge_list {
    u32 data;
    struct edge_list *next;
};
```

Листинг 23: элемент списка ребер

3.5 Интерфейс библиотеки

Функции, предназначеные для использования извне библиотеки, были вынесены в заголовчный файл `headers.h`, содержащий прототипы и описания этих функций. Стоит также заметить, что в отличие от всех остальных функций, написанных в ходе реализации библиотеки, функции в этом файле *не* отмечены спецификатором `static`, что гарантирует доступность этих функций извне данной единицы компиляции. Таким образом, библиотека может быть использована как в виде исходных кодов (как часть проекта), так и в виде уже скомпилированного объектного файла.

```
// NOTE: read a sequence of 4 byte words and produce an intermideate representation
struct ir
ir_eat(u32 *data, u32 size);
...
// NOTE: adds an outgoing edge at basic block 'from' to basic block 'to'
// as well as an incoming edge at basic block 'to' from basic block 'from'.
// Returns true if the action was succesful, and false otherwise
bool
cfg_add_edge(struct ir_cfg *cfg, u32 from, u32 to);
```

Листинг 24: пример функций из заголовочного файла `headers.h`

Заголовочный файл `headers.h` также содержит инструкций по расширению поддерживаемого набора инструкций.

3.6 Сериализация промежуточного представления

Сериализация промежуточного представления, почти зеркально противоположна функции разбора промежуточного представления, однако несколько важных отличий все-таки существует. Так, при сериализации базовых блоков, считывается информация из графа потока управления (и, возможно, дополнительных атрибутов базового блока) для создания инструкции условного или безусловного перехода.

```
struct instruction_t termination_inst;
if (edge_count == 0) {
        termination_inst.opcode = OpReturn;
        termination_inst.wordcount = 1;
    } else if (edge_count == 1) {
        termination_inst.opcode = OpBranch;
        termination_inst.wordcount = 2;
        termination_inst.OpBranch.target_label =
                    file->cfg.labels.data[file->cfg.out[i]->data];
    } else if (edge_count == 2) {
        termination_inst.opcode = OpBranchConditional;
        termination_inst.wordcount = 4;
        termination_inst.OpBranchConditional.condition = file->cfg.conditions[i];
        termination_inst.OpBranchConditional.true_label =
                    file->cfg.labels.data[file->cfg.out[i]->data];
        termination_inst.OpBranchConditional.false_label =
                    file->cfg.labels.data[file->cfg.out[i]->next->data];
    }

    words = instruction_dump(&termination_inst, buffer);
```

Листинг 25: конструирование инструкции ветвления

Аналогично, создается инструкция аннотации базового блока `OpLabel`.

```
struct basic_block block = file->blocks[i];
struct oplabel_t label_operand = {
```

```
    .result_id = file->cfg.labels.data[i]
};

struct instruction_t label_inst = {
    .opcode = OpLabel,
    .wordcount = 2,
    .OpLabel = label_operand
};

words = instruction_dump(&label_inst, buffer);
```

Листинг 26: конструирование инструкции `OpLabel`

Также стоит заметить, что неподдерживаемые инструкции сериализуются с использованием поля `instruction.unparsed_words`.

```
switch (inst->opcode) {
    ...
    default: {
        memcpy(buffer, inst->unparsed_words, inst->wordcount * 4);
    }
}
```

Листинг 27: сериализация неподдерживаемых инструкций

4 Тестирование

4.1 Инструментарий

Для тестирования полученной реализации использовались утилиты `spirv-dis` и `spirv-val`, предоставляемые `Khronos Group` [8].

Консольная утилита `spirv-dis` предоставляет возможность просмотра бинарного файла `SPIR-V` в виде последовательности инструкций.

Консольная утилита `spirv-val` позволяет валидировать данный бинарный файл `SPIR-V` на предмет выполнения правил, установленных спецификацией. Согласно постановке задачи, валидация на уровне библиотеки не производится, поэтому валидирующая утилита имела особую важность в проверке корректности пораждаемых библиотекой бинарных данных.

Для пораждения `SPIR-V` файлов использовался эталонный компилятор `glslangValidator`.

4.2 Проверка корректности SSA формы

Проверка полученной SSA формы производилось путем ручного изучения полученного бинарного файла утилитой `spirv-dis`, а также автоматически валидировалась утилитой `spirv-val`. В качестве примера можно рассмотреть следущий шейдер, написанный на языке `GLSL`:

```
1 #version 450
2
3 void main()
4 {
5     int a = 1;
6     int b;
7     int c = 1;
```

```
    int d;

    while (true) {
        if (true) {
            c = 2;
        }

        b = a + 1;
        d = c + 1;
    }
}
```

Листинг 28: код шейдера на языке GLSL

Соотвествующий бинарный файл представляется следующей последовательностью инструкций, полученных при помощи утилиты `spirv-dis`:

```
               OpCapability Shader
          %1 = OpExtInstImport "GLSL.std.450"
               OpMemoryModel Logical GLSL450
               OpEntryPoint Fragment %main "main"
               OpExecutionMode %main OriginUpperLeft
               OpSource GLSL 450
               OpName %main "main"
               OpName %a "a"
               OpName %c "c"
               OpName %b "b"
               OpName %d "d"
       %void = OpTypeVoid
          %3 = OpTypeFunction %void
        %int = OpTypeInt 32 1
%_ptr_Function_int = OpTypePointer Function %int
      %int_1 = OpConstant %int 1
       %bool = OpTypeBool
       %true = OpConstantTrue %bool
      %int_2 = OpConstant %int 2
       %main = OpFunction %void None %3
```

```
21           %5 = OpLabel
22           %a = OpVariable %_ptr_Function_int Function
23           %c = OpVariable %_ptr_Function_int Function
24           %b = OpVariable %_ptr_Function_int Function
25           %d = OpVariable %_ptr_Function_int Function
26                OpStore %a %int_1
27                OpStore %c %int_1
28                OpBranch %11
29          %11 = OpLabel
30                OpLoopMerge %13 %14 None
31                OpBranch %15
32          %15 = OpLabel
33                OpBranchConditional %true %12 %13
34          %12 = OpLabel
35                OpSelectionMerge %19 None
36                OpBranchConditional %true %18 %19
37          %18 = OpLabel
38                OpStore %c %int_2
39                OpBranch %19
40          %19 = OpLabel
41          %22 = OpLoad %int %a
42          %23 = OpIAdd %int %22 %int_1
43                OpStore %b %23
44          %25 = OpLoad %int %c
45          %26 = OpIAdd %int %25 %int_1
46                OpStore %d %26
47                OpBranch %14
48          %14 = OpLabel
49                OpBranch %11
50          %13 = OpLabel
51                OpReturn
52                OpFunctionEnd
```

Листинг 29: бинарный SPIR-V файл до преобразований

После вызова функций `ir_eat`, `ssa_convert` и `ir_dump`, был получен следующий бинарный файл:

```
               OpCapability Shader
          %1 = OpExtInstImport "GLSL.std.450"
               OpMemoryModel Logical GLSL450
               OpEntryPoint Fragment %main "main"
               OpExecutionMode %main OriginUpperLeft
               OpSource GLSL 450
               OpName %main "main"
               OpName %ssa0 "ssa0"
               OpName %ssa1 "ssa1"
               OpName %ssa1_0 "ssa1"
               OpName %ssa1_1 "ssa1"
               OpName %ssa1_2 "ssa1"
               OpName %ssa2 "ssa2"
               OpName %ssa3 "ssa3"
       %void = OpTypeVoid
          %3 = OpTypeFunction %void
        %int = OpTypeInt 32 1
%_ptr_Function_int = OpTypePointer Function %int
      %int_1 = OpConstant %int 1
       %bool = OpTypeBool
       %true = OpConstantTrue %bool
      %int_2 = OpConstant %int 2
       %main = OpFunction %void None %3
          %5 = OpLabel
       %ssa0 = OpCopyObject %int %int_1
       %ssa1 = OpCopyObject %int %int_1
               OpBranch %11
         %11 = OpLabel
         %27 = OpPhi %int %ssa1 %5 %ssa1_2 %14
     %ssa1_0 = OpCopyObject %int %27
               OpLoopMerge %13 %14 None
               OpBranch %15
         %15 = OpLabel
               OpBranchConditional %true %12 %13
         %12 = OpLabel
               OpSelectionMerge %19 None
               OpBranchConditional %true %18 %19
```

```
38          %18 = OpLabel
39      %ssa1_1 = OpCopyObject %int %int_2
40                OpBranch %19
41          %19 = OpLabel
42          %28 = OpPhi %int %ssa1_0 %12 %ssa1_1 %18
43      %ssa1_2 = OpCopyObject %int %28
44          %22 = OpCopyObject %int %ssa0
45          %23 = OpIAdd %int %22 %int_1
46        %ssa2 = OpCopyObject %int %23
47          %25 = OpCopyObject %int %ssa1_2
48          %26 = OpIAdd %int %25 %int_1
49        %ssa3 = OpCopyObject %int %26
50                OpBranch %14
51          %14 = OpLabel
52                OpBranch %11
53          %13 = OpLabel
54                OpReturn
55                OpFunctionEnd
```

Листинг 30: бинарный SPIR-V файл в SSA форме

Можно заметить, что были вставлены две фи-функции, а инструкции OpVariable, OpStore и OpLoad были удалены. Проверка утилитой spirv-val не показала каких-либо ошибок.

Заключение

В ходе выполнения курсовой работы был изучен бинарный формат SPIR-V, разработано промежуточное представление этого формата и интерфейс взаимодействия с этим промежуточным представлением.

На основании поставленных требований была разработана кроссплатформенная библиотека для написания оптимизаций бинарного формата SPIR-V, предоставляющая функции для получения информации о промежуточном представлении и графе потока управления. Для дальнейшего упрощения пользования библиотекой была разработана функция преобразования SPIR-V моделя в SSA форму, не использующую операторов OpStore и OpLoad, а также функции расчета дерева доминаторов, обходов в ширину и глубину.

Разработанный функционал был реализован с использованием языка C99 с использованием минимального набора системных заголовочных файлов, что позволило добиться компилятции библиотеки как на GNU/Linux, так и на Windows без каких-изменений.

Полученная реализация была протестирована и валидирована с использованием утилит spirv-dis и spirv-val. Процесс тестирования показал, что выбранные алгоритмы работают корректно и эффективно, а использование библиотеки для конечного пользователя максимально упрощено.

Дальнейшнее развитие проекта может быть направлено на расширение множества поддерживаемых инструкций, а также на написание оптимизирующих преобразований, использующих данную библиотеку.

Список литературы

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

[2] 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.

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

[4] Allen F. *Control flow analysis* // Proceedings of a symposium on Compiler optimization. Иллинойс: Urbana-Champaign, 1970.

[5] Muchnick S. *Advanced Compiler Design and Implementation.* Сан Франциско: Morgan Kaufmann Publishers, 1997.

[6] Cytron R., Ferrante J,. Rosen B. *Efficiently computing static single assignment form and the control dependence graph* // ACM Transactions on Programming Languages and Systems. 1991. С. 451-490.

[7] Lengauer T., Tarjan R. *A Fast Algorithm For Finding Dominators in a Flowchart* // ACM Transactions on Programming Languages and Systems. 1979. С. 121-141.

[8] *SPIRV-Tools* // GitHub URL: `https://github.com/KhronosGroup/SPIRV-Tools` (дата обращения: 12.05.2019).