Санкт-Петербургский государственный университет информационных технологий,
механики и оптики

Кафедра компьютерных технологий

Е. В. Селифонов

Оптимизация работы программ для графического процессора

Бакалаврская работа

Научный руководитель: А. А. Шалыто

Санкт-Петербург
2009

Оглавление

Введение

Ниже обосновывается целесообразность параллельных вычислений, показываются некоторые особенности графических процессоров, а также приводятся примеры задач, эффективно решаемых на графических процессорах.

Параллельные вычисления. Вычислительные мощности компьютеров постоянно растут, а вместе с ними усложняются задачи, возлагаемые на вычислительные машины. До последнего времени проблема несоответствия вычислительных мощностей машин и потребностей задач решалась увеличением тактовой частоты процессоров. Однако данный подход изжил себя, так как производители процессоров подошли вплотную к физическому пределу частот, связанному с максимальной скоростью прохождения сигнала. В то же время научные исследования требовали проведения все более сложных расчетов.

Проблема была решена при помощи *распараллеливания* вычислений. Рассмотрим различные способы осуществления данного подхода.

- Объединение нескольких компьютеров в сеть. Расчеты ведутся на каждом компьютере, а главный компьютер, называемый *координатором*, распределяет задачи и записывает результаты вычислений.
- Установка на один компьютер нескольких процессоров. Данный подход наиболее популярен в настоящее время благодаря созданию *многоядерных* процессоров, которые обладают высокой скоростью обмена данными, а также низкой стоимостью производства.
- Создание специальных процессоров с наборами инструкций с принципами *SIMD* (*Single Instruction – Multiple Data, одна инструкция – массив данных*) и *MIMD* (*Multiple Instructions – Multiple Data,*

несколько инструкций – массив данных), называемых также *векторными* процессорами. Такие процессоры способны за одну инструкцию обрабатывать целый массив данных (например, вектор из вещественных чисел), а некоторые способны обрабатывать несколько инструкций одновременно. Стоит отметить, что данный подход настолько распространен, что даже в самые популярные на данный момент процессоры, базирующиеся на архитектуре *x86*, изобретенной еще в конце 70-х годов прошлого века, постоянно добавляются новые *SIMD*-инструкции: *MMX*, *SSE*, *3DNow!*, *SSE2*, *(S)SSE3*, *SSE4*.

Графические процессоры. *Графические процессоры* (*GPU – Graphics Processing Unit*) были созданы специально для обработки трехмерной графики. Рассмотрим отличительные особенности их архитектуры:

- процесс обработки представляется в виде множества конвейеров за счет наличия множества потоковых процессоров;
- инструкции нацелены на работу с векторами и матрицами из вещественных чисел.

Стоит отметить, что если старые видеокарты (например, на базе чипа *NV34*) обрабатывали изображение в *четыре* потока, то для новых (например, на базе двух чипов *RV770*) эта цифра выросла в разы – они содержат *1600* потоковых процессоров [1].

До последнего времени использование графических процессоров было ограничено только обработкой трехмерной графики, в компьютерных играх и в программах создания трехмерных моделей. Однако недавно их начали использовать и для других целей, что привело к появлению концепции *GPGPU* (*General Purpose Graphics Processors Usage – использование графических процессоров для общих целей*) [2]. К сожалению, на данный момент данная концепция не получила широкого распространения в силу своей новизны и отсутствия единого стандарта. Тем не менее, начинают

создаваться средства для упрощения написания программ для графических процессоров, которые будут рассмотрены ниже.

Примеры задач, эффективно решаемых на графических процессорах. Рассмотрим простой пример задачи, которую выгоднее решать при помощи графического процессора, и сравним его с центральным процессором.

Пусть дан вектор X из N вещественных чисел. Требуется применить к нему следующий оператор:

$A: R^n \to R^n$

$AX = \alpha X + \beta$

при заданных коэффициентах α *и* β .

Закон Амдала [3] утверждает следующее.

Предположим, что необходимо решить некоторую вычислительную задачу. Предположим, что ее алгоритм таков, что доля α от общего объема вычислений может быть получена только последовательными расчетами, а, соответственно, доля $1 - \alpha$ может быть *распараллелена идеально* (время вычисления будет обратно пропорционально числу задействованных узлов p). Тогда ускорение, которое может быть получено на вычислительной системе из p процессоров, по сравнению с однопроцессорным решением не будет превышать величины:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} .$$

Отметим, что решение этой задачи идеально распараллеливается: значение компонента нового вектора зависит только от значения того же компонента старого вектора. Поэтому для решения задачи достаточно для каждого компонента сосчитать его новое значение:

$$\forall x_i \quad x_i^* = \alpha x_i + \beta .$$

Таким образом, для больших N будет справедливо следующее утверждение: чем больше процессоров выполняют вычисления, тем быстрее будет решена задача (прямо пропорционально числу процессоров).

Современные центральные процессоры персональных компьютеров имеют *четыре* ядра, тогда как для графических процессоров это число больше в *400* раз. Учитывая тот факт, что инструкции графических процессоров оптимизированы для работы с векторами из четырех элементов, получим еще большее ускорение. Отметим, что для *GPU* характерна меньшая частота процессора, чем для *CPU* (*Central Processing Unit, центральный процессор*): современные *CPU* имеют частоту порядка 3 ГГц, тогда как для современных *GPU* этот параметр равен 750 МГц (проигрыш в *четыре* раза). Однако для данной задачи, как можно видеть, проигрыш в частоте процессора уходит на второй план.

Естественен тот факт, что круг задач, эффективно решаемых графическими процессорами, весьма ограничен в силу их специфики, однако многие такие задачи важны для прикладных и исследовательских целей. Например, исследовательские проекты *Folding@home* [4] и *GPUGRID* [5] используют мощь графических процессоров пользователей-добровольцев со всего мира. Стоит также отметить, что задачи на клеточных автоматах, в которых на каждой итерации новое значение в клетке обычно зависит только от старого значения в ней и ее соседях, также могут быть эффективно решены при помощи графических процессоров [6]. Существует даже проект [7], позволяющий выполнять некоторые операции над базами данных при помощи *GPU*.

Глава 1. Обзор существующих средств для написания программ для *GPU*

В этой главе приводится краткое описание существующих способов программирования графических процессоров, объясняются некоторые базовые понятия, приводятся примеры специализированных *GPGPU*-библиотек.

1.1. Графические библиотеки и языки программирования

Программирование графических процессоров прошло такой же путь развития, как и для центральных процессоров.

Не существовало способов эффективно писать программы для первых графических процессоров. Если разработчик приложения хотел включить специальные эффекты для обработки изображения, то ему приходилось либо добавлять обработчики, работающие на *CPU*, что значительно замедляло скорость выполнения программы, либо довольствоваться сильно ограниченным набором поддерживаемых аппаратно эффектов.

Для удобного написания программ разработчикам требовалось создать специализированный язык, как в свое время для *CPU* был создан язык *C*, который бы позволял создавать простые и быстрые программы (язык должен быть приближен к возможностям аппаратной части) для создания и применения различных эффектов. Была разработана и стандартизирована концепция *шейдеров* (англ. *shader*).

Шейдер – это записанная на специальном языке программа для графического процессора. По своему типу шейдеры делятся на два основных типа:

- *вершинные шейдеры (vertex shaders)* оперируют с различными геометрическими данными вершин объектов в трехмерном пространстве;
- *пиксельные* или *фрагментные шейдеры* (*pixel / fragment shaders*) работают с фрагментами изображения, *пикселями*, преобразуя их некоторым образом, что может использоваться для *постпроцессинга* (*post-processing*) изображения – преобразования уже полученного изображения в новое (например, для применения эффекта черно-белого цвета).

Модель шейдеров (*Shader Model*) стандартизирует некоторые технические параметры шейдеров. В настоящее время последней версией является *Shader Model 4*.

Сейчас распространены две графические библиотеки: *OpenGL* [8] и *Direct3D* [9]. Для каждой из них были созданы *C*-подобные языки для написания шейдеров:

- для *OpenGL*: *GLSL* (*The OpenGL Shading Language*) [10];
- для *Direct3D*: *HLSL* (*High Level Shader Language*) [11];
- для *OpenGL* и *Direct3D*: *Cg* (*C for Graphics*) [12].

Данные языки очень схожи с *C*, поддерживают различные типы данных (например, *int*, *float*), структуры, функции. Начиная с *ShaderModel 3*, поддерживаются операторы ветвления. Отличительной особенностью является наличие встроенных типов для векторов и матриц, а также множества функций, оперирующих с ними.

Стоит отметить, что также существуют низкоуровневые, ассемблероподобные языки, например, *DirectX ASM* [13].

1.2. Графический конвейер

Для того чтобы разобраться, как работают шейдеры и в каком порядке они применяются, рассмотрим упрощенную схему графического конвейера (рис. 1).

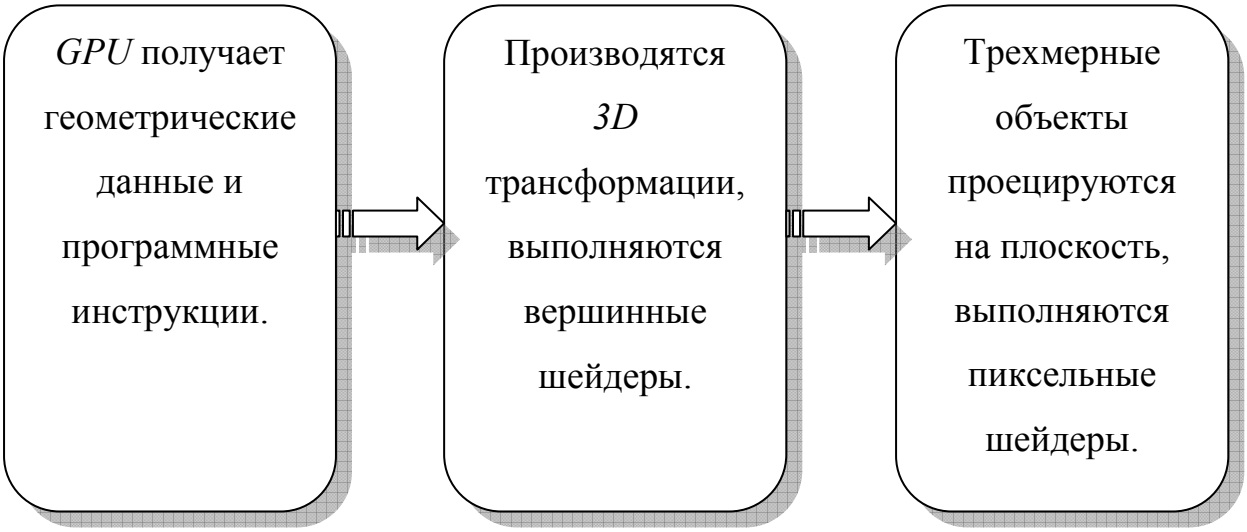

Рис. 1. Упрощенная схема графического конвейера

Обычно для решения задач на графических процессорах используются пиксельные шейдеры, поэтому остановимся на них подробнее. Как показано выше, на последнем этапе происходит проецирование трехмерных объектов на плоскость. На объекты накладывается *текстура* (двумерное изображение, накладываемое на грани трехмерных объектов), а для каждого пикселя вычисляется результирующее значение цвета. Далее для каждого пикселя выполняются фрагментные шейдеры. В результате работы последнего из них получается финальный цвет пикселя, который и выводится на экран. Таким образом, каждый фрагментный шейдер будет выполнен N раз, где N – число пикселей результирующего изображения. Именно за счет наличия множества конвейерных процессоров фрагментные шейдеры выполняются быстро.

1.3. *GPGPU*-средства

В настоящее время два наиболее известных производителя видеокарт, *AMD* и *nVidia*, выпустили свои версии архитектуры и высокоуровневые *SDK*

(*Software Development Kit, средство для разработки программ*) для общих расчетов на графических процессорах, соответственно, *CUDA* (*Compute Unified Device Architecture*) [14] и *CTM* (*Close to Metal*) [15].

Положительной стороной данных средств является их высокоуровневость: не требуется знать специальные языки для написания шейдеров, программы записываются в синтаксисе, схожем с языком *C*.

Их основным недостатком является «заточенность» только под графические процессоры собственного производства: *CUDA* не работает на видеокартах на базе чипов *AMD*. К сожалению, здесь проявляется отсутствие в данный момент единого стандарта. Ситуация может быть исправлена с появлением новой версии *Direct3D 11*: в ней должен появиться так называемый *вычислительный шейдер* (*Compute Shader*) [16].

Также следует отметить, что часто использование данных подходов не избавляет от требования тонкого знания архитектуры графических процессоров при возникновении специфических проблем (например, известно, что на графических процессорах серии *GeForce 8000* при переключении режимов работы между *CUDA* и традиционным возникают большие задержки). Более того необходимо знать ограничения, накладываемые на программы графическими процессорами.

Также существуют проекты от независимых исследователей (например, проект *Sh* [17]). К сожалению, в процессе работы с такими проектами все равно приходится сталкиваться с низкоуровневыми проблемами. Некоторые из этих проектов (в частности, проект *Sh*) были приобретены крупными компаниями и теперь носят коммерческий характер.

Выводы по главе 1

- Рассмотрены графические библиотеки и языки программирования для графических процессоров.
- Показана особенность фрагментных шейдеров: один и тот же код выполняется для каждого пикселя.

- Отмечены существующие средства разработки *GPGPU*-программ и выявлены их недостатки.

Глава 2. Традиционный подход к *GPGPU*

В этой главе более подробно раскрываются особенности низкоуровневого программирования графических процессоров, показываются его преимущества и недостатки. В приведенных примерах используются возможности среды *OpenGL* и языка программирования *GLSL*. Более подробные примеры приведены в статье [18].

2.1. Цели рендеринга

Традиционно видеокарты используются для *рендеринга* (*rendering*) – отрисовки изображения на экран. Целью *GPGPU* является проведение некоторых математических расчетов с использованием возможностей графического процессора, поэтому необходимо изменить стандартную схему работы и отказаться от вывода на экран.

Рассмотрим упрощенную схему отрисовки одного кадра изображения (рис. 2).

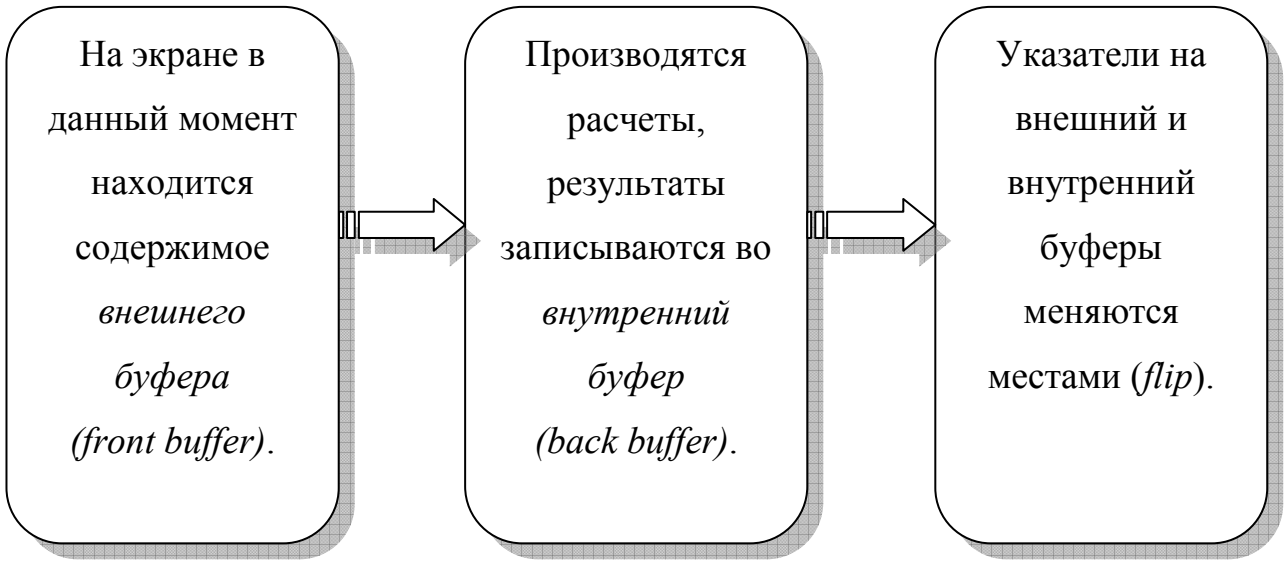

Рис. 2. Упрощенная схема отрисовки одного кадра изображения

Цель рендеринга (*render target*) – область памяти, в которую заносятся результаты рендеринга. Обычно целью рендеринга является *внутренний буфер* – специальная область памяти, в которую заносятся результаты рендеринга нового кадра. Наличие двух постоянно меняющихся буферов связано с тем, что в процессе расчетов необходимо выводить старую картинку на экран, иначе будет заметно мерцание.

OpenGL предоставляет специальный объект, называемый *фреймбуфер* (*framebuffer*), который можно сделать целью рендеринга, а затем прочитать из него результаты расчетов. Для создания такого объекта необходимо вызвать функцию *glGenFramebuffersEXT*, а для установки его в качестве цели рендеринга – функции *glBindFramebufferEXT*, *glDrawBuffer*.

2.2. Представление данных

Теперь рассмотрим вопрос о представлении данных в графическом процессоре. С точки зрения центрального процессора происходит оперирование *массивами* вещественных чисел. Такие массивы могут иметь разную размерность, но чаще всего приходится иметь дело с одномерными и двумерными массивами. С точки зрения графического процессора все операции проводятся с *текстурами*, которые, по сути, являются двумерными массивами. Однако следует отметить, что каждый элемент текстуры обычно представляет собой вектор из четырех вещественных чисел. Это связано с тем, что традиционно каждый элемент текстуры обозначает цвет, который состоит из четырех компонентов: *r* (*red*, красный), *g* (*green*, зеленый), *b* (*blue*, синий) и *a* (*alpha*, альфа-канал, при смешивании цветов обозначает долю данного цвета в смеси). Для цветов проводится нормализация их значения – оно лежит в диапазоне $[0.0; 1.0]$.

Таким образом, если требуется обработать одномерный массив из N чисел, необходимо воспользоваться текстурой, состоящей из $\frac{N}{4}$ элементов, с

размерами, например, $\frac{\sqrt{n}}{2} \times \frac{\sqrt{n}}{2}$. Если размеры получаются нецелые, можно воспользоваться другими, для которых их произведение дает $\frac{N}{4}$, или просто добавить в исходный массив необходимое количество лишних чисел.

В дальнейшем будет проводиться работа с текстурными координатами. Рассмотрим два различных вида текстур:

- *GL_TEXTURE_2D* – такие текстуры всегда имеют нормализованные координаты – координаты, лежащие в отрезке $[0; 1]$;
- *GL_TEXTURE_RECTANGLE_ARB* – такие текстуры имеют координаты от нуля до их размерности по оси.

Для удобства расчетов выгоднее пользоваться вторым типом текстур, так как за счет ненормализованных координат можно легко вычислять значения в соседних точках (об этом будет сказано ниже).

OpenGL предоставляет следующий набор инструкций для работы с текстурами:

- *glGenTextures* – создает текстуру и выделяет ей идентификатор;
- *glBindTexture* – устанавливает тип текстуры (*GL_TEXTURE_2D* или *GL_TEXTURE_RECTANGLE_ARB*);
- *glTexParameteri, glTexEnvi* – устанавливают различные целочисленные параметры;
- *glTexImage2D* – задает размеры текстуры и выделяет память под нее.

2.3. Перевод данных между основной памятью и видеопамятью

Для того чтобы перевести данные из массива в текстуру, необходимо воспользоваться функцией *glTexSubImage2D*. Для обратной операции используется *glGetTexImage*.

Данные функции в качестве параметров принимают идентификатор текстуры, указатель на массив в основной памяти, а также тип данных,

которые необходимо прочитать из массива или записать в него. Основными типами данных являются:

- *GL_FLOAT* – массив хранит вещественные числа;
- *GL_UNSIGNED_BYTE* – массив хранит целые числа от 0 до 255 (обычно используется для двумерных изображений, для которых в массиве записываются четверки байт, обозначающих *r*, *g*, *b* и *a* составляющие в диапазоне $[0; 255]$).

Стоит отметить, что операции по переводу данных из основной памяти в видеопамять и наоборот являются относительно дорогостоящими, поэтому пользоваться ими следует как можно реже.

Одним из способов сокращения числа таких операций является *рендеринг в текстуру*. Текстура *привязывается* (*is attached*) при помощи функции *glFramebufferTexture2DEXT* к фреймбуферу, и после проведения вычислений результаты хранятся внутри нее. Ей можно вновь воспользоваться, уже в качестве входных данных. Это может ускорить серию последовательных вычислений. Также отметим, что чтение из привязанной текстуры производится быстрее (необходимо воспользоваться функцией *glReadPixels*).

2.4. Описание вычислений при помощи шейдеров

Рассмотрим процесс использования шейдеров для вычислений на примере применения к вектору из *N* вещественных чисел следующей операции:

$$Y \leftarrow Y + \alpha X .$$

На языке *C* данную операцию можно записать следующим образом:

```
for (int i = 0; i < N; i++)
{
    Y[i] = Y[i] + alpha * X[i];
}
```

Для шейдеров необходимость в цикле пропадает: вычисление фрагментного шейдера происходит в каждой точке текстуры.

Рассмотрим шейдер, вычисляющий результат данной операции:

```
#extension GL_ARB_texture_rectangle : enable

uniform sampler2DRect textureY;
uniform sampler2DRect textureX;
uniform float alpha;

void main(void)
{
    vec4 y = texture2D(textureY, gl_TexCoord[0].st);
    vec4 x = texture2D(textureX, gl_TexCoord[0].st);
    gl_FragColor = y + alpha * x;
}
```

Первая строка включает расширение *OpenGL*, позволяя использовать текстуры типа *GL_TEXTURE_RECTANGLE_ARB*.

Далее указываются объявления переменных. Ключевое слово *uniform* обозначает, что переменная передается в шейдер из кода основной программы. Это можно сделать при помощи функций *glUniform1i* для целочисленных переменных (в том числе и идентификаторов текстур) и *glUniform1f* – для вещественных переменных. В качестве входных параметров эти функции принимают идентификатор переменной (который, в свою очередь, можно получить при помощи функции *glGetUniformLocation*, передав ей имя переменной в виде строки) и значение, которое необходимо записать. В данном примере шейдер получает на вход два идентификатора текстур и вещественное число.

Основная функция имеет сигнатуру *void main(void)*. Она возвращает свое значение путем записи в глобальную переменную *gl_FragColor*, являющуюся вектором из четырех элементов.

Традиционно геометрические координаты имеют обозначения *x*, *y*, *z*, тогда как текстурные – *s*, *t*. Стоит отметить, что язык поддерживает операцию *swizzle*, которая позволяет получать из вектора новый вектор с различными компонентами. Для получения текущих текстурных координат можно использовать компоненты *s* и *t* вектора *gl_TexCoord[0]*. Для того чтобы получить двухкомпонентный вектор с координатами достаточно написать *gl_TexCoord[0].st*.

Для того чтобы получить данные из текстуры, необходимо воспользоваться функцией *texture2D*, принимающей в качестве входных параметров идентификатор текстуры и координаты, по которым требуется считать значение. Здесь проявляется удобство текстур типа *GL_TEXTURE_RECTANGLE_ARB*, так как для того, чтобы, например, получить данные из клетки слева снизу от текущей, достаточно написать:

```
vec4 x = texture2D(textureX, vec2(gl_TexCoord[0].s - 1.0,
gl_TexCoord[0].t + 1.0));
```

Для обычных текстур (*GL_TEXTURE_2D*) значение 1.0 пришлось бы делить на ширину или высоту текстуры.

В остальном программа напоминает язык *C*. Стоит отметить наличие встроенных типов векторов (*vec2*, *vec3*, *vec4*, *ivec2*, *ivec3*, *ivec4*) и матриц (*mat2*, *mat3*, *mat4*, *mat2x3*, и т.д.).

С точки зрения *OpenGL*, *программа* – это набор шейдеров. Их исходный код хранится в памяти в виде текста. Рассмотрим функции работы с шейдерами, предоставляемые *OpenGL*:

- *glCreateProgram* – создает новую программу и присваивает ей идентификатор;
- *glCreateShader* – получает на вход тип шейдера (для фрагментного шейдера – *GL_FRAGMENT_SHADER_ARB*), создает его и присваивает идентификатор;
- *glShaderSource* – связывает шейдер с исходным кодом, получая на вход идентификатор шейдера и указатель на текст с исходным кодом;

- *glCompileShader* – компилирует шейдер; если во время компиляции произошли ошибки, то можно получить их описание при помощи функции *glGetShaderInfoLog*;
- *glAttachShader* – привязывает шейдер к программе;
- *glLinkProgram* – производит линковку всей программы;
- *glUseProgram* – указывает на то, что данная программа будет использована при рендеринге.

2.5. Рендеринг как запуск вычислений

Для того чтобы запустить вычисления, необходимо специальным образом подготовить трехмерную сцену, а затем «нарисовать» ее.

Вначале укажем (при помощи функции *glUseProgram*) необходимый шейдер, установим текстуру, в которую будет записан результат (она должна быть связана с фреймбуфером, воспользуемся функцией *glDrawBuffer*).

Для проведения вычислений будем рисовать прямоугольник с размерами, равными размерам текстуры, в которую будут записываться результаты вычислений. Необходимо, чтобы каждый пиксель изображения соответствовал *текселю* (*texel*, единица текстуры) текстуры, а также, чтобы прямоугольник целиком покрывал область рендеринга.

Для того чтобы сцена удовлетворяла этим требованиям, необходимо выполнить следующий код:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, width, 0.0, height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, width, height);
```

Здесь ***width*** – ширина целевой текстуры, а ***height*** – ее высота.

Следующий код рисует прямоугольник:

```
glPolygonMode(GL_FRONT,GL_FILL);
glBegin(GL_QUADS);
   glTexCoord2f(0.0, 0.0);
   glVertex2f(0.0, 0.0);
   glTexCoord2f(width, 0.0);
   glVertex2f(width, 0.0);
   glTexCoord2f(width, height);
   glVertex2f(width, height);
   glTexCoord2f(0.0, height);
   glVertex2f(0.0, height);
glEnd();
```

Именно после выполнения этих инструкций будут произведены все необходимые вычисления, а их результат будет записан в результирующую текстуру.

Важным моментом является то, что невозможно одновременно читать из текстуры и записывать результат в нее же. Поэтому для приведенного примера с операцией $Y \leftarrow Y + \alpha X$ необходимо создать *две* текстуры: одну для чтения, а другую – для записи. Если вычисления требуется провести несколько раз подряд, можно просто изменять предназначение текстуры (для чтения или для записи) при помощи функции *glDrawBuffer*.

2.6. Достоинства и недостатки традиционного подхода

Отметим следующие достоинства традиционного подхода.

- Данный подход является более-менее стандартизированным: стандартом являются функции, предоставляемые *OpenGL*, некоторые виды текстур, язык *GLSL*.
- Работа происходит на низком уровне, имеется возможность обнаруживать низкоуровневые ошибки и обрабатывать их. Также

низкоуровневость дает возможность создавать оптимизированные программы.

Среди недостатков необходимо отметить следующие.

- К сожалению, часто поведение на различных графических процессорах имеет различия. Например, на разных чипах по-разному работают фреймбуферы, поддерживаются не все виды текстур.
- Существует очень важное ограничение: один шейдер может выдавать только один результат. Из-за этого часто приходится писать сложные программы, состоящие из нескольких шейдеров. При этом код взаимодействия с графическим процессором становится очень сложным и слишком «заточенным» под конкретную задачу. Например, добавление нового шейдера в последовательность вычислений требует значительных усилий.
- Часто программы, состоящие из нескольких шейдеров, неоптимизированны. Оптимизация таких программ требует дополнительных знаний о *GPU*, а также требует больших усилий.

  Решению последних двух проблем и посвящена данная работа.

Выводы по главе 2

- Рассмотрены техники рендеринга в фреймбуфер и рендеринга в текстуру.
- Описано представление данных в текстуре.
- Рассмотрены основные функции взаимодействия *CPU* и *GPU*, приведен пример простейшего шейдера.
- Отмечены достоинства и недостатки традиционного подхода, а также направление данной работы по устранению недостатков.

Глава 3. Оптимизация программ, состоящих из нескольких шейдеров

В этой главе рассматриваются вопросы оптимизации программ, состоящих из нескольких шейдеров. Оптимизация проводится как с точки зрения увеличения скорости, так и с точки зрения уменьшения объема программного кода. Для примера приводится задача из практики автора, имеющая важное прикладное значение.

3.1. Пример задачи

Пусть с неподвижной веб-камеры через заданные промежутки времени считывается изображение. Часть объектов (фон) являются статичными, другие периодически движутся. Требуется определить движущийся объект и «подсветить» его.

Рассмотрим алгоритм решения этой задачи, который более подробно представлен в работе [19].

Предположим, что исходное изображение имеет размеры $M \times N$. Будем хранить три двумерных массива с размерами $M \times N$:

- «изображение» – сюда будет считываться новое изображение с камеры;
- «фон» – изначально инициализируется копией первого кадра, затем будет модифицироваться;
- «результат» – в данном массиве будет храниться результат вычислений, для каждой клетки будет записано, надо ли ее «подсвечивать».

На каждой итерации проведем следующие действия:

- «чтение» – прочитаем новое изображение с камеры в массив «изображение»;

- «фильтрация шумов» – применим некоторую последовательность фильтров к полученному изображению для того, чтобы избавиться от шумов, результат запишем в массив «изображение»;
- «трекинг» – для каждого пикселя изображения рассмотрим его значения в массивах «изображение» (X^t) и «фон» (X), в «результат» запишем:

$$\begin{cases} 1, & \text{если } \sqrt{(X_r^t - X_r)^2 + (X_g^t - X_g)^2 + (X_b^t - X_b)^2} > \delta; \\ 0, & \text{если } \sqrt{(X_r^t - X_r)^2 + (X_g^t - X_g)^2 + (X_b^t - X_b)^2} \le \delta. \end{cases}$$

- «продвижение» – для каждого пикселя изображения обновим его значение в массиве «фон» по следующей формуле:

$$X_{new} = \alpha \cdot X^t + (1 - \alpha) \cdot X.$$

Отметим, что могут быть использованы следующие фильтры:

- «эрозия-минимум» – для каждого пикселя изображения найдем минимальные компоненты цвета среди цвета этого пикселя и его соседей и присвоим их данному пикселю:

1	2	3
4	5	6
7	8	9

$$X_c^5 \leftarrow \min\{X_c^1, X_c^2, X_c^3, X_c^4, X_c^5, X_c^6, X_c^7, X_c^8, X_c^9\};\ c = r, g, b.$$

- «эрозия-максимум» – для каждого пикселя изображения найдем максимальные компоненты цвета среди цвета этого пикселя и его соседей и присвоим их данному пикселю:

$$X_c^5 \leftarrow \max\{X_c^1, X_c^2, X_c^3, X_c^4, X_c^5, X_c^6, X_c^7, X_c^8, X_c^9\};\ c = r, g, b.$$

- «черно-белый» – для каждого пикселя изображения изменим его цвет, чтобы он соответствовал градации серого цвета, по формуле:

$$X_r, X_g, X_b \leftarrow 0.56 * X_r + 0.33 * X_g + 0.11 * X_b.$$

Автором используются фильтры «черно-белый» и «эрозия-среднее»:

$$er_{mid} = \frac{er_{min} + er_{max}}{2}.$$

При этом использование того или иного фильтра контролируется булевской переменной. При включении фильтра уменьшается вероятность ложного результата, возникающего благодаря шумам, но несколько ухудшается точность. Комбинация фильтров, дающая наилучший результат, зависит от камеры и освещенности.

Отметим, что данная задача может быть эффективно решена на графическом процессоре: проводится ряд одинаковых операций над каждым пикселем изображения.

3.2. Неоптимизированное решение

Наилучшим подходом к решению данной задачи может показаться описание каждого фильтра и преобразования отдельным шейдером. Невозможность использовать один шейдер для решения задачи заключается в том, что имеется два массива, в которые необходимо записать результат: «результат» и «фон». Также неясно, каким образом можно описать последовательное применение фильтров, а также условия, при которых применяется тот или иной фильтр. Решение всех этих проблем будет приведено ниже. Рассмотрим неоптимизированное решение задачи, которое было изначально использовано автором.

Шейдер для фильтра «эрозия-минимум» будет выглядеть следующим образом:

```
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect img;

vec4 findMinCol(vec4 c1, vec4 c2, vec4 c3, vec4 c4, vec4 c5,
vec4 c6, vec4 c7, vec4 c8, vec4 c9)
{
     vec4 result = min(c1, min(c2, min(c3, min(c4, min(c5,
min(c6, min(c7, min(c8, c9)))))))));
     return result;
}
```

```
void main(void)
{
     float x = gl_TexCoord[0].s;
     float y = gl_TexCoord[0].t;
     vec4 v1 = texture2DRect(img,  vec2(x - 1.0, y - 1.0));
     vec4 v2 = texture2DRect(img,  vec2(x      , y - 1.0));
     vec4 v3 = texture2DRect(img,  vec2(x + 1.0, y - 1.0));
     vec4 v4 = texture2DRect(img,  vec2(x - 1.0, y      ));
     vec4 v5 = texture2DRect(img,  vec2(x + 1.0, y      ));
     vec4 v6 = texture2DRect(img,  vec2(x - 1.0, y + 1.0));
     vec4 v7 = texture2DRect(img,  vec2(x      , y + 1.0));
     vec4 v8 = texture2DRect(img,  vec2(x + 1.0, y + 1.0));
     vec4 v9 = texture2DRect(img,  vec2(x      , y      ));
     gl_FragColor=findMinCol(v1, v2, v3, v4, v5, v6, v7, v8, v9);
}
```

Код для фильтра «эрозия-максимум» имеет схожую структуру. Фильтр «черно-белый» описывается следующим образом:

```
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect img;
void main(void)
{
     vec4 oldColor   = texture2DRect(img,  gl_TexCoord[0].st);
     vec4 bwVector   = vec4(0.56, 0.33, 0.11, 0.0);
     float newColor  = dot(oldColor, bwVector);
     gl_FragColor    = vec4(newColor, newColor, newColor, 1.0);
}
```

Трекинг движения происходит благодаря выполнению следующего шейдера:

```
#extension GL_ARB_texture_rectangle : enable
uniform float           threshold;
uniform sampler2DRect   img;
uniform sampler2DRect   bg;
void main(void)
```

```
{
     vec4 vimg       = texture2DRect(img, gl_TexCoord[0].st);
     vec4 vbg        = texture2DRect(bg, gl_TexCoord[0].st);
     float diff      = length(vimg - vbg);
     gl_FragColor    = vec4(0.0, 0.0, 0.0, step(treshold, diff));
}
```

Наконец, запись нового значения в массив «фон» происходит следующим образом:

```
#extension GL_ARB_texture_rectangle : enable
uniform float           advance;
uniform sampler2DRect   img;
uniform sampler2DRect   bg;
void main(void)
{
     vec4 vimg       = texture2DRect(img, gl_TexCoord[0].st);
     vec4 vbg        = texture2DRect(bg, gl_TexCoord[0].st);

     gl_FragColor    = mix(vbg, vimg, advance);
     gl_FragColor.a  = 0.0;
}
```

Программа должна последовательно применить необходимые фильтры, затем провести процесс трекинга, а затем записать новое значение фона. Общая схема выполнения программы приведена на рис. 3.

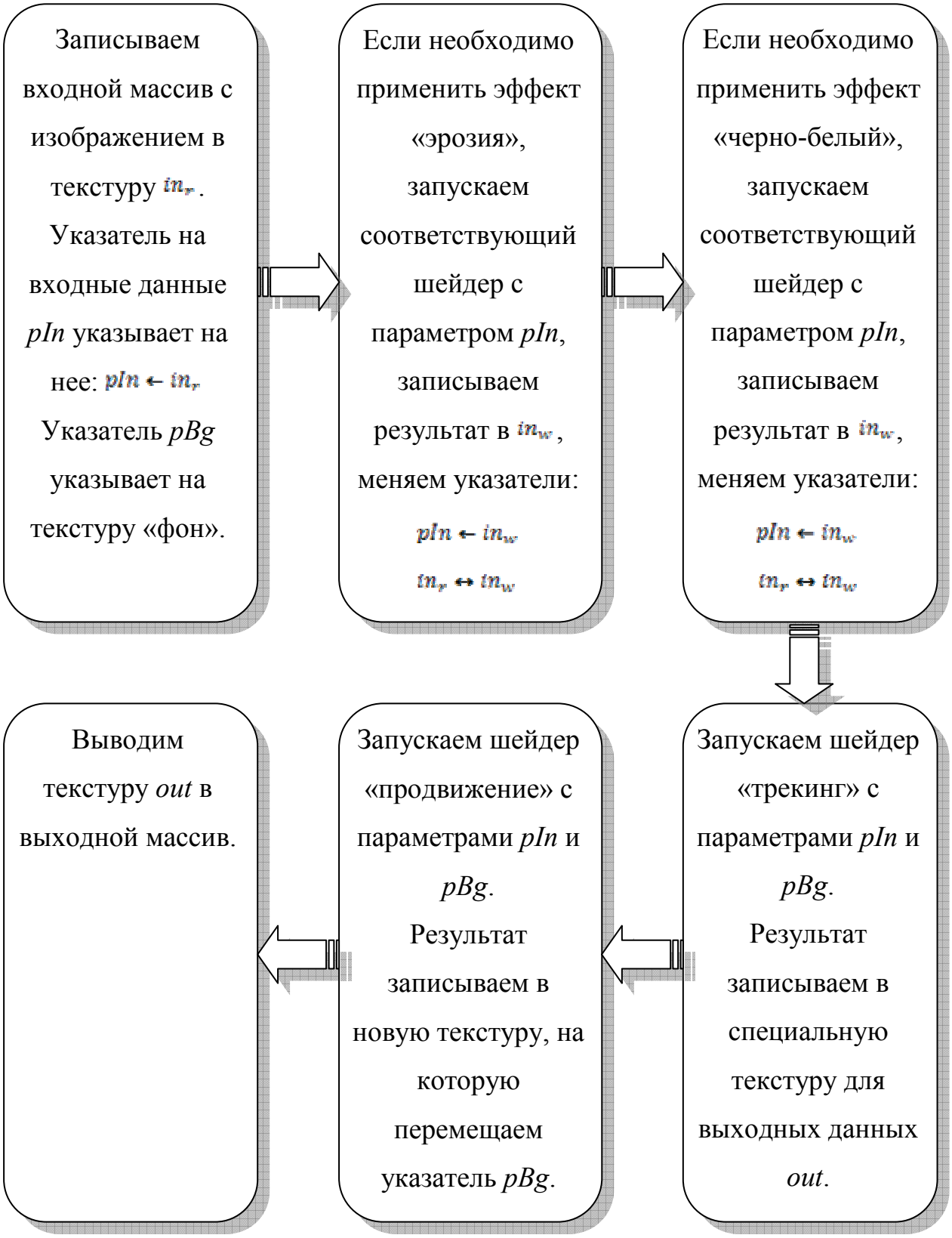

Рис. 3. Схема работы неоптимизированной программы

Как можно видеть, для данного решения необходимо выделить следующий набор текстур.

- Две текстуры для входных данных. При работе фильтров данные считываются из одной и записываются во вторую, что связано с невозможностью произведения операций чтения и записи с одной текстурой. После выполнения шейдера указатели на текстуры для чтения и записи меняются местами.
- Одна текстура для выходных данных. Производится только операция записи. Поэтому наличие пары текстур не требуется.
- Две текстуры для фона. При работе шейдера «продвижение» текущий фон считывается из одной, а результат записывается во вторую. После этого происходит смена указателей.

Стоит отметить, что под указателями в данном примере понимаются указатели на идентификаторы текстур.

В данном решении было получено наличие четырех шейдеров, описывающих отдельные операции. Однако существует возможность уменьшить число шейдеров, ускорив тем самым вычисления: будет требоваться меньшее число запусков процедуры рендеринга.

3.3. Объединение последовательности шейдеров

Типичный *GPGPU*-шейдер получает данные из текстуры по некоторым координатам, проводит над ними некоторые операции, а затем выводит результат в новую текстуру по некоторым координатам. Таким образом, состояние в ячейке новой текстуры зависит от состояния ячейки старой текстуры. При этом не равным единице может быть как число текстур, для которых проводится операция чтения, так и число ячеек, из которых считываются данные конкретной текстуры.

Выделим отдельный класс шейдеров – *шейдеры с единичной зависимостью*. Шейдер данного класса удовлетворяет следующему утверждению: для каждой входной текстуры состояние новой ячейки зависит от состояния только одной ячейки этой текстуры. Наилучшей с точки зрения

оптимизации является зависимость от ячейки с теми же координатами, что и у ячейки для записи.

Примером представителя этого класса является приведенный выше шейдер «черно-белый». На цвет пикселя нового изображения влияет только цвет пикселя старого. Единственное считывание данных выполняется инструкцией:

```
vec4 oldColor   = texture2DRect(img,  gl_TexCoord[0].st);
```

В случае с шейдером «эрозия» на состояние новой ячейки влияет не только та же ячейка в старой текстуре, но и все ее соседи. Это подтверждается необходимостью нескольких операций считывания данных:

```
vec4 v1 = texture2DRect(img,  vec2(x - 1.0, y - 1.0));
vec4 v2 = texture2DRect(img,  vec2(x       , y - 1.0));
```

и т.д.

Рассмотрим последовательное применение некоторого шейдера, а затем другого шейдера, обладающего свойством единичной зависимости. Первый шейдер записывает данные в некоторую текстуру, необходимую только для того, чтобы второй шейдер прочитал из нее данные (рис. 4).

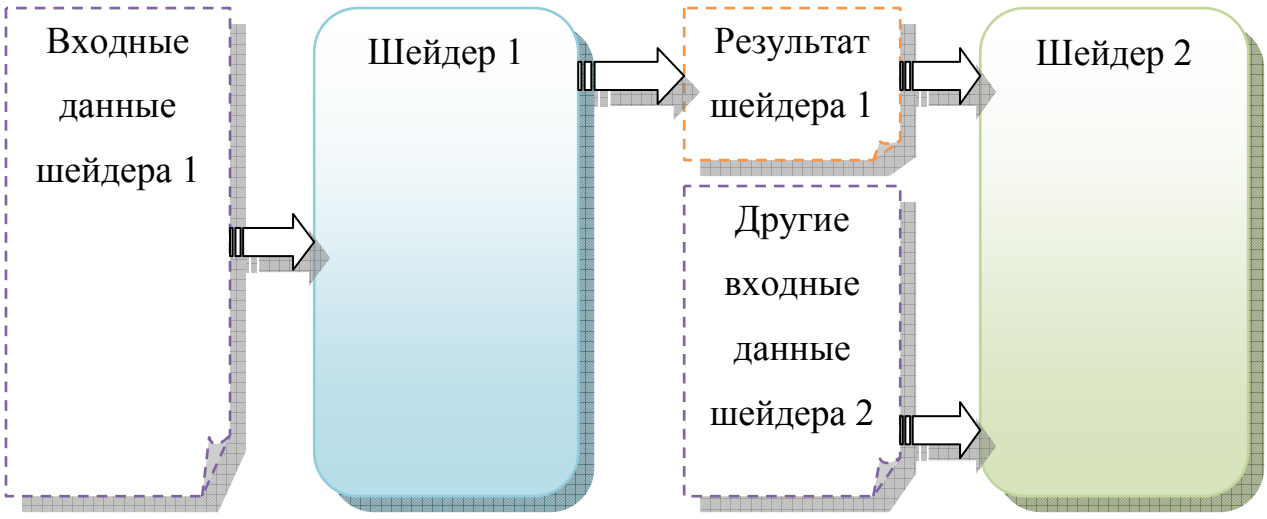

Рис. 4. Схема последовательного применения двух шейдеров

Существует возможность замены такой пары шейдеров одним. При этом не только уменьшится число шейдеров, но и пропадет необходимость в

наличии промежуточной текстуры. Требуется создать объединенный шейдер со структурой, приведенной на рис. 5.

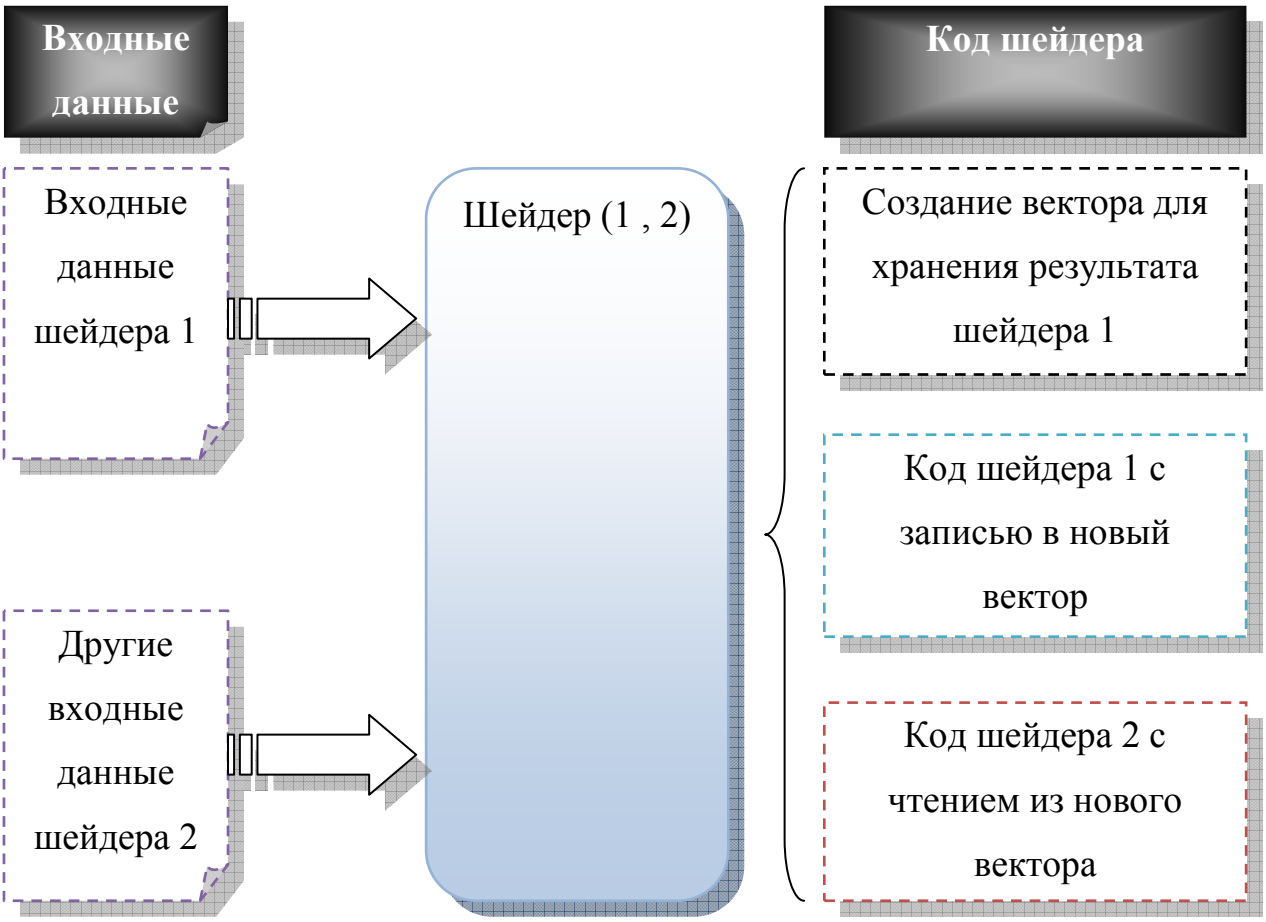

Рис. 5. Схема объединения двух шейдеров

Код нового шейдера будет состоять из последовательности из кода первого и второго шейдеров с некоторыми модификациями.

- Создадим новый вектор из четырех элементов, в котором будет храниться результат вычислений первого шейдера:

  ```
  vec4 result_1(0.0, 0.0, 0.0, 0.0);
  ```

- Заменим все вхождения *gl_FragColor* в коде первого шейдера на имя нового вектора:

  ```
  gl_FragColor.r = 1.0;        →        result_1.r = 1.0;
  ```

- В коде второго шейдера заменим обращение к текстуре с результатом первого шейдера на доступ к новому вектору:

```
vec4 oldColor   = texture2DRect(img,  gl_TexCoord[0].st);
                        ↓
vec4 oldColor   = result_1;
```

Отметим, что важен порядок применения шейдеров. Например, в случае применения сначала фильтра «эрозия», а затем фильтра «черно-белый», оптимизация возможна – именно второй шейдер, «черно-белый», обладает свойством единичной зависимости и может быть объединен с шейдером «эрозия». В противном случае оптимизация становится невозможной – шейдеру «эрозия» необходимы результаты шейдера «черно-белый» не только в той же ячейке, но и в ее соседях. Возможность объединения шейдеров все же имеется: внутри объединенного шейдера можно провести вычисления фильтра «черно-белый» для каждой соседней клетки. Однако при этом для каждой клетки вычисления будут в сумме проведены *девять* раз вместо одного, что даст проигрыш в скорости.

К сожалению, чаще всего порядок применения шейдеров фиксирован, и оптимизация в таких случаях невозможна. Однако если возможность смены порядка имеется, следует ей пользоваться.

3.4. Оптимизация арифметических операций

Арифметические операции: сложение, вычитание, умножение и деление – часто применяются для композиции нескольких фильтров, например, для нахождения их среднего арифметического.

Рассмотрим схему применения среднего арифметического фильтров к изображению (рис. 6).

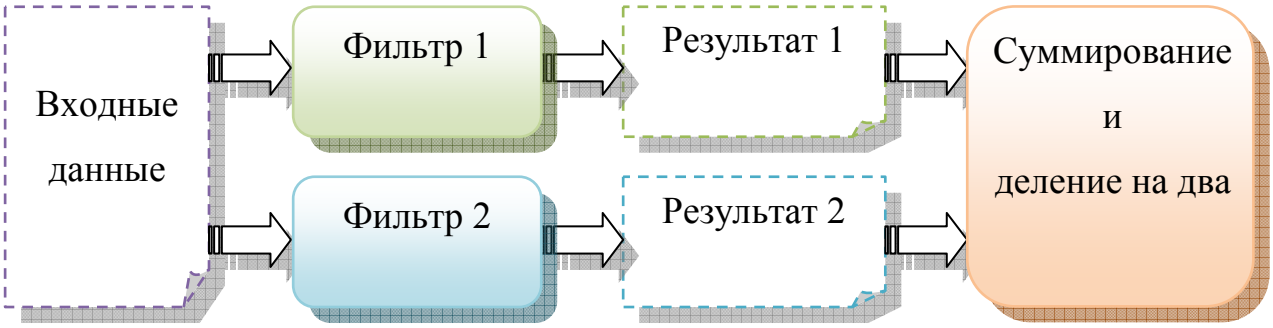

Рис. 6. Схема получения среднего двух фильтров

В данном случае суммирование и деление могут производиться как на *CPU*, так и на *GPU* при помощи отдельных шейдеров, использующих стандартные операторы, предоставляемые языком *GLSL* для описания арифметических операций.

Заметим, что при выражении арифметических операций при помощи шейдеров данные шейдеры будут обладать свойством единичной зависимости: арифметические операции применяются покомпонентно, и зависимости от других ячеек нет. Более того отметим, что фильтры 1 и 2 независимы друг от друга, а шейдеры, реализующие их, не содержат операций доступа к результатам другого фильтра. Таким образом, существует возможность замены всей данной схемы на единичный шейдер (рис. 7).

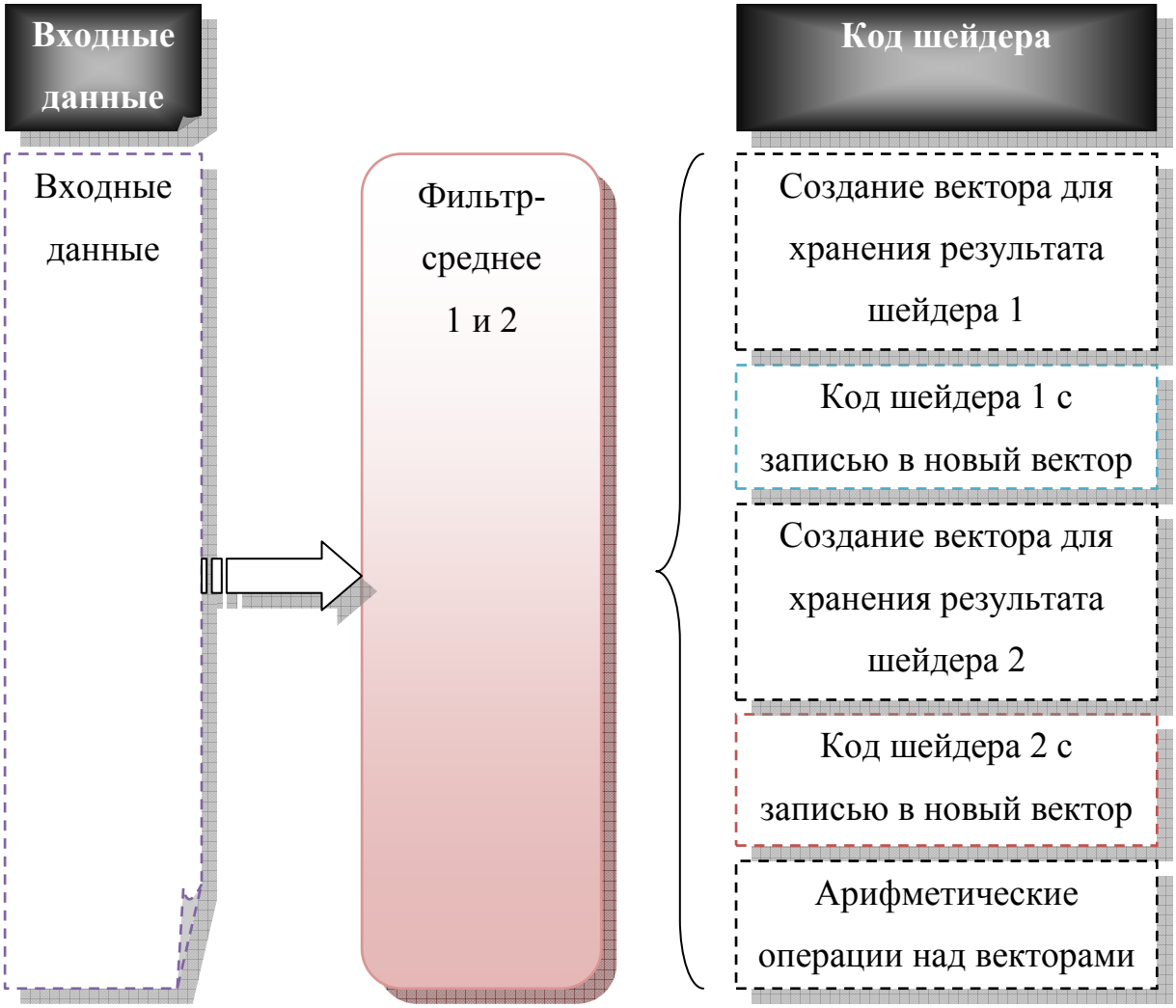

Рис. 7. Схема единичного шейдера, описывающего среднее двух фильтров

В данном случае в коде шейдеров, описывающих фильтры 1 и 2, необходимо создать новые векторы, которые будут хранить результаты этих шейдеров, а также заменить вхождения *gl_FragColor* на имя соответствующего вектора. Далее остается только провести необходимые арифметические операции над новыми векторами при помощи стандартных операторов языка *GLSL* и записать результат в вектор *gl_FragColor*.

Благодаря данному свойству легко реализуется, например, фильтр «эрозия-среднее»: шейдеры «эрозия-максимум», «эрозия-минимум» и арифметические операции легко преобразуются в один шейдер.

3.5. Одновременная запись нескольких результатов

Рассмотрим операции «трекинг» и «продвижение» в предложенной задаче. Они принимают одинаковые входные данные: текстуры «изображение» и «фон». Проблемой является то, что шейдер «трекинг» выдает выходной результат программы, а шейдер «продвижение» записывает новое значение в текстуру «фон». Здесь вновь приходится столкнуться с важным ограничением при вычислениях на графическом процессоре: невозможно при помощи одного шейдера провести запись в несколько текстур. Если бы существовала возможность одновременной записи нескольких результатов, то данные шейдеры можно было бы объединить.

Как уже было отмечено выше, каждая ячейка текстуры является вектором из четырех элементов. Однако зачастую при вычислениях используются не все из них. Например, для фильтров, работающих с *RGB*-изображениями, используются только три компонента вектора из четырех. Шейдер «трекинг» вообще использует только один *канал* (*channel*) – компонент текстуры (рис. 8).

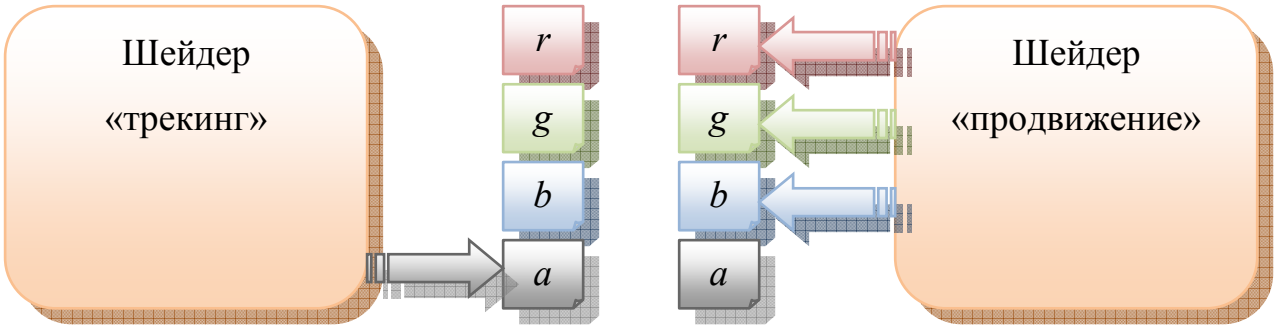

Рис. 8. Используемые каналы текстур для шейдеров «трекинг» и «продвижение»

В данном примере видно, что шейдер «продвижение» использует три текстурных канала, соответствующих компонентам цвета *RGB*-изображения, тогда как шейдер «трекинг» использует только один канал, так как от него требуется только установка флага, свидетельствующего о наличии или отсутствии движения в данной точке. Используемые каналы двух таких шейдеров не пересекаются, поэтому имеется возможность объединить две различные текстуры в одну. После проведения данной операции шейдеры

«трекинг» и «продвижение» можно будет объединить в один. При этом отметим, что шейдер «трекинг» должен использовать именно альфа-канал для записи своего результата, иначе произойдет пересечение каналов, используемых двумя шейдерами, и оптимизация станет невозможной.

Рассмотрим схему объединенного шейдера (рис. 9).

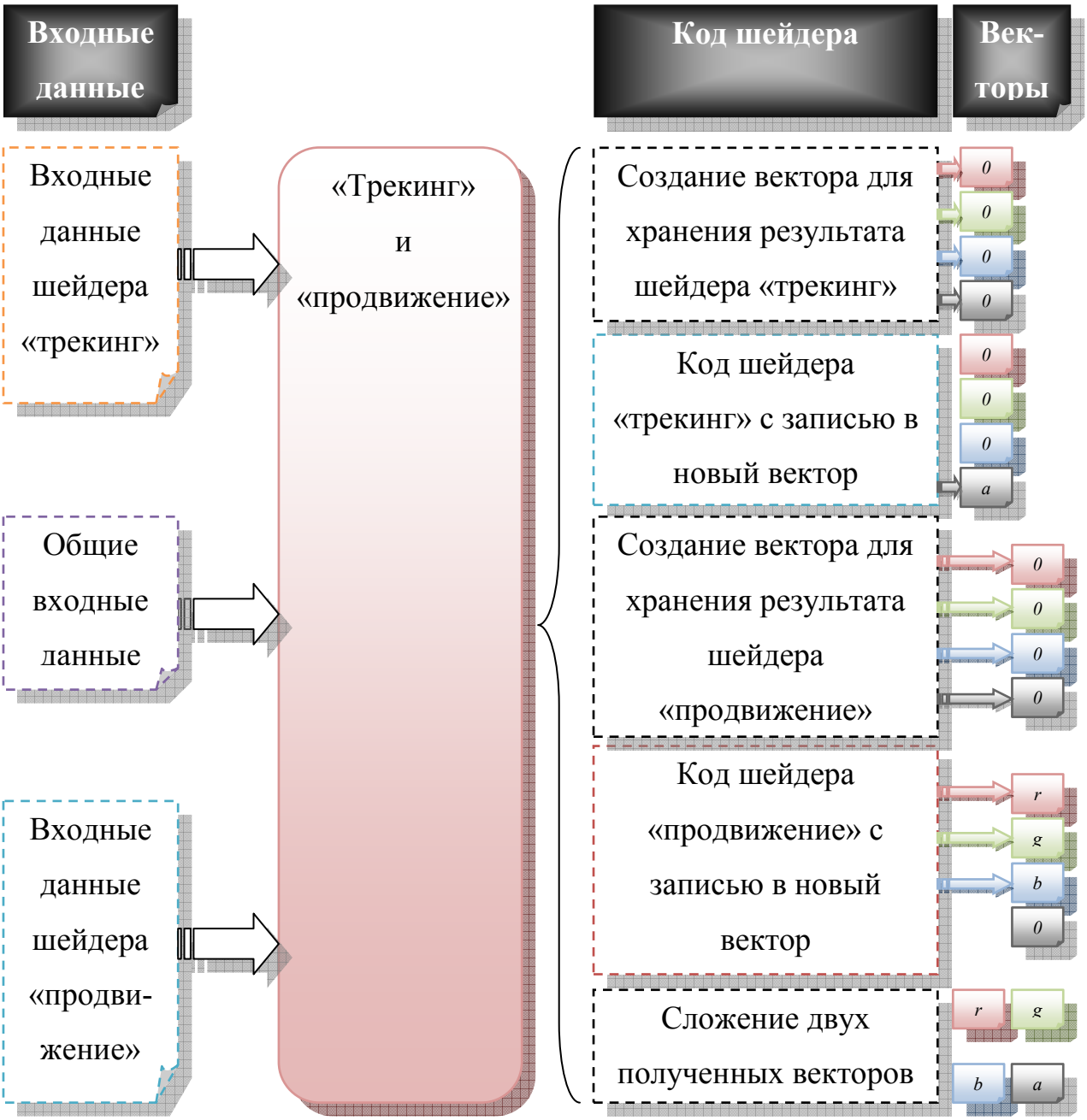

Рис. 9. Объединенный шейдер «трекинг и продвижение»

Отметим, что после получения результатов двух шейдеров для записи общего результата достаточно произвести сложение полученных векторов: они ортогональны, и их сумма раскладывается на слагаемые единственным

образом. Следует заметить, что при таком объединении придется некоторым образом изменить обработку массивов после копирования в них данных выходной текстуры.

3.6. Представление условных операторов

В рассмотренной выше задаче, как уже было сказано ранее, последовательность применяемых фильтров может варьироваться. Напомним, что на различных камерах эффект от применения или неприменения фильтров может быть различным. Поэтому целесообразно ввести условные операторы, чтобы можно было контролировать используемые фильтры.

Последовательность условных операций для данной задачи представлена на рис. 10.

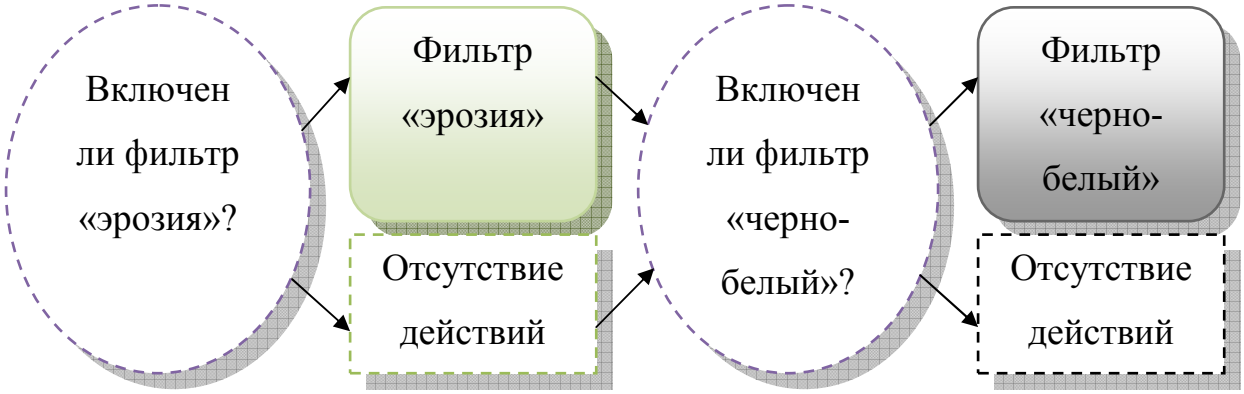

Рис. 10. Последовательность условных операций для предложенной задачи

Отметим, что вместо узла «отсутствие действий» в условном переходе может стоять другой фильтр. Вообще узел «отсутствие действий» можно представить в виде простейшего шейдера, выполняющего следующую операцию:

```
gl_FragColor   = texture2DRect(in,  gl_TexCoord[0].st);
```

В большинстве случаев операция условного перехода сильно затрудняет оптимизацию – требуется отдельно обрабатывать каждую его ветвь.

Первый подход к оптимизации условных операций утверждает, что оказывается выгодным представить как можно большую часть программы в виде единого условия и нескольких ветвей выполнения. Это позволяет проводить дальнейшие оптимизации. Отметим несколько приемов, позволяющих выполнить данное действие.

- Объединение нескольких идущих подряд условных операторов в один. Для нового оператора число ветвей выполнения будет равно произведению числа ветвей в двух старых операторах, поэтому не стоит выполнять данную операцию многократно. Данная операция может быть полезна, например, если внутри ветвей содержатся шейдеры с единичной зависимостью.
- Внесение безусловной операции в каждую ветвь условного оператора. Например, выгодно внести внутрь условного оператора шейдер с единичной зависимостью – это может существенно оптимизировать часть программы внутри условного оператора.

Для предложенной задачи оптимизированную схему можно увидеть на рис. 11. На данной схеме уже удалены все узлы с отсутствием операций.

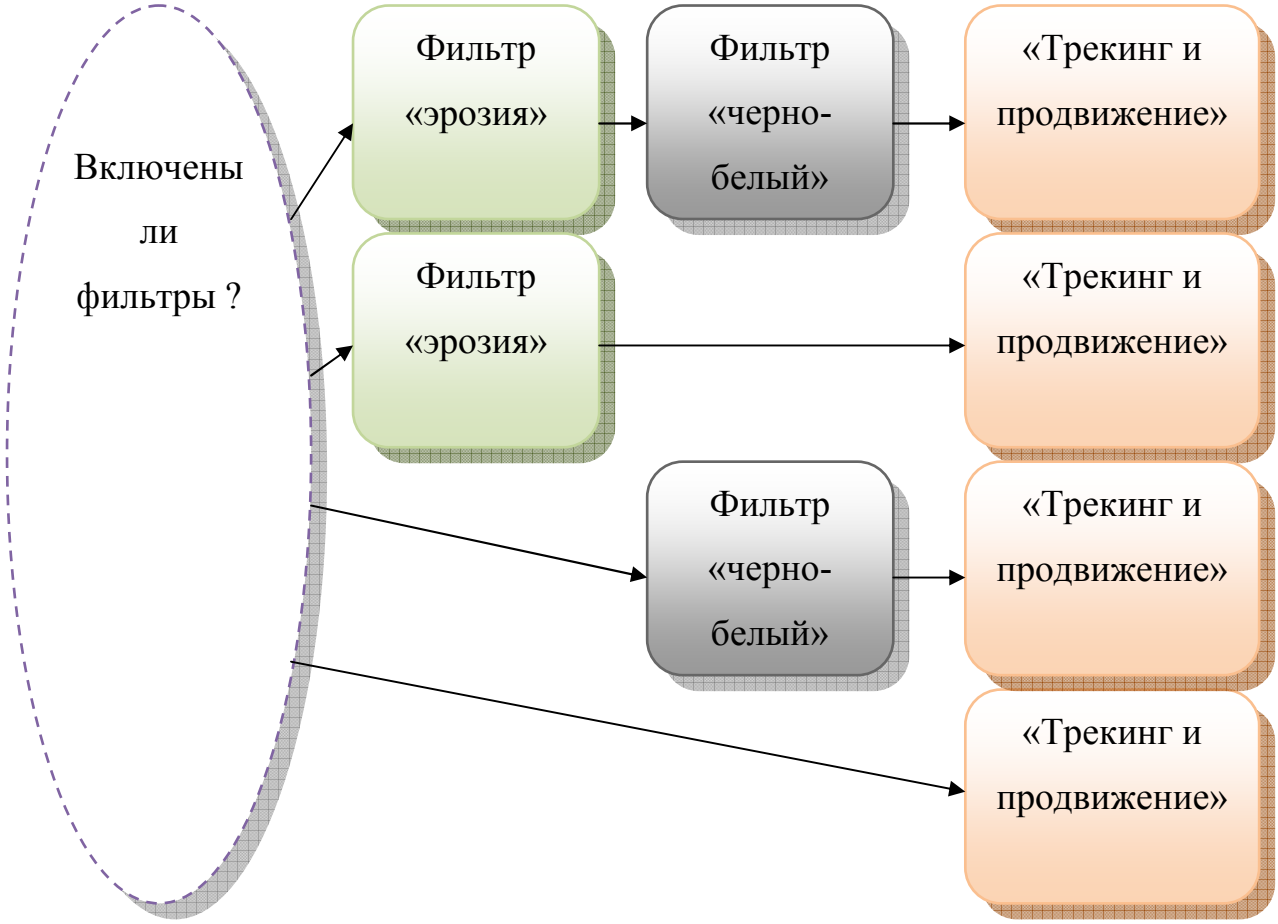

Рис. 11. Схема программы с оптимизированным условным оператором

Стоит отметить, что для данной схемы можно продолжить оптимизацию, объединив фильтры с шейдером «трекинг и продвижение».

Существует также второй подход, который позволяет полностью избавиться от условных операций при наличии только одного шейдера в каждой ветви выполнения. Суть его заключается в том, чтобы заключить само условие внутрь шейдера. К сожалению, условные операторы в шейдерах поддерживаются, начиная с *Shader Model 3*, что делает невозможным их использование на графических процессорах, отвечающих более старым версиям стандарта. Более того условные операторы могут значительно замедлить выполнение программы – некоторые процессоры предугадывают будущие инструкции, что затруднительно в условиях наличия условных операторов. Тем не менее, существует способ представления данных операторов и в контексте *Shader Model 2*.

Воспользуемся стандартной функцией *mix*, которая проводит следующую операцию:

$$mix(u, v, \alpha) = u \cdot (1 - \alpha) + v \cdot \alpha$$

Проведем вычисления для каждой ветви, записав их результаты в два вектора, а затем получим конечный, применив к ним данную функцию – в зависимости от требуемой ветви выполнения коэффициент α будет равен либо нулю, либо единице.

Пример шейдера, полученного в результате избавления от условного оператора, приведен на рис. 12.

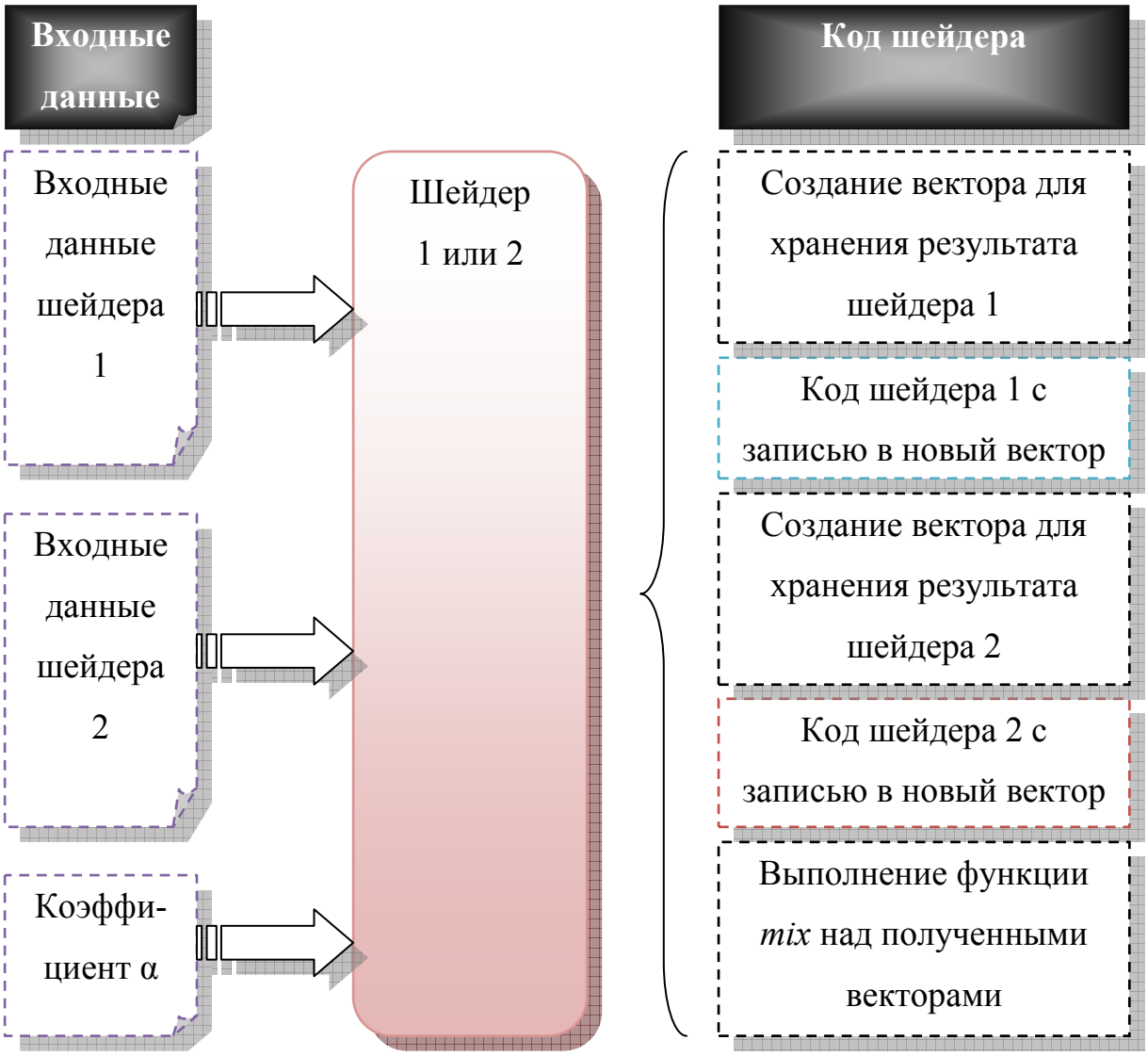

Рис. 12. Схема шейдера, полученного после избавления от условного перехода

Следует заметить, что в случае, когда одна из ветвей содержит узел «отсутствие операции», данный подход не приводит к потере производительности в половине случаев. В других случаях потеря присутствует, но обычно она незначительна. Однако без избавления от условных операторов приходится поддерживать несколько шейдеров, что может быть невыгодно по сравнению с небольшой потерей производительности. Также важно то, что если оба шейдера обладают свойством единичной зависимости (по одинаковым координатам), то и полученный шейдер сохранит это свойство.

3.7. Результаты оптимизации

За счет последовательного применения предложенных техник оптимизации можно добиться значительных результатов. Например, решение предложенной задачи сводится до одного шейдера. Приведем последовательность применяемых оптимизаций, дающих такой результат.

- Избавимся от условного оператора для фильтра «эрозия».
- Избавимся от условного оператора для фильтра «черно-белый».
- Объединим полученные фильтры в один. Данная операция возможна благодаря тому, что фильтр «черно-белый или ничего» обладает свойством единичной зависимости.
- Объединим текстуры «результат» и «фон» в одну. Данная операция возможна благодаря использованию в этих текстурах различных каналов.
- Объединим шейдеры «трекинг» и «продвижение», потому что они теперь производят запись в одну текстуру.
- Произведем слияние объединенного шейдера фильтров с шейдером «трекинг и продвижение». Данная операция возможна благодаря тому, что фильтр «трекинг и продвижение» обладает свойством единичной зависимости.

В итоге получится шейдер со следующим кодом:

```
#extension GL_ARB_texture_rectangle : enable

// Включать ли фильтр "эрозия"
uniform float enableER;
// Входное изображение
uniform sampler2DRect img;
// Включать ли фильтр "черно-белый"
uniform float enableBW;
// Порог определения движения
uniform float threshold;
// Фон
uniform sampler2DRect bg;
// Коэффициент продвижения
uniform float advance;

vec4 findMinCol(vec4 c1, vec4 c2, vec4 c3, vec4 c4, vec4 c5,
vec4 c6, vec4 c7, vec4 c8, vec4 c9)
{
vec4 result = min(c1, min(c2, min(c3, min(c4, min(c5, min(c6,
min(c7, min(c8, c9)))))));
return result;
}

vec4 findMaxCol(vec4 c1, vec4 c2, vec4 c3, vec4 c4, vec4 c5,
vec4 c6, vec4 c7, vec4 c8, vec4 c9)
{
vec4 result = max(c1, max(c2, max(c3, max(c4, max(c5, max(c6,
max(c7, max(c8, c9)))))));
return result;
}
```

```
void main(void)
{
// Код фильтра "эрозия"
float x = gl_TexCoord[0].s;
float y = gl_TexCoord[0].t;
vec4 v1 = texture2DRect(img, vec2(x - 1.0, y - 1.0));
vec4 v2 = texture2DRect(img, vec2(x , y - 1.0));
vec4 v3 = texture2DRect(img, vec2(x + 1.0, y - 1.0));
vec4 v4 = texture2DRect(img, vec2(x - 1.0, y ));
vec4 v5 = texture2DRect(img, vec2(x + 1.0, y ));
vec4 v6 = texture2DRect(img, vec2(x - 1.0, y + 1.0));
vec4 v7 = texture2DRect(img, vec2(x , y + 1.0));
vec4 v8 = texture2DRect(img, vec2(x + 1.0, y + 1.0));
vec4 v9 = texture2DRect(img, vec2(x , y ));
// Результат применения фильтра "эрозия-минимум"
vec4 result_er = findMinCol(v1,v2,v3,v4,v5,v6,v7,v8,v9);
// Результат применения фильтра "эрозия-максимум"
vec4 result_er2 = findMaxCol(v1,v2,v3,v4,v5,v6,v7,v8,v9);
// Среднее арифметическое
result_er_er2 = (result_er + result_er2) / 2.0;
// Случай неприменения фильтра "эрозия"
vec4 result_0 = texture2DRect(img, gl_TexCoord[0].st);
// Итоговый результат в зависимости от условия
vec4 result_er_final = mix(result_er_er2, result_0, enableER);

// Код фильтра "черно-белый"
vec4 bwVector = vec4(0.56, 0.33, 0.11, 0.0)
float newColor = dot(result_er_final, bwVector)
// Результат применения фильтра "черно-белый"
vec4 result_bw = vec4(newColor, newColor, newColor, 1.0);
// Случай неприменения фильтра "черно-белый"
vec4 result_1 = result_er_final;
// Итоговый результат в зависимости от условия
vec4 result_filtered = mix(result_bw, result_1, enableBW);
```

```
// Код шейдера "трекинг"
// Получение цвета фона
vec4 vbg = texture2DRect(bg, gl_TexCoord[0].st);
vbg.a = 0.0;
float diff = length(vimg - vbg);
// Резльтат шейдера "трекинг"
vec4 result_tr = vec4(0.0, 0.0, 0.0, step(treshold, diff));
// Код шейдера "продвижение"
vec4 result_ad = mix(vbg, result_filtered, advance);
result_ad.a = 0.0;
// Суммирование векторов
gl_FragColor = result_tr + result_ad;
}
```

Таким образом, программа, представленная на рис. 3, сведется к оптимизированной программе, представленной на рис. 13.

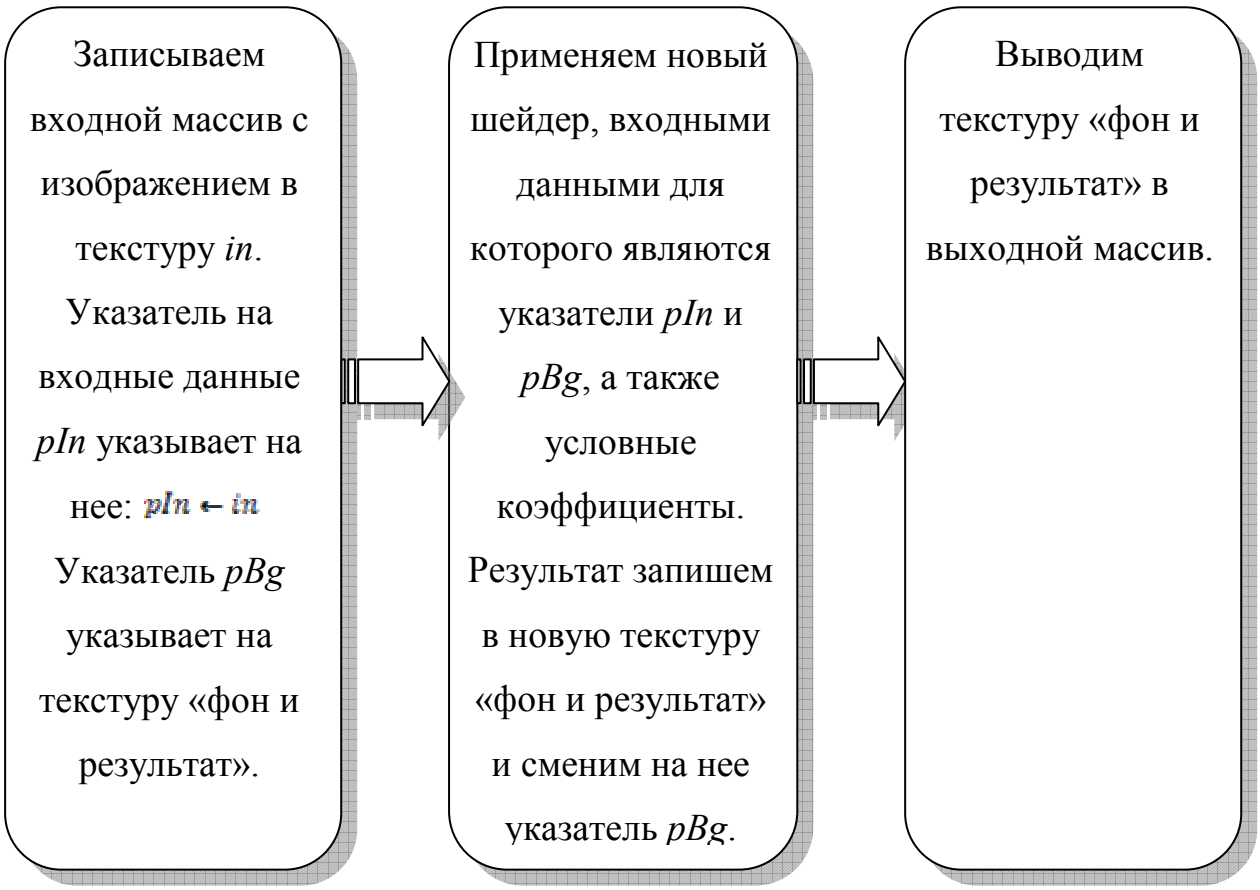

Рис. 13. Схема работы оптимизированной программы

Выводы по главе 3

- Рассмотрено неоптимизированное решение предложенной задачи, выявлены его особенности.
- Приведены и обоснованы техники оптимизации, показана возможность их применения к предложенной задаче.
- Получен шейдер, эквивалентный исходному решению.

Глава 4. Обзор программного средства «*GPGPU-Optimizer*»

В этой главе будет проведен обзор программного средства «*GPGPU-Optimizer*», предложенного автором. Данное средство позволяет автоматически провести оптимизацию записанной в специальном формате программы и вывести ее в виде конечного автомата.

4.1. Составляющие части *GPGPU*-программы

Прежде чем переходить к рассмотрению представления программы в формате, предложенном автором, хотелось бы отметить основные составляющие части *GPGPU*-программы. Итак, *GPGPU*-программу можно условно представить при помощи следующих множеств:

- множество шейдеров, выполняющих вычисления;
- множество переменных, управляющих вычислениями;
- множество данных, над которыми проходят вычисления и в которые заносятся их результаты;
- множество инструкций, которые запускают тот или иной шейдер, предоставляют ему на вход те или иные данные и выводят результат в некоторую текстуру.

4.2. Абстракции

Отметим, что каждая инструкция выполняет одну операцию и выводит результат в новую текстуру. К сожалению, данный подход не всегда удобен: часто требуется провести серию операций, занося промежуточные результаты в некоторые временные переменные, от которых оптимизатор, возможно, сможет отказаться. Например, серию применения фильтров в предложенной выше задаче удобно было бы записать в виде единой функции *filterInput*.

Другой проблемой является то, что часто требуется вызывать схожую серию операций. Хотелось бы уметь описывать эту часто используемую серию в виде единой функции. Например, в предложенной задаче дважды используется условное применение фильтров, и хотелось бы иметь функцию *applyIf*, которая позволяла бы вызывать некий фильтр только при определенном условии. Другим примером является создание универсальной функции *average*, которая находила бы среднее двух фильтров.

Для решения этих проблем автором была предложена возможность вводить *абстракции* – описания новых функций на основе старых, благодаря которым проблемы, обозначенные выше, легко решаются.

Например, описание абстракции *applyIf* выглядит следующим образом:

```
def applyIf<c,f> = if<c,f,nop>
```

В скобках записываются аргументы абстракции, а после знака равенства – ее определение. В данном примере используются встроенные функции *if* и *nop*.

Абстракция *average* описывается так, как представлено ниже:

```
def avg<f1,f2> = div2(add(f1,f2))
```

Здесь используются встроенные арифметические функции *add* и *div2*.

4.3. Запись *GPGPU*-программы в формате *GProject*

Ознакомимся с форматом, предложенным автором для записи *GPGPU*-программы. Данный формат позволяет получить достаточно информации для проведения оптимизации. Вся программа записывается в одном текстовом файле, состоящем из нескольких секций.

- В начале объявляются используемые в *GPGPU*-программе шейдеры. Объявление производится следующим образом:

  ```
  program    bw    <bw.shader>
  ```

  Оно состоит из следующих частей:

  - ключевое слово *program*;

- имя шейдера;
- название файла с исходным кодом шейдера.

- Далее происходит объявление используемых переменных:

```
var    float    trackTreshold
```

Оно состоит из трех частей:

- ключевое слово *var*;
- тип переменной (*bool* или *float*);
- имя переменной.

- В следующей секции объявляются используемые текстуры:

```
data    rgb    input    in
```

Здесь описываются:

- ключевое слово *data*;
- используемые текстурные каналы (например, *rgb* или *a*);
- тип текстуры:
  - *input* – для текстур с входными данными;
  - *internal* – для внутренних текстур;
  - *output* – для текстур с выходными данными;
- имя текстуры.

- Далее описываются используемые абстракции:

```
def advancedER<> = avg<er,er2>
```

Описание строится по следующему формату:

- ключевое слово *def*;
- имя абстракции;
- аргументы абстракции через запятую в скобках;
- знак равенства;
- определение абстракции.

- Ключевое слово *begin* обозначает начало программы.
- Далее записываются инструкции:

```
out = tr(trackTreshold,in,bg)
```

Инструкция состоит из четырех частей:

- имя выходной текстуры;
- знак равенства;
- имя выполняемой функции (шейдер, абстракция или встроенная функция);
- аргументы функции в скобках через запятую.

- Программа завершается ключевым словом *end*.

Запишем программу для задачи, предложенной в качестве примера в данной работе:

```
program     bw  <bw.shader>
program     er  <er.shader>
program     er2 <er2.shader>
program     tr  <tr.shader>
program     ad  <ad.shader>

var    bool     enableBW
var    bool     enableER
var    float    trackTreshold
var    float    advanceValue

data  rgb  input      in
data  rgb  internal   bg
data  a    output     out

def applyIf<c, f> = if<c,f,nop>
def avg<f1,f2> = div2(add(f1,f2))

def advancedER<> = avg<er,er2>

def filterInput<> = applyIf<enableBW, bw>(applyIf<enableER,
advancedER>)

begin
```

```
    in  = filterInput(in)
    out = tr(trackTreshold,in,bg)
    bg  = ad(advanceValue,in,bg)
end
```

После сохранения текстового файла для запуска оптимизатора достаточно предоставить имя этого файла в качестве параметра.

4.4. Схема работы оптимизатора

Процесс работы оптимизатора можно разделить на несколько этапов, которые приведены на рис. 14.

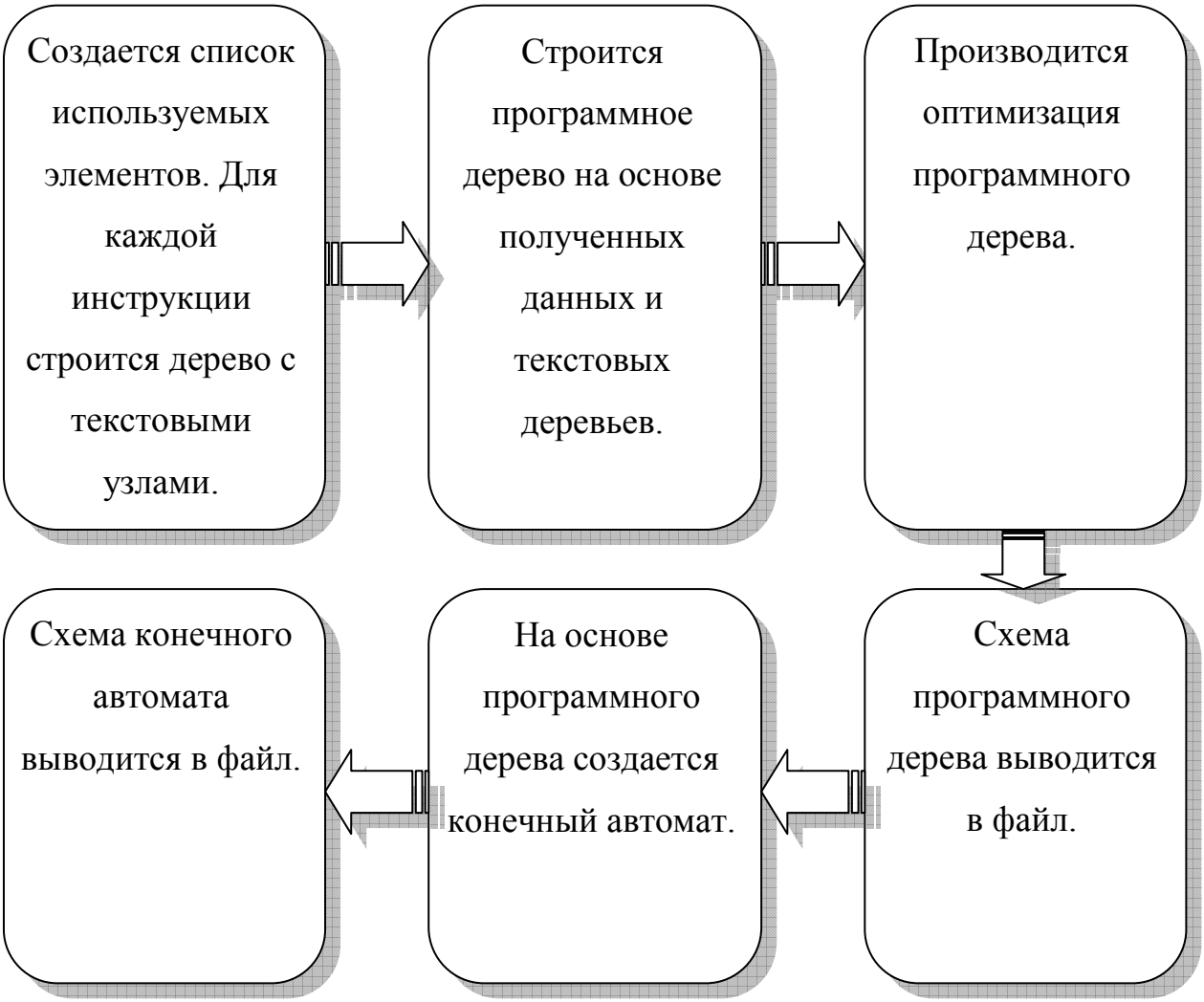

Рис. 14. Схема работы оптимизатора

В качестве выходных данных программа предоставляет схемы оптимизированного программного дерева и сгенерированного автомата, а также полученные шейдеры.

Программное дерево состоит из узлов различных типов, каждый из которых описывает поведение программы:

- корневой узел;
- узел с данными;
- узел с переменной;
- узел с отсутствием операций;
- узел с шейдером;
- узел с арифметической операцией;
- узел с началом ветвления;
- узел с завершением ветвления.

Автомат представляет собой средство описания последовательности действий, необходимых для выполнения программы. Каждое его состояние – это либо выполнение шейдера, либо специальное служебное состояние, необходимое, например, для начала ветвления.

Выходные схемы генерируются при помощи свободно-распространяемой библиотеки *GraphViz* [20].

4.5. Опции оптимизации

Программа позволяет проводить гибкую манипуляцию опциями оптимизации. Можно как отключить оптимизацию целиком, так и работать с отдельными ее методами.

Настройки задаются через текстовый файл *optimizer.cfg*. Рассмотрим предлагаемые опции:

- *opt_enable* – включает или отключает оптимизацию целиком;
- *opt_nop* – включает или отключает удаление узлов с отсутствием операций;

- *opt_ifif* – включает или отключает слияние двух идущих подряд условных операторов;
- *opt_shadershader* – включает или отключает слияние двух идущих подряд шейдеров;
- *opt_eliminateif* – включает или отключает избавление от условных операторов;
- *opt_arithmetics* – включает или отключает оптимизацию арифметических операций;
- *opt_channels* – включает или отключает слияние текстур с различными используемыми каналами;
- *opt_interimdata* – включает или отключает избавление от промежуточных текстур;
- *opt_aggressiveif* – включает или отключает внесение безусловных операций внутрь условного оператора;
- *opt_maxinstructions* – устанавливает максимальное количество инструкций, из которых может состоять шейдер; необходимость в данной опции связана с тем, что графический процессор накладывает ограничения на это количество [21].

4.6. Пример работы оптимизатора

Рассмотрим схемы, которые генерируются оптимизатором для обозначенной выше задачи. Для сравнения приведены как оптимизированные, так и неоптимизированные схемы деревьев и автоматов.

Неоптимизированное дерево представлено на рис. 15.

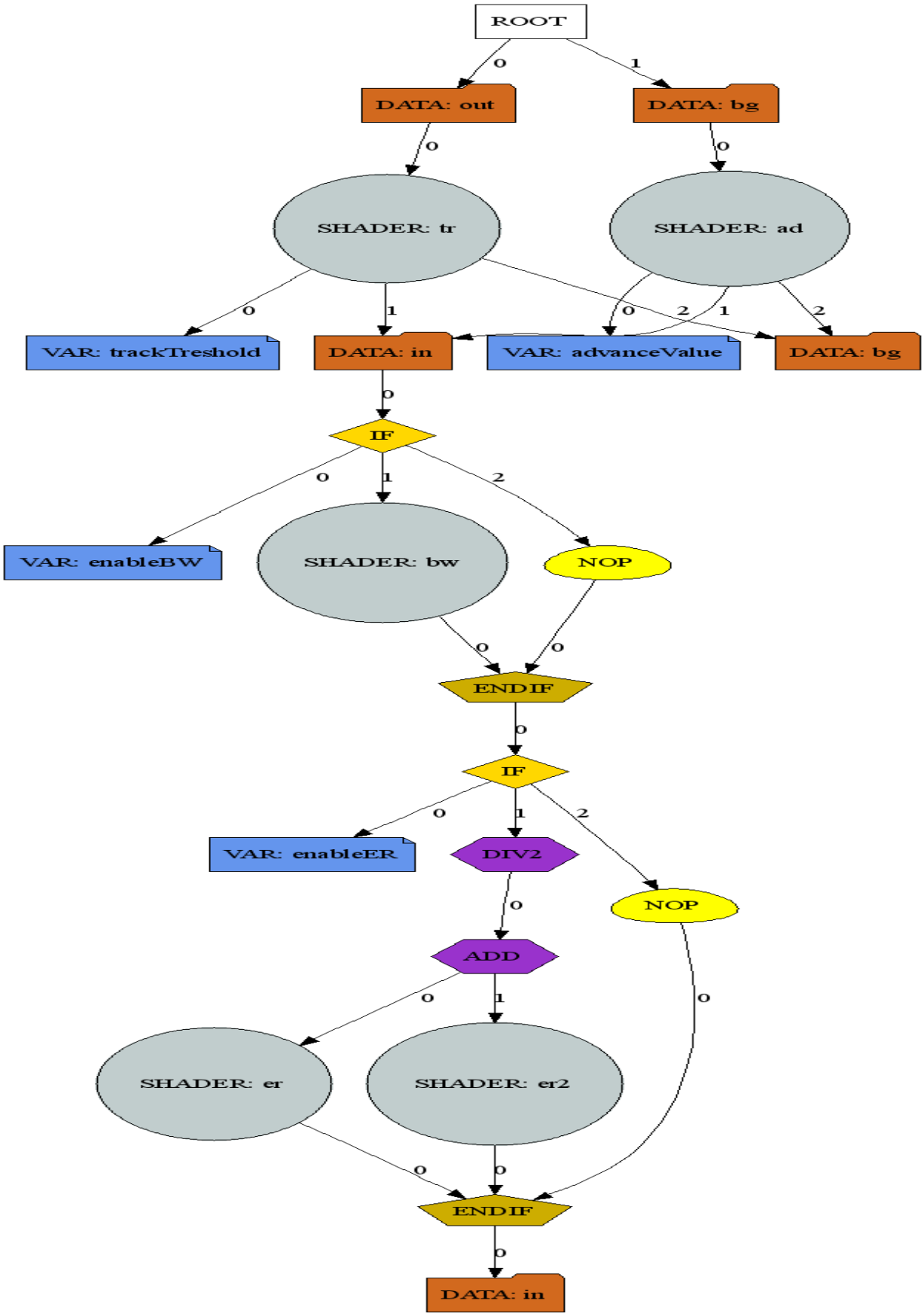

Рис. 15. Схема неоптимизированного программного дерева для предложенной задачи

Оптимизированное дерево (включены все опции оптимизации) представлено на рис. 16.

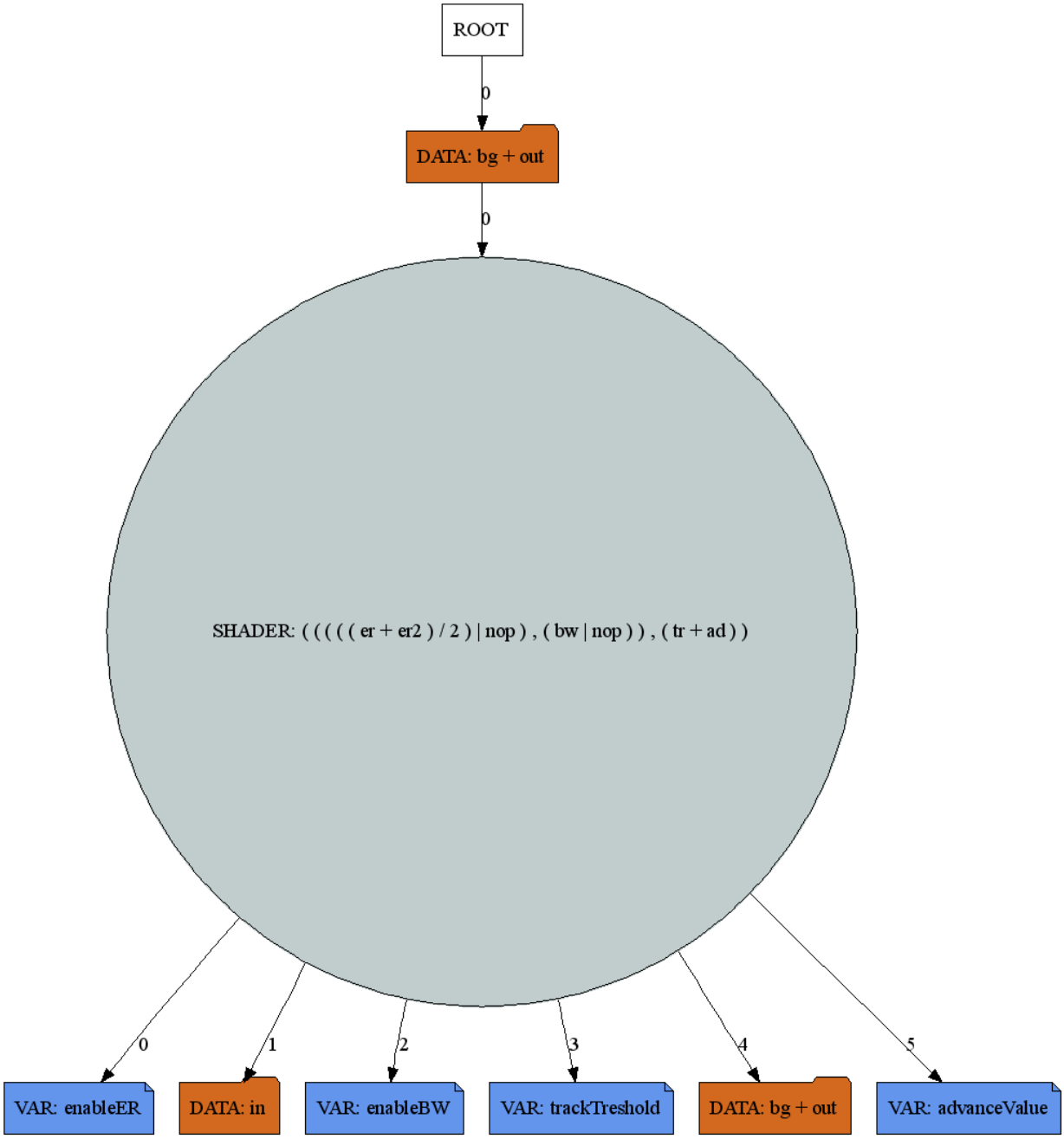

Рис. 16. Схема оптимизированного программного дерева для предложенной задачи

Автоматы, соответствующие неоптимизированной и оптимизированной программам, представлены на рис. 17.

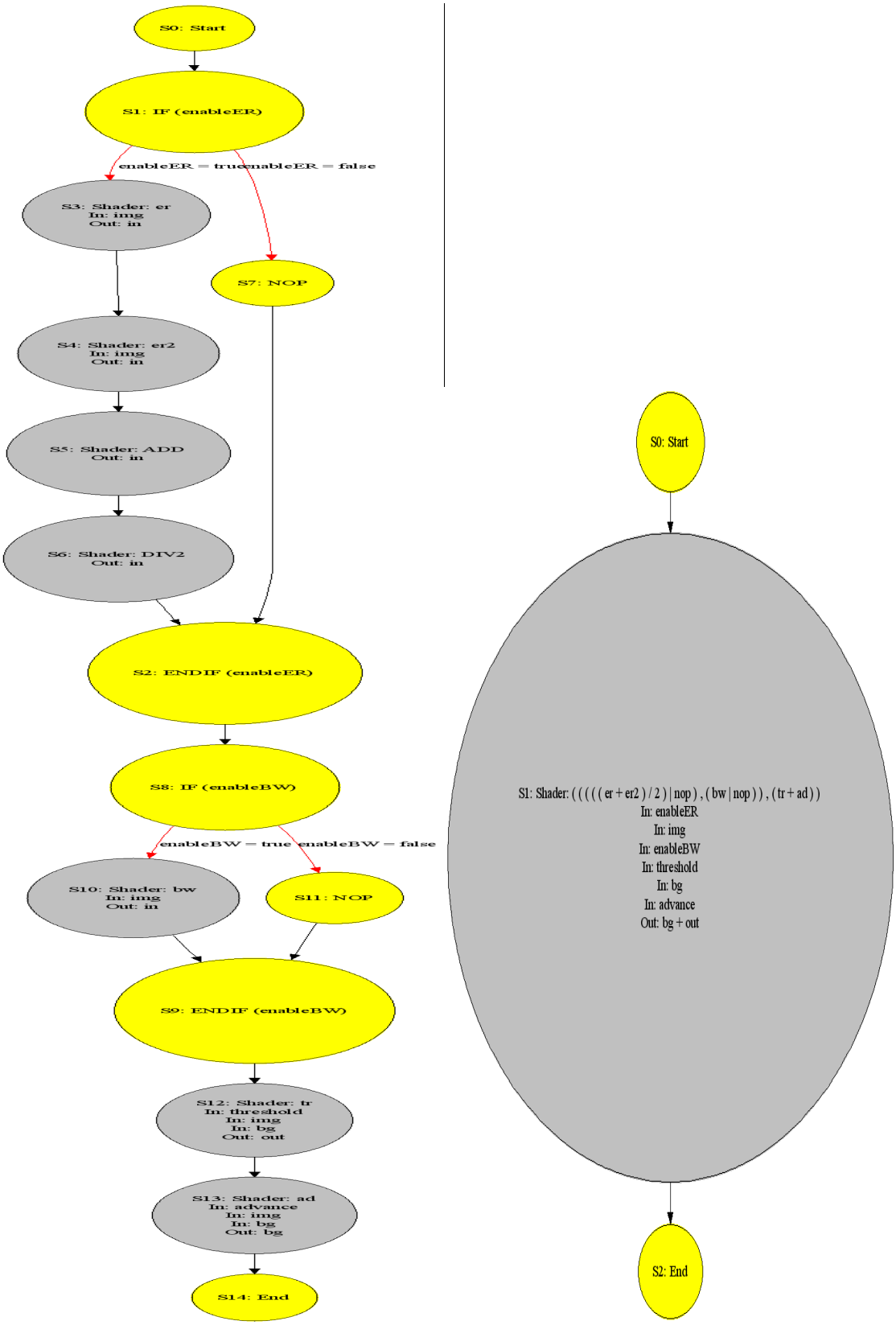

Рис. 17. Схемы неоптимизированного (слева) и оптимизированного (справа) автоматов

4.7. Достоинства и недостатки оптимизатора

Отметим достоинства предложенного программного средства.

- Процесс оптимизации проводится автоматически. Пользователю не требуется знать многие низкоуровневые особенности. Достаточно лишь описать программу в предложенном формате и запустить оптимизатор.
- Предложенный формат описания программы прост для понимания и в то же время обладает мощными возможностями, такими как абстракции.
- Существует возможность гибкой настройки параметров оптимизации для продвинутых пользователей.
- Удобной является возможность обновления оптимизатора. В случае выпуска новой версии пользователю достаточно запустить оптимизатор, предоставив ему уже описанную программу, после чего пользователь получит новый, возможно лучше оптимизированный, результат.

Среди недостатков стоит отметить следующие:

- Оптимизатор не является транслятором в код, необходимый для графического процессора. Поэтому пользователю требуется самостоятельно написать все необходимые шейдеры. Следовательно, требуется знание некоторых особенностей работы с графическим процессором.
- Оптимизатор предоставляет только схему оптимизированного решения и необходимые шейдеры. Пользователю требуется поддерживать связь между *CPU* и *GPU* самостоятельно.

4.8. Перспективы

Важным направлением для дальнейшей работы является создание удобного формата для экспорта полученного автомата и библиотеки для удобного контроля вычислений на графическом процессоре.

Такая библиотека, например, позволит пользователю не заботиться о связи между *CPU* и *GPU*: достаточно будет загрузить экспортированный автомат и начать управление им при помощи простейших функций:

- переход вперед;
- переход назад;
- выполнение до финального состояния.

Подобный подход позволяет также проводить процедуру отладки шейдеров в реальном времени благодаря несложной реализации переходов между состояниями автомата.

Выводы по главе 4

- Выделены составляющие программы для графического процессора.
- Показано удобство абстракций при проектировании данных программ.
- Показан формат записи программы для предложенного автором средства.
- Выявлены достоинства, недостатки и перспективы программного средства.

Выводы по работе

1. Рассмотрены средства реализации вычислений на графическом процессоре и выявлены особенности таких вычислений.
2. Рассмотрен традиционный подход к *GPGPU* и выявлена возможность проведения оптимизаций для сложных программ.
3. Получены и обоснованы методы оптимизации таких программ.

4. Разработано программное средство *GPGPU-Optimizer*, позволяющее оптимизировать *GPGPU*-программы, записанные в специальном виде.
5. Выявлены достоинства и недостатки, а также перспективы данного средства.

Список источников

1. Интернет-ресурс http://www.gpureview.com/show_cards.php
2. Интернет-ресурс http://www.gpgpu.org
3. Интернет-ресурс http://ru.wikipedia.org/wiki/Закон_Амдала
4. Интернет-ресурс http://folding.stanford.edu
5. Интернет-ресурс http://www.gpugrid.net
6. *Мордвинцев А. С.* Использование графического ускорителя для моделирования динамики жидкости методом Lattice-Boltzmann. http://is.ifmo.ru/projects/lattice_boltzmann
7. *Mancheril N.* GPU-based Sorting in PostgreSQL. School of Computer Science Carnegie Mellon University. http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/ngm/15-823/project/Final.pdf
8. Интернет-ресурс http://www.opengl.org
9. Интернет-ресурс http://www.microsoft.com/windows/directx/default.mspx
10. *Kessenich J.* The OpenGL® Shading Language. http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.40.05.pdf
11. *Chang L.* HLSL Introduction. http://www.neatware.com/lbstudio/web/hlsl.html
12. *William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard*. Cg: A System for Programming Graphics Hardware in a C-like Language. http://www-csl.csres.utexas.edu/users/billmark/papers/Cg
13. Интернет-ресурс http://msdn.microsoft.com/en-us/library/bb219840(VS.85).aspx
14. Интернет-ресурс http://www.nvidia.com/cuda
15. Интернет-ресурс http://ati.amd.com/technology/streamcomputing
16. *Boyd C.* The DirectX 11 Compute Shader. http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf

17. Интернет-ресурс http://libsh.org
18. *Göddeke D.* GPGPU::Basic Math Tutorial. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html
19. *Kirillov A.* Motion Detection Algorithms. http://www.codeproject.com/KB/audio-video/Motion_Detection.aspx
20. Интернет-ресурс http://www.graphviz.org
21. Интернет-ресурс http://en.wikipedia.org/wiki/High_Level_Shader_Language