МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Факультет **Информационных технологий и программирования**
Направление **Прикладная математика и информатика**
Специализация **Технологии разработки ПО**
Академическая степень **Магистр прикладной математики и информатики**

Кафедра **Компьютерных технологий** Группа **6539**

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Оптимизация высокопроизводительных вычислений с использованием графических процессоров

Автор магистерской диссертации ___**Селифонов Е. В.**_____(подпись)
(Фамилия, И., О.)

Научный руководитель_____**Шалыто А. А.**_____(подпись)
(Фамилия, И., О.)

Руководитель магистерской программы_____(подпись)
(Фамилия, И., О.)

К з а щ и т е д о п у с т и т ь

Зав. кафедрой _____ВАСИЛЬЕВ В.Н. _____(подпись)
(Фамилия, И., О.)

“____”_____20 ____г.

Санкт-Петербург, 2011 г.

Санкт-Петербургский государственный университет информационных технологий, механики и оптики

АННОТАЦИЯ
ПО МАГИСТЕРСКОЙ ДИССЕРТАЦИИ

Студента Селифонова Е.В.
(Фамилия, И., О.)

Факультет ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ПРОГРАММИРОВАНИЯ

Кафедра КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ **Группа** 6539

Направление (специальность) ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА

Квалификация (степень) МАГИСТР ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Наименование темы: Оптимизация высокопроизводительных вычислений с использованием графических процессоров

Руководитель Шалыто А.А. – д.т.н., проф. СПбГУИТМО
(Фамилия, И., О., ученое звание, степень)

Консультант –
(Фамилия, И., О., ученое звание, степень)

КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ И ОСНОВНЫЕ ВЫВОДЫ

объем 70 стр., **графический материал** – стр., **библиография** 19 наим.

- Направление и задача исследований
  Выявление способов оптимизации программ для графического процессора.
  Создание библиотеки для упрощения взаимодействия с графическим процессором.
  Использование связки из центрального и графического процессоров для проведения параллельных вычислений.
- Проектная или исследовательская часть (с указанием основных методов исследований, расчетов и результатов)
  Выявление алгоритма оптимизации программы, состоящей из нескольких шейдеров, на основе ее текстового описания.
  Определение методов оптимизации кода объединенных шейдеров.
  Выявление структур данных, описывающих взаимодействие с графическим процессором и создание библиотеки для упрощения взаимодействия.
  Определение возможности одновременного использования центрального и графического процессоров для выполнения программы для графического процессора.
- Является ли работа продолжением курсовых проектов (работ), есть ли публикации
  Работа является продолжением бакалаврской работы «Оптимизация работы программ для графического процессора».
  Есть публикация:
  Труды VIII Всероссийской межвузовской конференции молодых ученых. СПбГУ ИТМО. 2011.

Практическая ценность работы. Рекомендации по внедрению.
Использование предложенного программного средства для оптимизации вычислений, проводимых с использованием графического процессора, например, для задачи определения движения на основе видеоинформации, получаемой с веб-камеры.

Выпускник _____
(подпись)

Руководитель _____
(подпись)

“_____” _____ 20 г.

Оглавление

Введение

Ниже описывается роль графических процессоров (*Graphics Processing Unit – GPU*) в современных высокопроизводительных вычислительных системах, а также рассказывается о новых видах вычислительных устройств, совмещающих в себе мощь центрального (*Central Processing Unit – CPU*) и графического процессоров.

Работу многих современных систем невозможно представить без использования высокопроизводительных вычислений. Еще недавно они использовались лишь для узкого круга научных задач, но уже сейчас с выполнением расчетов над большими массивами данных приходится сталкиваться повсеместно. Если раньше для выполнения таких вычислений требовались суперкомпьютеры, то сейчас существует большое количество способов их проведения на современных персональных компьютерах.

Одним из способов является объединение нескольких компьютеров в сеть. Данные разбиваются центральным компьютером (*координатором*) на несколько частей, затем посылаются компьютерам-вычислителям. Далее координатор получает результаты от каждого вычислителя и при необходимости может провести над ними операцию *свертки*, преобразующую обработанный массив в результирующее скалярное значение. Наиболее известной реализацией данного подхода является схема *MapReduce* от компании *Google* [1], на основе которой построено множество алгоритмов. Рассмотрение подхода объединения нескольких компьютеров выходит за рамки тематики данной работы. Мы будем рассматривать вычисления на одном компьютере, но с использованием нескольких вычислителей.

Наибольшая часть расчетов в компьютерах проводится на центральном процессоре. Для повышения его производительности изначально требовалось лишь повышать тактовую частоту. Однако ее повышение подошло к технологическому пределу, поэтому производители процессоров стали

увеличивать количество вычислителей и оптимизировать инструкции для одновременной работы с векторами данных. Современные процессоры архитектуры *x86* имеют небольшое количество вычислительных ядер с высокой тактовой частотой. Наиболее популярны модели с четырьмя ядрами с частотой около 3 ГГц. Широко распространены расширенные наборы *SIMD*-инструкций (*Single Instruction – Multiple Data, одна инструкция – массив данных*): *MMX*, *SSE*, *SSE2*, *SSE3*, *SSE4*, которые будут использованы в данной работе.

В первых компьютерах центральный процессор производил все необходимые расчеты. Затем появились дополнительные функциональные модули: сопроцессоры и отдельные процессоры. Они предназначались для выполнения определенных задач с более высокой, чем центральный процессор, скоростью и были выполнены в виде отдельных микросхем. В современных компьютерах помимо центрального процессора существует множество вычислительных устройств, среди которых хочется отметить графические процессоры. Они изначально были предназначены для обработки трехмерной графики с целью разгрузки центрального процессора, для которого данная задача является достаточно трудоемкой. Отличительной особенностью графических процессоров является очень большое количество вычислителей (шейдерных процессоров), работающих на меньшей частоте, чем центральный процессор. Для современных видеокарт число шейдерных процессоров превышает одну тысячу и достигает значения в 1536 для видеокарты *AMD Radeon HD 6970* [2]. Частота таких процессоров может достигать 1544 МГц для видеокарты *nVidia GeForce GTX 580*.

В связи с высоким вычислительным потенциалом графических процессоров образовалось направление *GPGPU* (*General Purpose Graphics Processors Usage – использование графических процессоров для общих целей*) [3], целью которого является использование их не только для обработки трехмерной графики, но и для решения общих задач. В данной работе будет

использоваться традиционный подход к *GPGPU* с улучшенными авторскими оптимизациями, описанными в работе [4].

В настоящее время крупные производители центральных процессоров решили вновь решили объединить функциональные модули на одной плате, что привело к появлению нового вида процессоров: *APU* (*Accelerated Processing Unit – процессоры ускоренной обработки*). Данные устройства сочетают на одном кристалле микросхемы:

- центральный процессор;
- другое вычислительное устройство, например:
    - графический процессор;
    - программируемую логическую интегральную схему (*ПЛИС*).

Отметим крупные проекты современных производителей:

- *AMD Fusion* [5];
- *Intel Sandy Bridge* [6];
- *NVIDIA Project Denver* [7].

В данной работе будут одновременно использоваться мощности центрального и графического процессоров, что позволит применять ее результаты на современных *APU*.

Глава 1. Традиционный подход к использованию *GPU* для высокопроизводительных вычислений

В этой главе приводится описание традиционного способа выполнения *GPGPU*-расчетов, определяются его особенности и ограничения. Также производится краткий обзор существующих библиотек, позволяющих использовать графические процессоры для высокопроизводительных вычислений, с указанием их особенностей.

1.1. Стандартные графические библиотеки

Традиционный подход к программированию графических процессоров для общих целей предусматривает эмуляцию процесса *рендеринга* (*rendering*) – отрисовки трехмерного изображения. Существуют две стандартные библиотеки для работы с трехмерной графикой, поддерживаемые практически всеми современными видеокартами:

- *OpenGL* [8];
- *DirectX* [9].

Данные библиотеки предоставляют стандартизированный набор функций, необходимых для рендеринга трехмерных сцен, которые реализуются в драйвере видеокарты.

Существующие специализированные *GPGPU*-библиотеки, о которых будет сказано ниже, в своей основе используют одну из двух данных библиотек. Стоит отметить, что среда *OpenGL* является кроссплатформенной, тогда как *DirectX* работает только в операционной системе *Microsoft Windows*. В данной работе будет использована библиотека *OpenGL*.

1.2. Представление данных на *GPU*

В случае проведения высокопроизводительных вычислений центральный процессор оперирует массивами данных. В качестве данных мы будем рассматривать вещественные числа. С логической точки зрения мы можем оперировать массивами разных размерностей. Наиболее распространенными являются одномерные и двумерные массивы. С точки зрения памяти мы чаще всего работаем с ее непрерывной линейной областью.

В случае использования графического процессора операции проводятся над *текстурами* – изображениями, чаще всего двумерными. Каждая ячейка текстуры обычно представляет собой вектор из четырех вещественных чисел – компонентов цвета:

- r – красный компонент;
- g – зеленый компонент;
- b – синий компонент;
- a – альфа-канал, отвечает за прозрачность при смешивании цветов.

Каждый компонент обычно хранит нормализованное значение цвета в промежутке $[0; 1]$, но для проведения вычислений можно записывать любые вещественные числа.

Таким образом, текстура размерности $M \times N$ хранит в себе $4 * M * N$ вещественных чисел, а для отображения одномерного массива из N чисел необходима текстура из $\frac{N}{4}$ элементов, с размерами $\left[\frac{\sqrt{n}}{2}\right] \times \left[\frac{\sqrt{n}}{2}\right]$.

Отметим важные характеристики текстур:

- размеры:
  - ширина;
  - высота;
- *текстурная цель* (*texture target*) – задает набор внутренних операций *OpenGL* по работе с текстурой; нам важен вид представления текстурных координат:

    - *GL_TEXTURE_2D* – такая текстура всегда имеет нормализованные координаты, лежащие в отрезке $[0; 1]$;
    - *GL_TEXTURE_RECTANGLE_ARB* – такая текстура имеет координаты от нуля до ее размерности по оси; в работе мы будем пользоваться этим видом из-за удобства доступа к соседним ячейкам текстуры;
- формат – задает, какие данные можно извлечь из каждой ячейки текстуры; отметим наиболее важные форматы:
    - *GL_RGBA* – хранит все четыре описанных выше компонента цвета;
    - *GL_RGB* – не хранит альфа-канал;
    - *GL_LUMINANCE* – хранит только один компонент цвета;
- внутренний формат – задает внутреннее представление текстуры в видеопамяти; покажем соответствие между внутренним и внешним форматами:
    - *GL_RGBA32F_ARB* ↔ *GL_RGBA*;
    - *GL_RGB32F_ARB* ↔ *GL_RGB*;
    - *GL_LUMINANCE32F_ARB* ↔ *GL_LUMINANCE*;
- метод *фильтрации* (*filtering*) – процесса обработки нескольких пикселей исходной текстуры для получения окончательного значения пикселя результирующего изображения; для проведения точных вычислений фильтрацию требуется отключить (*GL_NEAREST*);
- метод *обертки* (*wrapping*) – обработки текстурных координат вне текстурных размерностей; рекомендуется фиксировать координаты в их допустимом диапазоне для проведения точных вычислений на границах массивов (*GL_CLAMP*).

Существует возможность записи данных из оперативной памяти компьютера в текстуру, которая хранится в видеопамяти, а также чтения

данных из текстуры в оперативную память. При этом имеется возможность задать формат передачи данных:

- считываемые или записываемые ячейки:
  - *GL_RGBA*;
  - *GL_RGB*;
  - *GL_RED*;
  - *GL_GREEN*;
  - *GL_BLUE*;
  - *GL_ALPHA*;
- формат данных в оперативной памяти:
  - *GL_FLOAT* – компоненты цвета хранятся как вещественные числа;
  - *GL_UNSIGNED_BYTE* – компоненты цвета хранятся как целые числа в диапазоне $[0; 255]$.

1.3. Представление программ на *GPU*

Для центрального процессора мы записываем код программы на специальном языке программирования, который затем компилируется в набор процессорных инструкций или интерпретируется в другой программе. Существует единый для архитектуры набор инструкций, которые могут быть выполнены на любом процессоре данной архитектуры.

Программы для графического процессора называются *шейдерами*. Они чаще всего записываются в текстовом представлении и компилируются драйвером видеокарты в процессе выполнения приложения. Существует несколько языков программирования, среди которых стоит отметить следующие:

- *GLSL* – для среды *OpenGL* [11];
- *HLSL* – для среды *DirectX* [12];
- *Cg* – для сред *OpenGL* и *DirectX* [13];

- *DirectX ASM* – для среды *DirectX* [14].

Первые три языка похожи на язык *C*, используемый для написания программ для центрального процессора, с небольшими изменениями и дополнениями. Язык *DirectX ASM* является ассемблероподобным и позволяет описывать низкоуровневые программные операции. В дальнейшем в работе рассматривается язык *GLSL*.

Язык *GLSL* поддерживает следующие встроенные типы данных:

- скалярные типы данных:
  - *float*;
  - *int*;
  - *bool*;
- вектора до четырех элементов:
  - *vec2*, *vec3*, *vec4*;
  - *ivec2*, *ivec3*, *ivec4*;
  - *bvec2*, *bvec3*, *bvec4*;
- матрицы:
  - *mat2*, *mat3*, *mat4*;
- специализированные текстурные типы:
  - *sampler1D*, *sampler2D*, *sampler3D*.

Стоит отметить, что существует понятие *модели шейдеров* (*Shader Model*), которая устанавливает некоторые ограничения на возможность использования тех или иных инструкций. Например, в *Shader Model 1* и *2* невозможно использовать условные операторы. Также в старых версиях существовало строгое ограничение на максимальное количество инструкций в одном шейдере [14]. Текущей версией является *Shader Model 5* [15].

Шейдеры делятся на два основных типа:

- *вершинные шейдеры (vertex shaders)* оперируют с геометрическими данными вершин объектов в трехмерном пространстве;

- *пиксельные* или *фрагментные шейдеры* (*pixel / fragment shaders*) применяют некоторое преобразование к фрагментам изображения, *пикселям*.

Шейдеры применяются в порядке, показанном на упрощенной схеме (рис. 1).

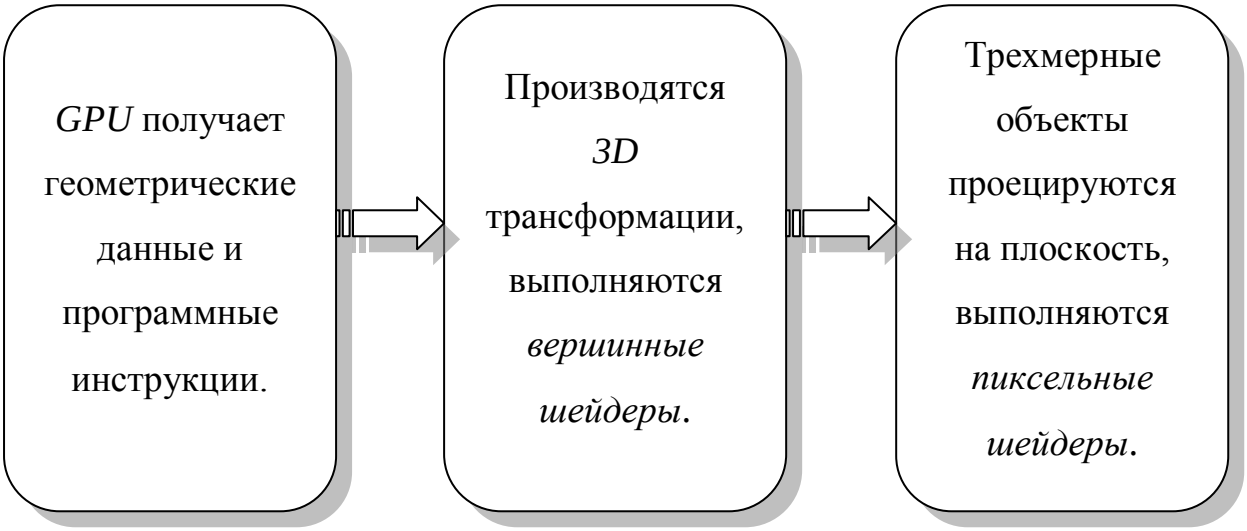

Рис. 1. Упрощенная схема применения шейдеров

В *GPGPU* чаще всего используются пиксельные шейдеры, и именно они будут рассмотрены в данной работе. Фрагментный шейдер применяется к каждому участку изображения, поэтому он параллельно выполняется столько раз, сколько пикселей имеется в целевом изображении (экранном буфере или текстуре) – для изображения размерности $M \times N$ он будет выполнен $M * N$ раз.

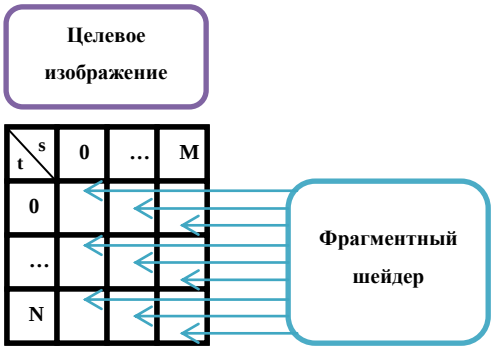

Рис. 2. Применение фрагментного шейдера к каждой ячейке целевого изображения

Приведем пример простейшего фрагментного шейдера:

```
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect input_tex;
uniform float coefficient;
void main(void)
{
    vec4 inp = texture2DRect(input_tex, gl_TexCoord[0].st);
    vec4 result = inp * coefficient;
    gl_FragColor = result;
}
```

Типичный фрагментный шейдер состоит из следующих частей:

- объявление использование расширения *OpenGL* для поддержки текстур *GL_TEXTURE_RECTANGLE_ARB* (строка 1);
- объявление входных данных с ключевым словом *uniform*, передающихся кодом, вызывающим шейдер со стороны центрального процессора (строки 2-3); дополнительно шейдеру передаются:
    - текстурные координаты *gl_TexCoord* целевого изображения;
    - указатель *gl_FragColor* на ячейку для записи результата;
- функция *main*, вызывающаяся при обработке каждой ячейки целевого изображения шейдером (строки 4-9); обычно в основной функции происходят следующие операции:
    - чтение из одной или нескольких входных текстур по определенным координатам при помощи функции *texture2DRect* (строка 6);
    - обработка входных данных и прочитанных значений и вывод результирующего вектора (строка 7);
    - запись результирующего вектора в *gl_FragColor* (строка 8).

1.4. Запуск вычислений на *GPU* и запись результатов в текстуру

Традиционно видеокарты используются для отрисовки изображения на экран компьютера. Целью *GPGPU* является проведение некоторых расчетов с использованием возможностей графического процессора, поэтому необходимо изменить стандартную схему работы и отказаться от вывода на экран. Входные данные, как уже было показано, хранятся в текстурах, поэтому результаты также необходимо записывать в текстуры, чтобы их можно было использовать в качестве входных данных для новых расчетов.

Рассмотрим упрощенную схему рендеринга одного кадра изображения (рис. 3).

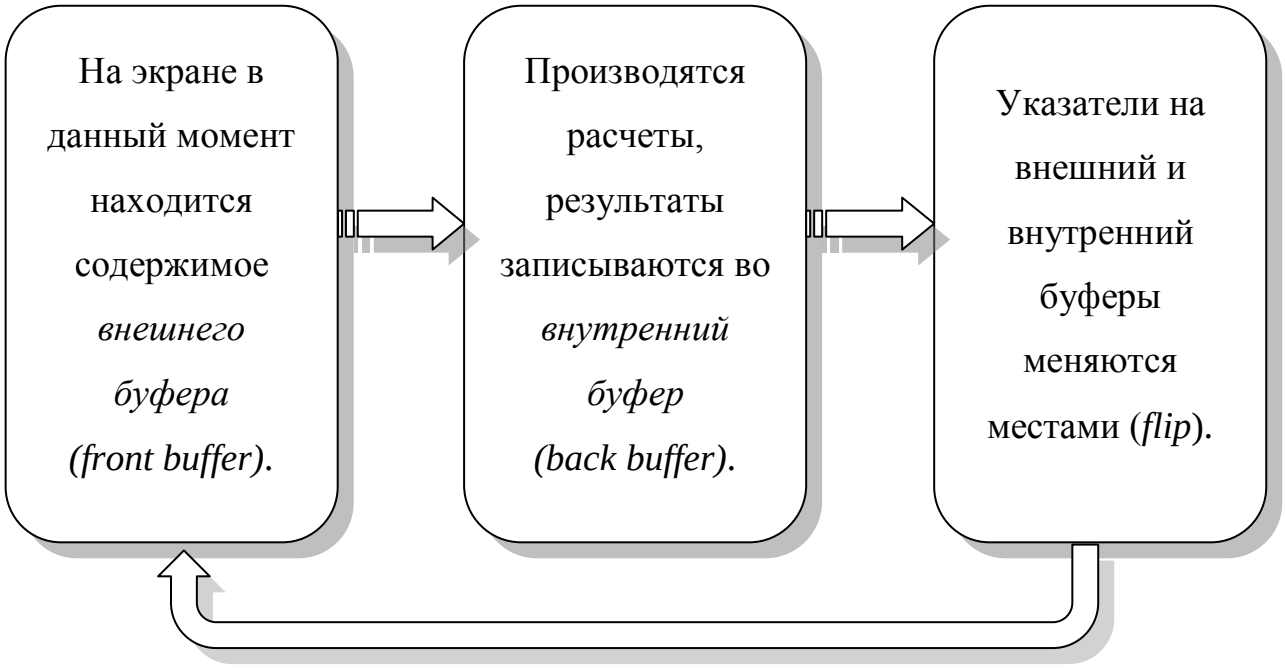

Рис. 3. Упрощенная схема рендеринга одного кадра изображения

Требуется изменить *цель рендеринга* (*render target*) – область видеопамяти, в которую заносятся результаты рендеринга. Стандартной целью рендеринга является *внутренний буфер* (*back buffer*) – специальная область памяти, которая будет выведена на экран по окончании операции рендеринга. Наличие двух постоянно меняющихся буферов связано с тем,

что в процессе расчетов необходимо выводить старую картинку на экран, иначе будет заметно мерцание.

Среда *OpenGL* предоставляет специальный объект, называемый *фреймбуфер* (*framebuffer*), который можно сделать целью рендеринга. Фреймбуфер позволяет *привязать* (*attach*) к нему одну или несколько текстур, тогда результаты рендеринга попадут в текущую привязанную текстуру, что нам и требуется.

Для того чтобы провести запись в целевую текстуру, требуется нарисовать прямоугольник. Необходимо, чтобы каждый пиксель изображения соответствовал *текселю* (*texel* – ячейка текстуры) текстуры (рис. 4), для этого поставим следующие условия:

- размер прямоугольника должен быть равен размеру целевой текстуры;
- размер области рендеринга должен быть равен размеру целевой текстуры;
- проекция должна быть ортографической, параметрами матрицы должны являться размеры целевой текстуры.

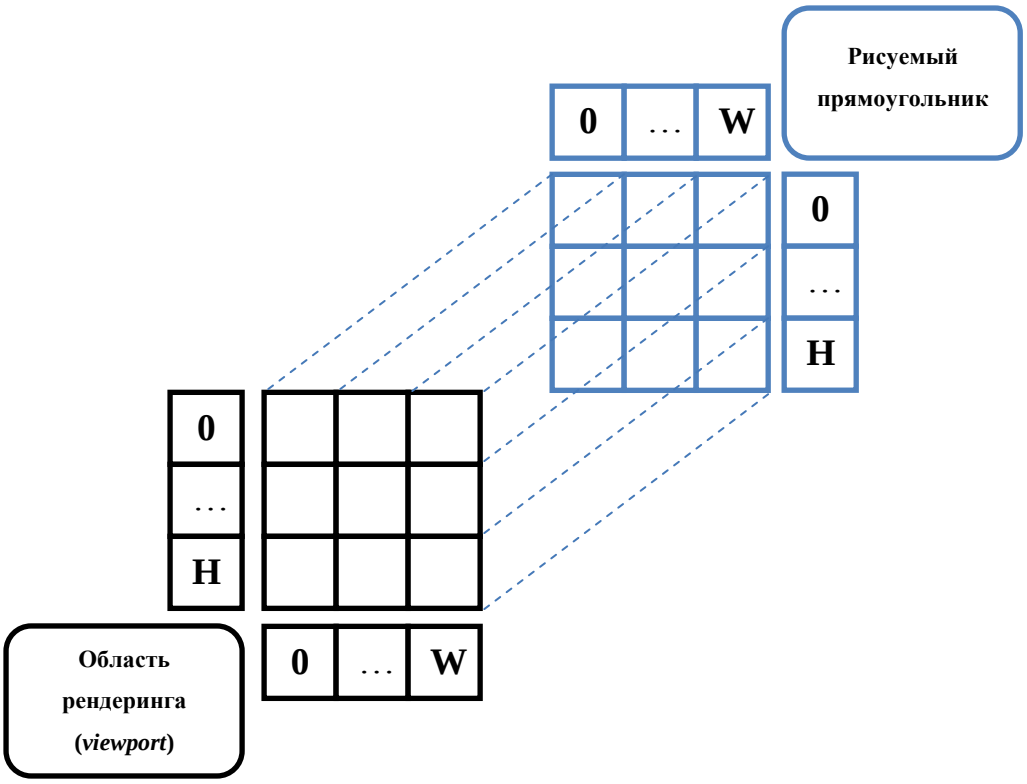

Рис. 4. Проекция прямоугольника с наложенной текстурой на область рендеринга

Для запуска вычислений необходимо:

- указать целевой фреймбуфер и, если это еще не было сделано или требуется указать новую целевую текстуру, привязать к нему текстуру;
- указать используемый шейдер;
- передать шейдеру данные, помеченные ключевым словом *uniform*;
- при обновлении входного массива данных в оперативной памяти произвести запись в соответствующую ему текстуру;
- нарисовать прямоугольник необходимых размеров;
- при необходимости использования результатов на центральном процессоре считать данные из текстуры или из фреймбуфера в оперативную память.

Важным ограничением является невозможность одновременных операций чтения и записи для одной текстуры, поэтому часто создается две текстуры, одна из которых служит для чтения, а другая – для записи; после вычислений происходит обмен функций этих текстур.

1.5. Достоинства и недостатки традиционного подхода к *GPGPU*

Отметим следующие достоинства традиционного подхода:

- данный подход использует стандартизированные графические библиотеки и языки программирования;
- работа происходит на низком уровне, что предоставляет возможность обнаруживать низкоуровневые ошибки и обрабатывать их;
- имеется возможность создавать оптимизированные программы.

Среди недостатков необходимо отметить следующие:

- иногда требуется использовать расширения стандартов, которые могут не поддерживаться на старых видеокартах или на графических процессорах определенного производителя;

- один шейдер может выдавать только один результат; этот факт приводит к необходимости писать сложные программы, состоящие из нескольких шейдеров;
- программы, состоящие из нескольких шейдеров, зачастую являются неоптимизированными, их оптимизация требует дополнительных знаний и усилий;
- процесс добавления нового шейдера в последовательность существующих на уровне *OpenGL* требует написания большого количества кода и делает код слишком «заточенным» под конкретную задачу;
- в процессе вычислений на графическом процессоре центральный процессор простаивает в ожидании результатов.

Работа [4] была посвящена исправлению второго и третьего недостатков. В данной работе будут предложены новые оптимизации и методы для избавления от третьего, четвертого и пятого недостатков.

1.6. Обзор существующих *GPGPU*-библиотек

В настоящее время существует две приобретающие популярность библиотеки: *CUDA* (*Compute Unified Device Architecture*) [16] и *OpenCL* (*Open Computing Language*) [17]. Данные библиотеки упрощают взаимодействие с *GPU*, избавляют от необходимости написания низкоуровневого шейдерного кода и предоставляют свой язык, похожий на *C*.

Основной проблемой библиотеки *CUDA* является ограниченный набор поддерживаемых графических процессоров: данная библиотека не работает на видеокартах *AMD*.

Библиотека *OpenCL* лишена этого недостатка и может в ближайшее время стать стандартом. Стоит отметить, что язык написания программ для этой библиотеки абстрагирован от типа устройства, на котором этот код будет выполняться. Один и тот же код можно выполнить как на графическом,

так и на центральном процессорах. К сожалению, на текущий момент не все устройства поддерживают данную библиотеку, и некоторые драйверы имеют проблемы со стабильностью.

Выводы по главе 1

- Рассмотрен традиционный подход к проведению вычислений на графических процессорах и выявлены его достоинства и недостатки.
- Определены проблемы традиционного подхода, решаемые в данной работе.
- Рассмотрены существующие *GPGPU*-библиотеки и обозначены их основные недостатки.

Глава 2. Оптимизация *GPGPU*-программ из нескольких шейдеров

В этой главе будут рассмотрены оптимизации для программы, состоящей из нескольких шейдеров, производимые при помощи приложения *GPGPU-Optimizer*. Будут выполнены как оптимизации, описанные в работе [4], так и новые, встроенные в обновленную версию приложения.

2.1. Составные части *GPGPU*-программы и их текстовая запись

Перечислим необходимый набор данных, необходимых для записи программы из нескольких шейдеров:

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

Подробную информацию о записи программы в формате, принимаемом оптимизатором, можно прочитать в работе [4]. В качестве примера рассмотрим тестовую задачу из этой работы:

```
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
```

2.2 Построение программного дерева

Для проведения оптимизации программы необходимо построить дерево, представляющее текстовое описание в удобном для оптимизатора формате. В программном дереве может быть несколько типов узлов:

- корневой узел – в качестве потомков содержит все данные, которые необходимо вычислить в ходе выполнения программы;
- узел с данными – представляет массив данных, который требуется передать на вход предку или в который необходимо записать результат потомка;

- узел с переменной – представляет переменную, которую надо передать на вход шейдеру-предку или которая описывает условие условного оператора-предка;
- узел с отсутствием операций – передает данные потомка предку без их изменения;
- узел с шейдером – выполняет шейдер с входными данными, полученными от потомков, и передает результат предку;
- узел с арифметической операцией – выполняет арифметическое преобразование входных данных, полученных от потомков, и передает результат предку;
- узел с началом ветвления – описывает условный оператор; первым потомком является переменная, описывающая, какую ветвь следует выбирать, далее следуют различные ветви выполнения;
- узел с завершением ветвления – описывает завершение условного оператора, все его ветви сводятся к этому узлу.

Деревья поочередно строятся для каждой инструкции. Парсер разбирает строковое представление, определяет выходной массив с данными, при необходимости раскрывает абстракции и определяет набор функций, которые необходимо выполнить для получения результата.

В каждом последующем дереве узел с данными, в которые была произведена запись в одной их предыдущих инструкций, заменяется деревом последней такой инструкции. Пример для первых двух инструкций предложенной программы можно увидеть на рис. 5. Красным цветом помечен заменяемый узел с данными.

Программное дерево получается после обработки последней инструкции. Стоит отметить, что программное дерево может не являться деревом по определению – некоторые узлы могут иметь более одного предка, но чаще всего оно представляет древовидную или близкую к ней структуру.

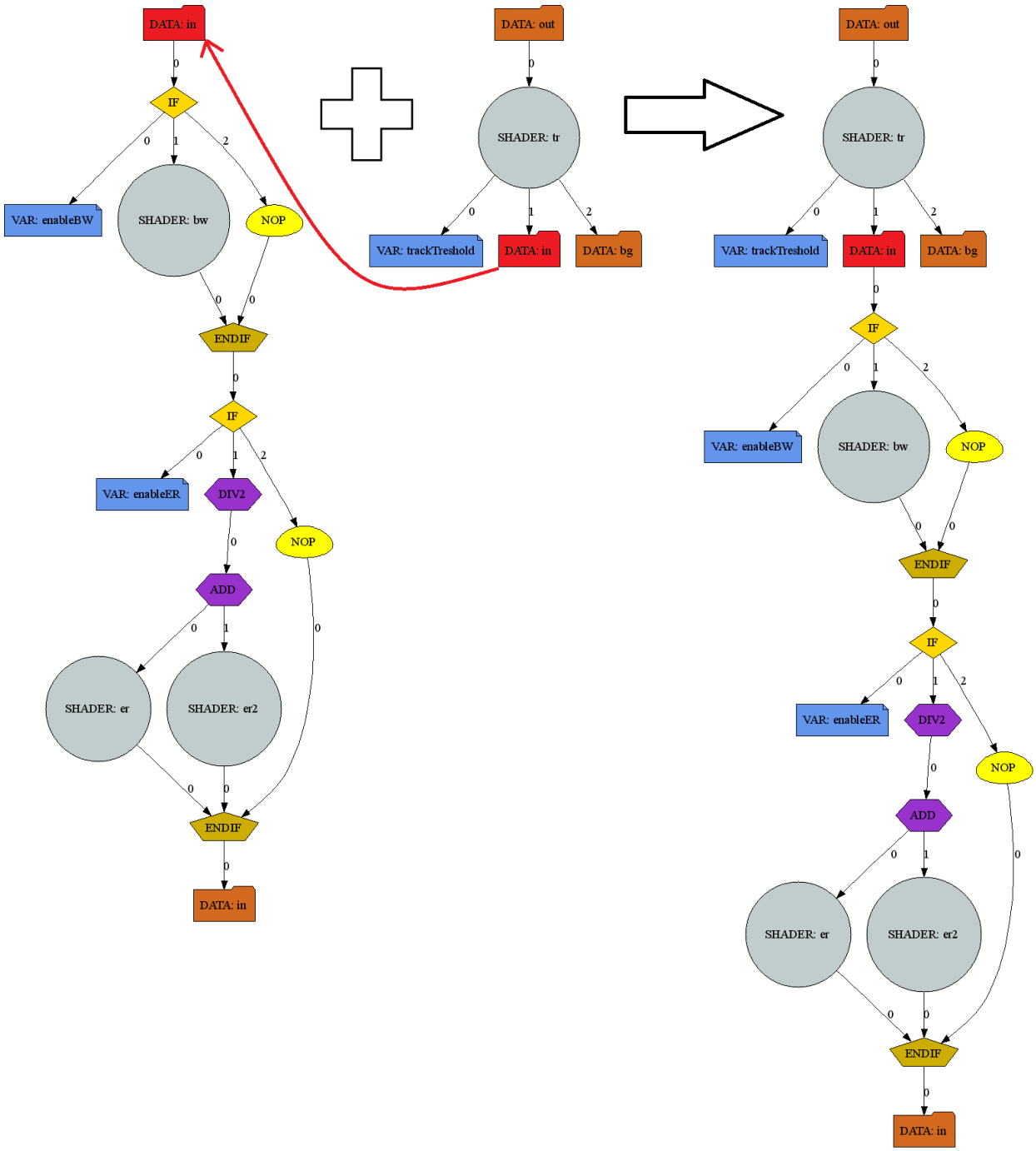

Рис. 5. Схема объединения двух деревьев, представляющих инструкции

2.3. Оптимизация программного дерева

После построения программного дерева можно начинать процесс его оптимизации. Мы будем применять техники, предложенные в работе [4].

Основой алгоритма оптимизации является многократный обход дерева. Нам понадобятся следующие вспомогательные переменные:

- *_stack* – стек посещаемых узлов, необходимый для удобного доступа к узлу-предку;
- *_changed* – булевская переменная, указывающая, произошли ли изменения в структуре дерева на данной итерации;
- *_dataSubstitute* – словарь, указывающий какие узлы с данными необходимо заменить другими узлами с данными.

Приведем последовательность действий алгоритма оптимизации.

1. Проверим узлы с данными, являющиеся потомками корневого узла, на возможность объединения нескольких соответствующих им массивов данных в один за счет применения разных текстурных каналов.
    a. Если найдено одно или несколько множеств таких узлов, создадим новые узлы с данными, являющимися объединением элементов этих множеств, и добавим в словарь *_dataSubstitute* замену соответствующих узлов на созданные узлы (рис. 6).
2. Установим переменную *_changed* в *false*.
3. Запустим *процедуру обхода*, описанную ниже, для корневого узла.
4. Если переменная *_changed* имеет значение *true*, перейдем к пункту 2, иначе завершим выполнение алгоритма.

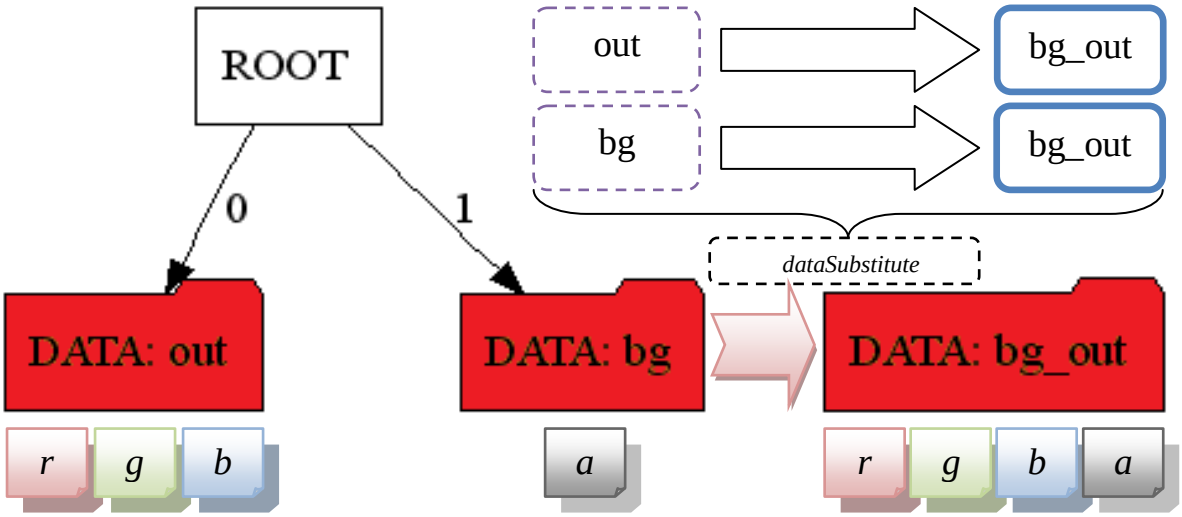

Рис. 6. Схема работы шага 1.a. алгоритма оптимизации

Процедура обхода получает на вход узел дерева и выполняет следующую последовательность действий.

1. Добавим текущий узел в *_stack*.
2. Для каждого потомка текущего узла в порядке от первого к последнему вызовем *процедуру обхода*.
3. Вытащим из *_stack* текущий узел.
4. Выполним специфичные для типа текущего узла действия.

Рассмотрим действия, выполняющиеся на шаге 4 процедуры обхода, для каждого типа узла.

2.3.1. Корневой узел

Для корневого узла проверим узлы с данными, являющиеся его потомками на наличие дубликатов, которые могли появиться вследствие оптимизации использования текстурных каналов. В случае обнаружения таких дубликатов выполним следующие действия.

- Склеим узлы-дубликаты в один узел.
- Установим потомком объединенного узла арифметический узел с операцией сложения.
- Потомками нового арифметического узла установим потомков исходных узлов-дубликатов.

Схема данной операции представлена на рис. 7.

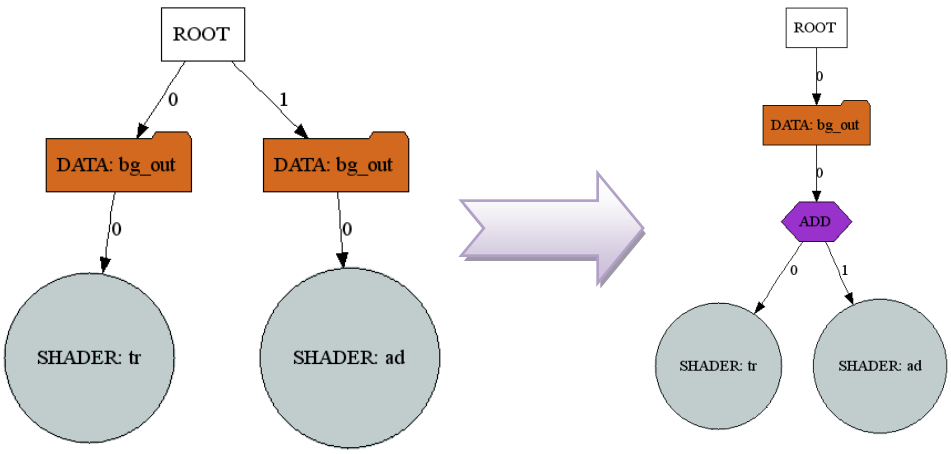

Рис. 7. Схема удаления узлов-дубликатов

2.3.2. Узел с данными

Для узла с данными проверим, не находится ли он в словаре замены данных *_dataSubstitute*. Если его требуется заменить, найдем этот узел в списке потомков родительского узла, который может быть получен из вершины стека *_stack*, и поменяем на узел-замену.

Далее проверим, является ли этот узел промежуточным – у него один предок, не являющийся корнем и один потомок. Если это так, удалим этот узел из дерева – запись данных в него на данном этапе не является финальной, а оптимизатор может выбрать более удачный целевой массив для записи. Схема данной операции представлена на рис. 8.

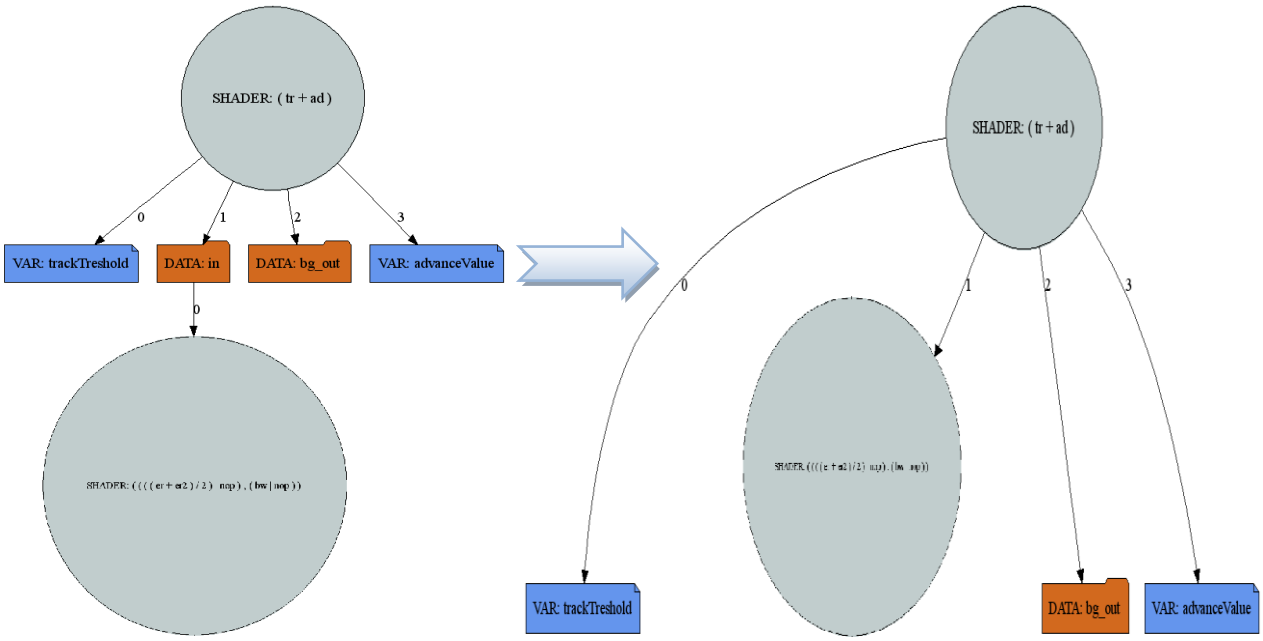

Рис. 8. Схема удаления промежуточных узлов-данных

2.3.3. Узел с отсутствием операции

Узел с отсутствием операции появляется в дереве, например, в случае использования условных операторов с пустой ветвью выполнения. Он может быть рассмотрен как промежуточный узел и удален из дерева по схеме, предложенной для промежуточных узлов-данных в разд. 2.3.2. и представленной на рис. 8.

2.3.4. Узел с шейдером

Для узла с шейдером возможно проведение двух видов оптимизаций:

- объединение с шейдером-потомком;
- внесение под каждую ветвь условного узла-потомка.

Объединение с шейдером-потомком возможно лишь в случае, если шейдер, соответствующий текущему узлу, является шейдером с единичной зависимостью – читает данные из входной текстуры, предоставляемой шейдером-потомком, только по своим текущим координатам.

- Проверим, что текущий шейдер является шейдером с единичной зависимостью. Если не является, прекращаем действия.
- Проверим число шейдеров-потомков, просмотрев типы узлов-потомков. Если их число не равно одному, прекращаем действия.
- Создадим новый узел с объединенным шейдером и заменим на него текущий узел в списке потомков узла-предка. Алгоритм генерации кода объединенного шейдера будет рассмотрен ниже.
- Добавим в список потомков нового узла всех потомков шейдера-потомка.
- Добавим в список потомков нового узла всех потомков текущего узла-шейдера, кроме шейдера-потомка.
- Удалим из списка потомков нового узла все дубликаты, если они имеются.

Схема данной операции представлена на рис. 9.

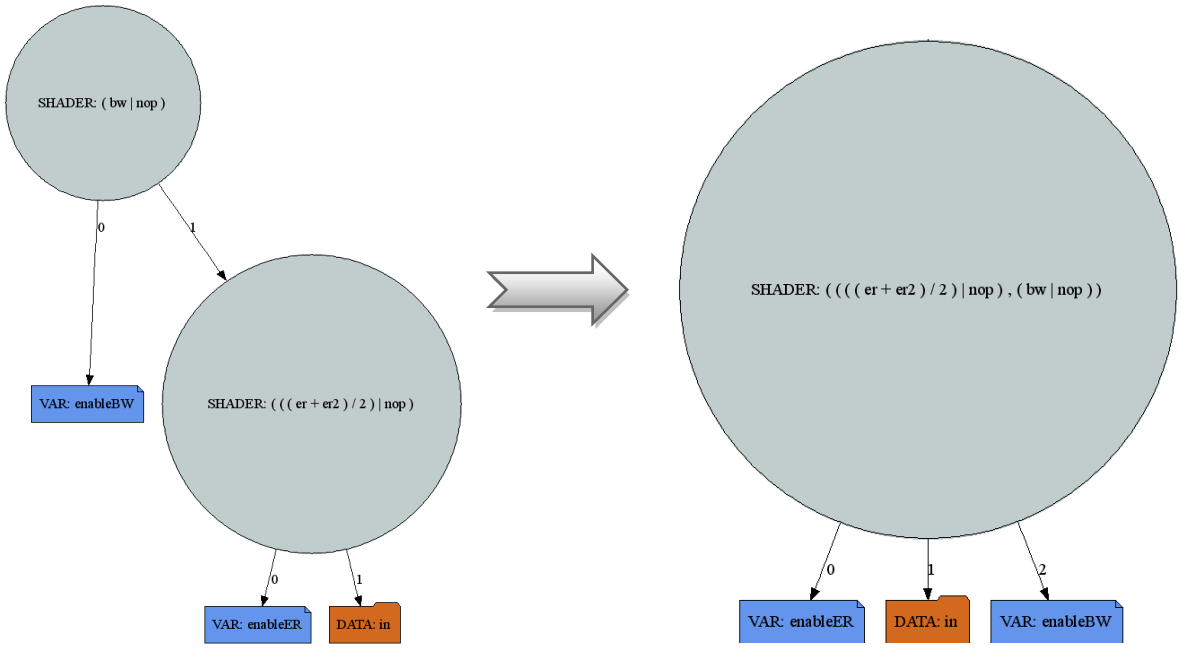

Рис. 9. Схема объединения узлов-шейдеров

Внесение шейдера под ветви условного оператора возможно при наличии ровно одного условного оператора-потомка.

- Проверим число условных операторов-потомков, просмотрев типы узлов-потомков. Если их число не равно одному, прекращаем действия.
- Для каждой *i*-ой ветви условного оператора назовем узлом текущей ветви узел, который является *i+1*-ым потомком узла-условного оператора и представляет текущую ветвь выполнения, и выполним следующие действия.
    - Создадим копию текущего узла-шейдера и добавим в нее всех потомков оригинала.
    - Заменим в списке потомков узла-условного оператора *i+1*-ый элемент, узел текущей ветви выполнения, на созданную копию.
    - Заменим в списке потомков созданной копии узел-условный оператор на узел текущей ветви.

Схема данной операции представлена на рис. 10.

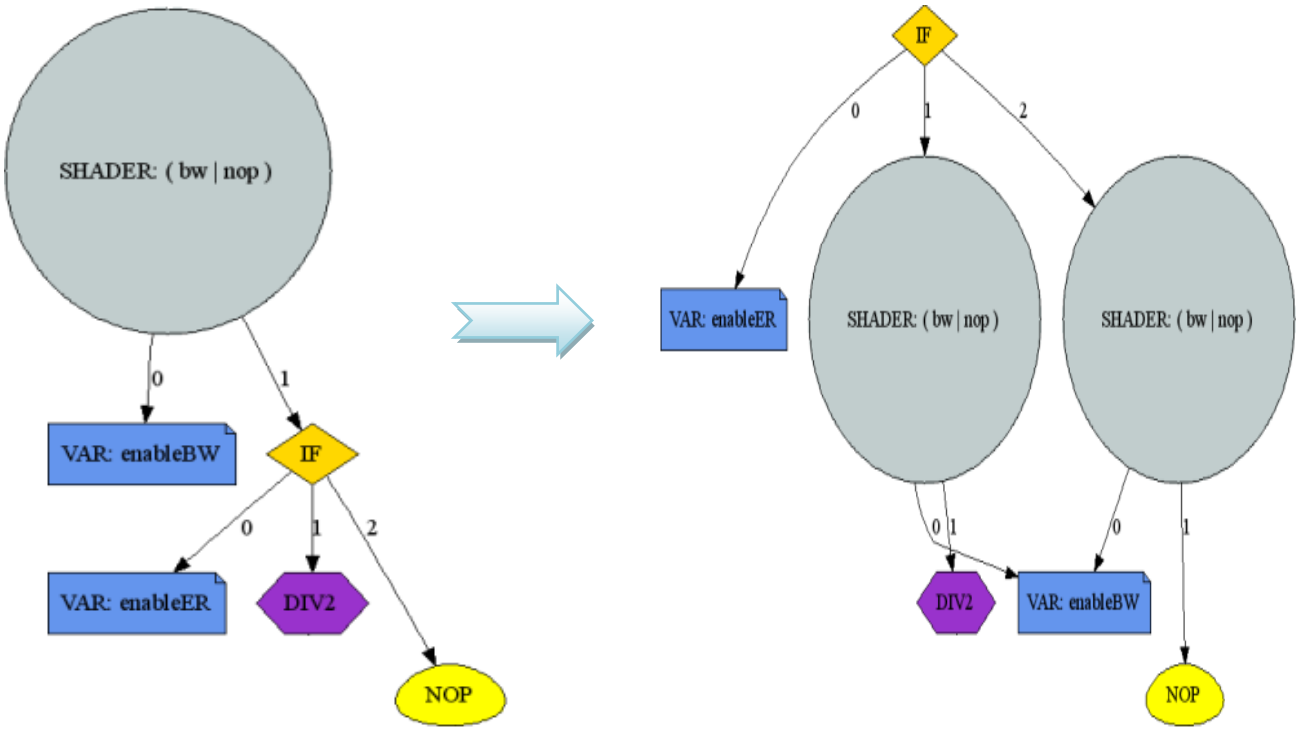

Рис. 10. Схема внесения шейдера под ветви условного оператора

2.3.5. Узел с арифметической операцией

Метод проведения оптимизаций для узла с арифметической операцией похож на представленный в разд. 2.3.4. и на рис. 9 метод объединения последовательности из двух шейдеров. Он зависит от того, является ли операция унарной или бинарной.

- Создадим новый узел-шейдер и заменим им текущий узел в списке потомков узла-предка. Алгоритм генерации кода объединенного шейдера будет рассмотрен ниже.
- Добавим в список потомков нового узла всех потомков первого шейдера-потомка.
- Для бинарных операций добавим в список потомков нового узла всех потомков второго шейдера-потомка.
- Для бинарных операций удалим из списка потомков нового узла все дубликаты, если они имеются.

Схема данной операции представлена на рис. 11.

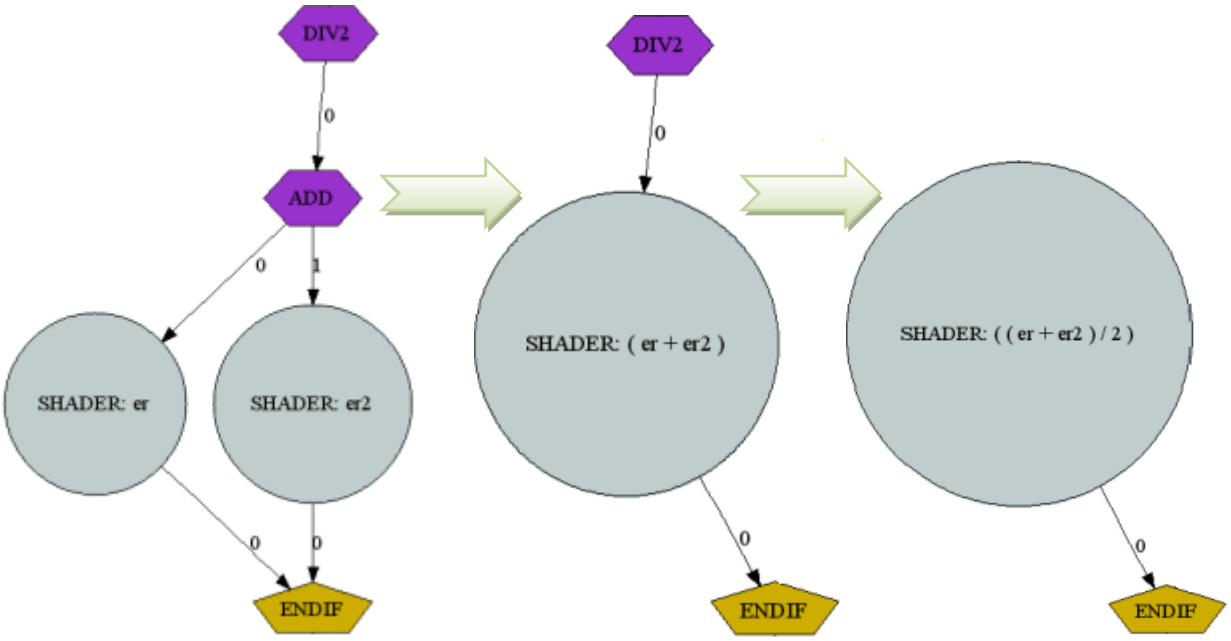

Рис. 11. Схема изменения арифметических узлов с бинарной и унарной операциями

2.3.6. Узел с началом ветвления

Существует возможность внесения условного оператора внутрь кода шейдера. Проверим, что в каждой своей ветви выполнения, заканчивающейся узлом окончания ветвления, есть:

- либо ровно один шейдер;
- либо ровно один узел с отсутствием операции;
- либо ни одного узла.

В таком случае проведем следующие операции.

- Создадим новый узел-шейдер, если хотя бы в одной ветви выполнения есть шейдер, или узел с отсутствием операции в противном случае.
- Если был создан узел-шейдер, добавим ему в качестве потомка узел с переменной, отвечающей за выбор ветви выполнения.
- Добавим в качестве потомков новому узлу всех потомков шейдеров, встречающихся в ветвях выполнения.
- Удалим дубликаты из потомков нового узла, если они имеются.

- Заменим в списке потомков предка узла с началом ветвления текущий узел на вновь созданный.

Схема данной операции представлена на рис. 12.

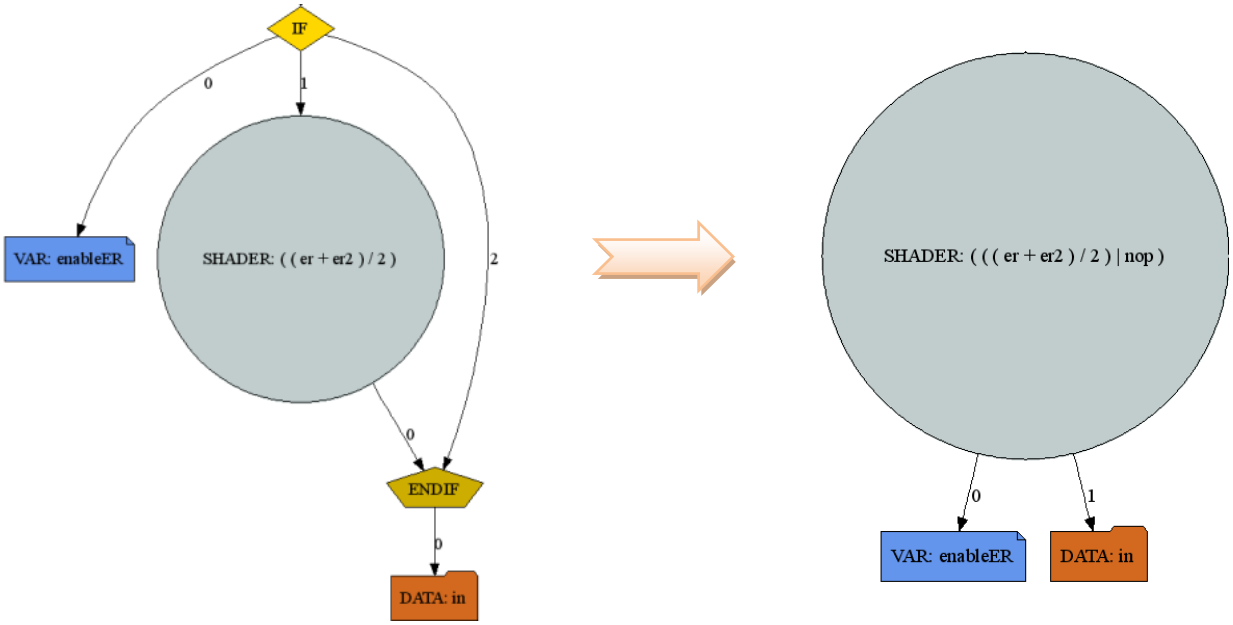

Рис. 12. Схема внесения условного оператора внутрь кода шейдера

2.3.7. Остальные типы узлов

Для остальных типов узлов не требуется выполнения каких-либо специфических операций. В данных лишь производится обход их потомков.

2.3.8. Итоги оптимизации программного дерева

Последовательно применяя изложенные выше методики, мы стараемся насколько это возможно уменьшить размер программного дерева, тем самым уменьшая число операций рендеринга, которые необходимо провести для получения результата вычислений. Заметим, что при каждом действии внутри узла, кроме внесения шейдера под знак условного оператора, число узлов в дереве сокращается. Таким образом, при малом числе условных операторов или при невыполнении данного вида оптимизации алгоритм в худшем случае работает за время $O(V^2)$, где V – это число узлов в неоптимизированном дереве.

Для задачи, текстовое описание которой приводилось в разд. 2.1., удалось сократить число узлов с 23 до 9, а число шейдеров – с 5 (+ 2 на арифметические операции) до одного. Пример приведен на рис. 13.

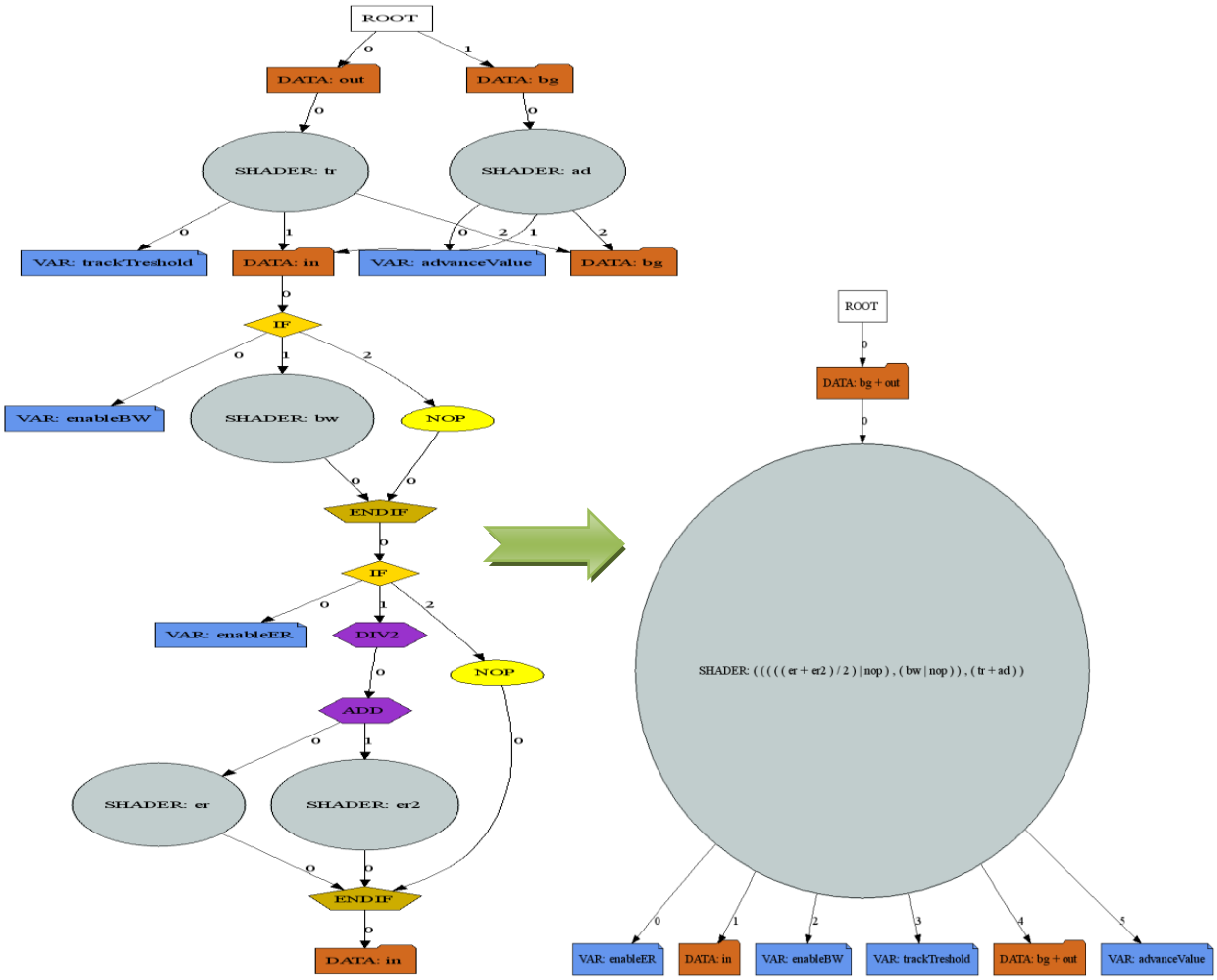

Рис. 13. Результат оптимизации программного дерева для задачи из примера

2.4. Оптимизация кода объединенного шейдера

Одним из ключевых этапов оптимизации программ, состоящих из нескольких шейдеров, является объединение двух шейдеров в один. Необходимо сгенерировать такой код нового шейдера, чтобы он был эквивалентен применению двух исходных шейдеров, последовательному или параллельному в зависимости от операции, приводящей к объединению. В работе [4] предложены схемы генерации кода для различных операций.

Отметим общие составляющие кода объединенного шейдера.

- Входные данные первого шейдера.
- Входные данные второго шейдера с удаленными дубликатами, являющиеся входными данными как первого, так и второго шейдера. Если шейдеры применяются последовательно, входная текстура, отвечающая за получение результата из первого шейдера, также удаляется.
- Внешние функции первого шейдера.
- Внешние функции второго шейдера. Заметим, что может потребоваться переименование этих функций в случае совпадения их названий с названиями внешних функций первого шейдера.
- Функция *main*
    - Код первого шейдера с заменой записи в *gl_FragColor* на запись в новую переменную-вектор.
    - Код второго шейдера. Заметим, что может потребоваться переименование переменных в случае совпадения их названий с названиями переменных первого шейдера.
    - Дополнительные операции над *gl_FragColor* в случае параллельного применения шейдеров с последующей арифметической или условной операцией.

Заметим, что хотя применение полученного шейдера и будет эквивалентным применению двух исходных, существует возможность дальнейшей оптимизации его кода. Продемонстрируем пример объединения двух шейдеров, для которого будут видны неоптимальные участки кода.

Код первого шейдера:

```
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect input_tex;
void main(void)
{
```

```
    vec4 inp1 = texture2DRect(input_tex, gl_TexCoord[0].st);
    vec4 inp2 = texture2DRect(input_tex, gl_TexCoord[0].st +
vec2(0.0, 1.0));
    gl_FragColor = inp1 * inp2;
}
```

Код второго шейдера:

```
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect input_tex;
uniform sampler2DRect shader1_tex;
uniform float         coeff;
void main(void)
{
    vec4 first = texture2DRect(shader1_tex,
gl_TexCoord[0].st);
    vec4 inp = first + texture2DRect(input_tex,
gl_TexCoord[0].st);
    gl_FragColor = inp * coeff;
}
```

Проведем слияние последовательности этих шейдеров:

```
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect input_tex;
uniform float         coeff;
void main(void)
{
    vec4 inp1 = texture2DRect(input_tex, gl_TexCoord[0].st);
    vec4 inp2 = texture2DRect(input_tex, gl_TexCoord[0].st +
vec2(0.0, 1.0));
    vec4 result = inp1 * inp2;
    vec4 first = result;
```

```
    vec4 inp = first + texture2DRect(input_tex,
gl_TexCoord[0].st);

    gl_FragColor = inp * coeff;
}
```

Отметим две существенные проблемы.

- Дублирование чтения из одной и той же текстуры по одним и тем же координатам (строки 6 и 10, текстура *input_tex*, координаты *gl_TexCoord[0].st*).
- Наличие промежуточной переменной (строка 9, переменная first дублирует переменную *result*).

Для решения первой проблемы при генерации кода шейдера запомним для каждой пары текстура-координата переменную, в которую произведено чтение. Для последующих чтений из этой же пары текстура-координата проверим, могла ли измениться эта переменная. Если не могла, заменим чтение из текстуры (вызов функции *texture2DRect*) чтением из этой переменной. Таким образом, строка 10 будет заменена на следующую строку:

```
vec4 inp = first + inp1;
```

Для решения второй проблемы обойдем все присвоения переменных, для которых больше нет последующих присвоений, и проверим следующие условия:

- инициализируются ли эти переменные напрямую другими переменными (*a = b;*);
- имеют ли эти переменные идентичную другим ранее объявленным переменным инициализацию (*a = expression; b = expression;*).

В случае соблюдения одного из двух условий заменим все последующие операции чтения из этих переменных на чтения из переменных-дубликатов. Таким образом, строка 9 будет удалена, а строка 10 примет следующий вид:

```
vec4 inp = result + inp1;
```

Итоговый оптимизированный код шейдера будет выглядеть следующим образом:

```
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect input_tex;
uniform float         coeff;
void main(void)
{
    vec4 inp1 = texture2DRect(input_tex, gl_TexCoord[0].st);
    vec4 inp2 = texture2DRect(input_tex, gl_TexCoord[0].st +
vec2(0.0, 1.0));
    vec4 result = inp1 * inp2;
    vec4 inp = result + inp1;
    gl_FragColor = inp * coeff;
}
```

Имеется возможность и дальше проводить оптимизации кода, избавляясь от некоторых переменных, но с этим может справиться оптимизирующий компилятор шейдеров.

С узлом шейдера в программном дереве связывается код соответствующего шейдера. При проведении оптимизаций, затрагивающих узлы шейдеров, требуется модифицировать код этих шейдеров. В таком случае сначала проводится стандартная генерация кода нового шейдера, а затем – оптимизация полученного кода. В дальнейшем при объединении сгенерированного шейдера с другим будет использоваться новый код, над которым могут быть продолжены оптимизации.

Выводы по главе 2

- Рассмотрена схема построения программного дерева на основе текстового описания *GPGPU*-программы.
- Рассмотрен алгоритм оптимизации программного дерева, позволяющий получить более простые и быстрые *GPGPU*-программы.
- Рассмотрены методы оптимизации кода шейдера, полученного объединением двух шейдеров.

Глава 3. Обзор библиотеки для упрощения взаимодействия с *GPU*

В этой главе будет рассмотрена авторская *GPGPU*-библиотека на языке *C++*, целью которой является упрощение взаимодействия с графическими процессорами. Упрощение достигается за счет внесения в библиотечный код всех операций по инициализации, созданию основных *OpenGL*-объектов и запуску вычислений.

3.1. Использование *API OpenGL* для взаимодействия с *GPU*

Как уже было показано ранее, взаимодействие центрального и графического процессоров требует использования одной из стандартных графических библиотек; в данной работе используется библиотека *OpenGL*. Данная библиотека предоставляет набор функций в стиле языка программирования *C*. Большинство сущностей описывается числовыми идентификаторами и константами. Для корректной работы с *GPGPU*-вычислениями требуется также подключение некоторых расширений *OpenGL*.

Приведем основные функции, которые используются для взаимодействия с графическим процессором. Мы будем использовать популярные библиотеки, упрощающие работу с *OpenGL*: *GLUT* [18] и *GLEW* [19].

- Инициализация:
  - *glutInit* – инициализирует *GLUT*;
  - *glewInit* – инициализирует *GLEW*;
  - *glutCreateWindow* – создает фиктивное «окно» для обработки сообщений.
- Работа с фреймбуфером:
  - *glGenFramebuffersEXT* – создает фреймбуфер;

- *glBindFramebufferEXT* – делает фреймбуфер целью рендеринга;
- *glFramebufferTexture2DEXT* – привязывает текстуру к фреймбуферу;
- *glMatrixMode* – выбирает тип модифицируемой матрицы;
- *gluOrtho2D* – создает матрицу ортографической проекции;
- *glViewport* – создает матрицу области рендеринга;
- *glDrawBuffer* – устанавливает привязанную текстуру, в которую будет произведен рендеринг;
- *glReadBuffer*, *glReadPixels* – записывают данные из привязанной текстуры в оперативную память;
- *glDeleteFramebuffersEXT* – удаляет фреймбуфер.

- Работа с текстурами:
  - *glGenTextures* – создает одну или несколько текстур и выделяет им идентификатор;
  - *glBindTexture* – устанавливает текущую текстуру, над которой далее проводятся операции;
  - *glTexParameteri*, *glTexEnvi* – устанавливают различные целочисленные параметры;
  - *glTexImage2D* – задает размеры текстуры и выделяет память под нее;
  - *glActiveTexture* – устанавливает текущий глобальный индекс текстуры для привязки к нему текстуры с последующей передачей этого индекса в шейдер;
  - *glTexSubImage2D* – записывает данные из оперативной памяти в текстуру;
  - *glGetTexImage* – записывает данные из текстуры в оперативную память;
  - *glDeleteTextures* – удаляет одну или несколько текстур.

- Работа с шейдерами:
    - *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 ему 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;
    - *glGetUniformLocation, glUniform1i, glUniform1f* – служат для установки входных данных шейдера, описанных ключевым словом *uniform*;
    - *glDetachShader* – отвязывает шейдер от программы;
    - *glDeleteShader* – удаляет шейдер;
    - *glDeleteProgram* – удаляет программу.
- Запуск рендеринга:
    - *glPolygonMode* – устанавливает режим рендеринга полигонов;
    - *glBegin* – начинает процесс рендеринга;
    - *glTexCoord2f* – устанавливает текстурные координаты текущей вершины;
    - *glVertex2f* – устанавливает координаты текущей вершины;

- *glEnd* – завершает процесс рендеринга.

- Завершение выполнения:
  - *glutDestroyWindow* – удаляет созданное фиктивное «окно».

Как мы видим, требуется использовать множество функций, а *C*-подобный интерфейс усложняет работу в объектно-ориентированной среде. Например, код для выполнения задачи из примера занимает около 600 строк без учета комментариев.

3.2. Пример использования авторской библиотеки

Целью автора было создание простой и интуитивно-понятной библиотеки на языке *C++* и использованием объектно-ориентированного подхода. Приведем пример кода, который проводит вычисления, решающие тестовую задачу из работы [4].

```
// init
gpgpu::Core::init();
_frameBuffer = gpgpu::FrameBuffer::create(_width, _height);
for (int i = texIn; i < texMax; ++i)
{
    _textures[i] = gpgpu::Texture::create(gpgpu::Texture::Format::RGBA, w, h);
}
_frameBuffer->attachTexture(_textures[texTracking], attTracker);
_shaders[shTracker]  = gpgpu::Shader::create("shaders\\tracker.frag");
_shaders[shTracker]->setArgument("img", _textures[texIn]);
_shaders[shTracker]->prepareArgument("bg", gpgpu::Shader::AT_TEXTURE);
_shaders[shTracker]->setArgument("diff", 0.2f);
_shaders[shTracker]->setArgument("width", (float)_width);
// compute
_textures[texIn]->write(img);
_shaders[shTracker]->setArgument("bg", _textures[texBgOld]);
gpgpu::Core::inst()->compute(_textures[texTracking], _shaders[shTracker]);
_textures[texTracking]->read(out, gpgpu::Texture::IOFormat::A);
// term
gpgpu::Core::term();
```

Данный пример содержит весь необходимый код для инициализации, настройки, проведения и завершения вычислений. Легко заметить, что число

строк кода по сравнению с традиционным решением значительно сократилось. Также была улучшена читаемость кода.

3.3. Обзор этапов взаимодействия с *GPU* при помощи авторской библиотеки

Разделим продемонстрированный выше пример на этапы взаимодействия с *GPU*.

Сначала происходит инициализация библиотеки и создание необходимых объектов (фреймбуферов, текстур и шейдеров).

- Инициализация ядра (класс *gpgpu::Core*):

```
gpgpu::Core::init();
```

- Создание фреймбуфера (класс *gpgpu::FrameBuffer*):

```
auto _frameBuffer = gpgpu::FrameBuffer::create(w, h);
```

- Создание необходимого количества текстур (класс *gpgpu::Texture*):

```
for (int i = texIn; i < texMax; ++i)
    _textures[i] = gpgpu::Texture::create(Format::RGBA, w, h);
```

- Привязка текстур к фреймбуферу:

```
_frameBuffer->attachTexture(_textures[texTracking], attTracker);
```

- Создание необходимых шейдеров из их исходного кода (класс *gpgpu::Shader*):

```
auto _shader = gpgpu::Shader::create("shaders\\tracker.frag");
```

- Привязка постоянных параметров шейдеров и подготовка переменных:

```
_shader->setArgument("img", _textures[texIn]);
_shader->prepareArgument("bg", AT_TEXTURE);
```

На данном этапе библиотека готова к проведению расчетов. Целесообразно вынести приведенные ниже операции в отдельную функцию, которая и будет производить вычисления.

- Запись входных данных в текстуру:

```
_textures[texIn]->write(img);
```

- Установка переменных параметров:

```
_shaders[shTracker]->setArgument("bg", _textures[texBgOld]);
```

- Запуск шейдера:

```
gpgpu::Core::inst()->compute(_textures[texTracking], _shaders[shTracker]);
```

- Чтение результата из текстуры:

```
_textures[texTracking]->read(out, IOFormat::A);
```

- Обмен текстур для чтения и записи (в случае использования пары текстур для чтения-записи одного массива данных):

```
_textures[texBgNew].swap(_textures[texBgOld]);
```

При завершении программы необходимо завершить работу библиотеки.

- Завершение работы ядра:

```
gpgpu::Core::term();
```

Отметим, что в библиотеке используются «умные» указатели со счетчиком ссылок. Все созданные объекты будут удалены в корректном порядке автоматически, когда на них не останется ссылок.

Выводы по главе 3

- Рассмотрены основные функции библиотеки *OpenGL*, используемые для взаимодействия с графическим процессором.
- Показана сложность прямой работы с *OpenGL*.
- Написана библиотека на языке *C++*, позволяющая существенно упростить взаимодействие с графическим процессором.

Глава 4. Использование связки из *CPU* и *GPU* для проведения высокопроизводительных вычислений

В этой главе будет решен недостаток блокировки центрального процессора на время выполнения вычислений графическим процессором. Будут показаны способы оптимизированного выполнения программ для графического процессора на центральном процессоре параллельно с графическим.

4.1. Избавление от блокировки *CPU*

Как уже было отмечено выше, процессы записи данных в текстуру, проведения вычислений на графическом процессоре и чтения данных из текстуры являются блокирующими для центрального процессора – он не может продолжать дальнейшую работу без получения корректного результата. Таким образом, происходит простой мощного вычислителя, который мог бы быть использован для решения части поставленной задачи.

Для избавления от блокировки необходимо производить вычисления на графическом процессоре и операции чтения-записи в одном *потоке* (*thread*), а вычисления на центральном процессоре – в другом. Более того, можно использовать столько потоков для центрального процессора, сколько вычислительных ядер он содержит, распределив объем обрабатываемых данных в равных пропорциях между ними.

Для упрощения работы с потоками можно воспользоваться паттерном *пул потоков* (*Thread Pool*). Заранее создадим $N + 1$ поток, где N – количество вычислительных ядер центрального процессора, а дополнительный поток служит для запуска вычислений на графическом процессоре. Когда необходимо произвести вычисления, распределим объем данных для каждого вычислителя и отправим каждому потоку команду на проведение вычислений. В конце вычислительных функций необходимо установить

барьеры для оповещения основного потока о завершении вычислений, после которого потоки-работники (*Worker Threads*) приостанавливаются до получения новых команд на проведение вычислений.

4.2. Выполнение одинаковых вычислений на *CPU* и *GPU*

Выше мы установили, что при решении задачи можно использовать центральный процессор параллельно с графическим для проведения одинаковых вычислений над распределенными объемами данных. Далее необходимо понять, каким образом можно описать задачу для центрального процессора, используя уже существующее описание для графического процессора.

Одним из подходов является написание отдельного кода, который будет выполнен на центральном процессоре. Для этого нам необходимо описать алгоритм, например, на языке *C++*. Отметим проблемы такого подхода:

- требуются трудозатраты программиста на написание нового кода;
- существует вероятность ошибки в коде, что может привести к различным результатам выполнения вычислений на центральном и графическом процессорах;
- требуется описать алгоритм так, чтобы его можно было легко распараллелить между вычислителями;
- зачастую может потребоваться создание отдельных структур данных, например, для представления четырехкомпонентного вектора вещественных чисел; более того, код таких структур может быть неоптимизированным.

Как мы видим, данный подход имеет существенные недостатки, которые являются следствием неунифицированности кода, выполняемого на центральном и графическом процессорах.

Рассмотрим другой подход: использование существующего кода программ для графического процессора для выполнения на центральном процессоре. Однако в силу особенностей графического процессора и языков программирования для него мы не можем просто скомпилировать код шейдера как код языка *C* – некоторые конструкции придется определить самостоятельно или модифицировать. В языке *GLSL* проблемными с этой точки зрения являются следующие конструкции:

- директивы препроцессора *#extension*;
- переменные, объявленные с ключевым словом *uniform*;
- встроенные типы данных, например, *vec4*;
- встроенные функции, например, *length*;
- встроенные ключевые переменные, например, *gl_TexCoord*, *gl_FragColor*;
- встроенная в векторные типы операция перемешивания компонентов (*swizzle*), например, *v.xzzy*.

Реализация или модификация этих конструкций позволит избавиться от недостатков, характерных для написания отдельного кода. Отметим необходимые действия:

- реализация стандартных типов данных с сохранением их особенностей;
- реализация стандартных функций;
- проведение некоторых модификаций кода шейдера.

Способ реализации данных действий для языка *C++* будет приведен ниже.

4.3. Инструкции *SSE*

Как уже было сказано выше, процессоры архитектуры *x86* поддерживают набор векторных инструкций *SSE*. Эти инструкции работают со 128-битными процессорными регистрами *XMM0-XMM7* (*XMM0-XMM15*

для *x64*). Такие регистры могут хранить вектор из четырех вещественных чисел, что позволяет записывать в него ячейку текстуры, которая также является четырехкомпонентным вектором.

Набор инструкций позволяет проводить различные операции над такими векторами. Условно можно разделить такие операции на вертикальные и горизонтальные: вертикальные операции выводят результат из двух исходных векторов покомпонентно, а горизонтальные обрабатывают компоненты одного вектора и записывают результат в один или все компоненты целевого вектора (рис. 14). Существуют также смешанные операции, например, операция скалярного произведения.

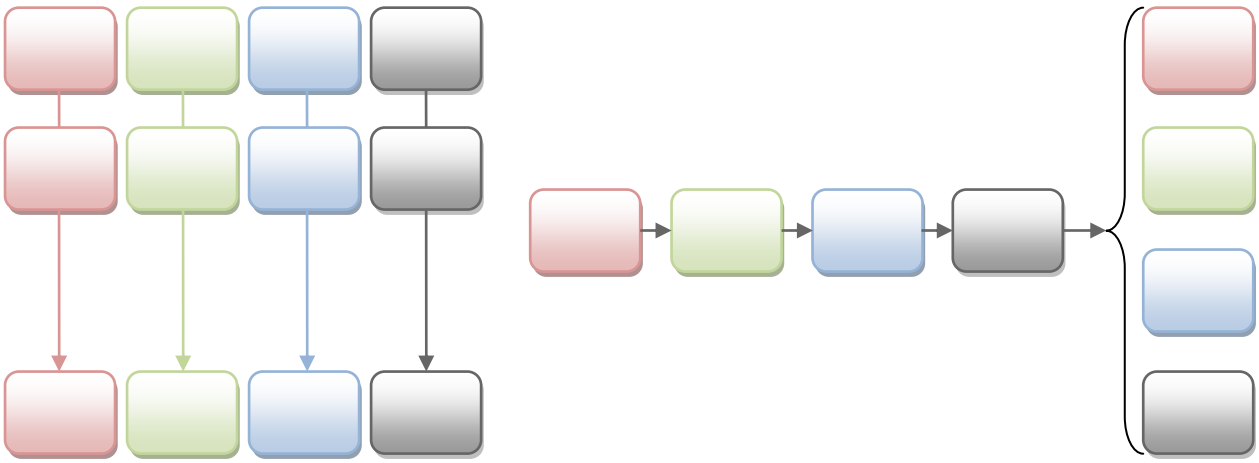

Рис. 14. Вертикальные и горизонтальные операции

Отметим, что существует несколько версий наборов инструкций *SSE* и не все инструкции могут быть доступны на всех центральных процессорах. Например, горизонтальные операции появились в *SSE3*, а операция скалярного произведения – в *SSE4*, доступном в процессорах *Intel*, начиная с *Core 2 Duo*.

Инструкции *SSE* могут быть использованы компилятором автоматически в случае включения соответствующих опций оптимизации, однако существует возможность их ручного использования благодаря применению *intrinsic*-функций – функций, вызов которых напрямую преобразуются в ассемблерный код *SSE*-инструкции. Для всех современных

SSE-инструкций в компиляторе *Microsoft 32-bit C/C++ Optimizing Compiler* представлены соответствующие *intrinsic*-функции. Также необходимо использовать специальный тип данных, *__m128*, представляющий 128-битный вектор. *Intrinsic*-функции используют этот тип данных для принимаемых параметров и возвращаемых значений. Отметим, что объекты данного типа должны располагаться в памяти по адресу, кратному 16 байтам. Данное ограничение позволяет процессору ускорить работу с векторами вещественных чисел.

4.4. Реализация стандартных структур данных и функций языка *GLSL* на языке *C++*

4.4.1. Основы реализации вектора

Рассмотрим реализацию стандартных структур данных на примере четырехкомпонентного вектора вещественных чисел *vec4*. Отметим основные операции, которые нам необходимо реализовать:

- создание вектора из четырех вещественных чисел или другого вектора:

```
vec4 bwVec = vec4(0.56, 0.33, 0.11, 0.0);
vec4 v     = bwVec;
```

- чтение и запись компонентов вектора; отметим, что доступ к одному компоненту может быть осуществлен с использованием различных идентификаторов (например, *x*, *s* или *r* для первого компонента):

```
float x = vbg.x;
vbg.r   = 1.0;
```

- арифметические операторы:

```
vec4 result = v1 + v2;
```

- *swizzle*:

```
vec4 v4 = v.xzzy;
```

```
vec2 v2 = v.rg;
```

Мы будем применять *intrinsic*-функции и тип данных *__m128* для получения кода, оптимизированного под целевую архитектуру *x86*. Далее мы увидим, какой прирост производительности дает их использование.

Создадим структуру *vec4* и укажем, что ее адрес должен быть упорядочен до 16 байт. Данное указание зависит от используемого компилятора; для *Microsoft 32-bit C/C++ Optimizing Compiler* код будет выглядеть следующим образом:

```
__declspec(align(16)) struct vec4
{
};
```

4.4.2. Реализация создания вектора и чтения-записи его компонентов

Поместим в созданную структуру объединение (*union*) – область в памяти, которую можно интерпретировать как объект одного из нескольких описанных типов. Мы включим туда следующие описания типов:

- `struct { float x, y, z, w; };` – структура для чтения-записи векторных компонентов *x*, *y*, *z* и *w*;
- `struct { float s, t, u, v; };` – структура для чтения-записи векторных компонентов *s*, *t*, *u* и *v*;
- `struct { float r, g, b, a; };` – структура для чтения-записи векторных компонентов *r*, *g*, *b* и *a*;
- `float array[4];` – массив из четырех вещественных чисел для доступа к вектору как к массиву;
- `__m128 hw;` – описанная выше структура, благодаря которой имеется возможность использования инструкций *SSE*;
- `NEST_SHUFFLERS(4)` – специальный макрос, добавляющий в объединение необходимые структуры для реализации операции *swizzle*; о нем будет сказано ниже.

Теперь необходимо реализовать конструкторы и операторы присваивания. Их реализация будет описана в теле структуры и будет помечена ключевым словом *inline*, которое указывает компилятору, что по возможности необходимо встраивать код функции прямо в код, вызывающий данную функцию, вместо проведения операции вызова. Данная операция позволяет ускорить выполнение кода. В конструкторах и операторах присваивания мы будем работать с переменной *hw* типа *__m128*. После присваивания ей нового значения другие объекты объединения автоматически получат верные значения – они хранятся по тому же адресу в памяти. Реализуемые конструкторы и операторы присваивания описаны ниже:

- `inline vec4() : hw(_mm_setzero_ps()) {}` – создание нового вектора и заполнение его компонентов нулями;
- `inline vec4(const vec4& v) : hw(v.hw) {}` – создание нового вектора на основе другого вектора;
- `inline vec4(__m128 hw) : hw(hw) {}` – создание нового вектора на основе структуры *__m128*; данная операция в основном применяется в других функциях, например при реализации *swizzle*;
- `inline vec4(const float f) : hw(_mm_set_ps1(f)) {}` – создание нового вектора с заполнением всех его компонентов указанным вещественным числом;
- `inline vec4(const float x, const float y, const float z, const float w) : hw(_mm_set_ps(w, z, y, x)) {}` – создание нового вектора из четырех указанных вещественных чисел;
- `inline vec4& operator=(const vec4& v) { hw = v.hw; return *this; }` – присваивание текущему вектору всех компонентов другого вектора;

- `inline vec4& operator=(__m128 ohw) { hw = ohw; return *this; }` – присваивание текущему вектору всех компонентов структуры *__m128*.

4.4.3. Реализация операторов вектора

Реализуем операторы сравнения:

- `inline bool operator==(const vec4& v) const { return x == v.x && y == v.y && z == v.z && w == v.w; }` – проверка покомпонентного равенства двух векторов;
- `inline bool operator!=(const vec4& v) const { return x != v.x || y != v.y || z != v.z || w != v.w; }` – проверка неравенства хотя бы одного компонента у двух векторов.

Реализуем арифметические операторы:

- `inline vec4 operator+() const { return *this; }` – унарный плюс;
- `inline vec4 operator-() const { return vec4(_mm_sub_ps(zero.hw, hw)); }` – унарный минус; здесь используется константный вектор, состоящий из нулей;
- `inline vec4 operator+(const vec4& other) const { return vec4(_mm_add_ps(hw, other.hw)); }` – покомпонентное сложение;
- `inline vec4 operator-(const vec4& other) const { return vec4(_mm_sub_ps(hw, other.hw)); }` – покомпонентное вычитание;
- `inline vec4 operator*(const vec4& other) const { return vec4(_mm_mul_ps(hw, other.hw)); }` – покомпонентное умножение;
- `inline vec4 operator/(const vec4& other) const { return vec4(_mm_div_ps(hw, other.hw)); }` – покомпонентное деление;

- `inline vec4 operator*(const float f) const { return vec4(_mm_mul_ps(hw, _mm_set_ps1(f))); }` – покомпонентное умножение на число (число – справа);
- `inline vec4& operator+=(const vec4& other) { hw = _mm_add_ps(hw, other.hw); return *this; }` – покомпонентное сложение с записью в текущий вектор;
- `inline vec4& operator-=(const vec4& other) { hw = _mm_sub_ps(hw, other.hw); return *this; }` – покомпонентное вычитание с записью в текущий вектор;
- `inline vec4& operator*=(const vec4& other) { hw = _mm_mul_ps(hw, other.hw); return *this; }` – покомпонентное умножение с записью в текущий вектор;
- `inline vec4& operator/=(const vec4& other) { hw = _mm_div_ps(hw, other.hw); return *this; }` – покомпонентное деление с записью в текущий вектор;
- `inline vec4& operator*=(const float f) { hw = _mm_mul_ps(hw, _mm_set_ps1(f)); return *this; }` – покомпонентное умножение на число с записью в текущий вектор.

Также требуется реализовать некоторые операции вне тела структуры:

- `inline vec4 operator*(const float f, const vec4& v) { return vec4(_mm_mul_ps(_mm_set_ps1(f), v.hw)); }` – покомпонентное умножение на число (число – слева);
- `DECL_SHUFFLERS(4)` – специальный макрос, объявляющий необходимые структуры для реализации операции *swizzle*; о нем будет сказано ниже;
- `IMPL_SHUFFLERS(4)` – специальный макрос, реализующий необходимые структуры для реализации операции *swizzle*; о нем будет сказано ниже.

4.4.4. Итоги реализации вектора без операции *swizzle*

Мы произвели первую, вторую и третью операции, описанные в разд. 4.4.1. Итоговый код реализации структуры *vec4* приведен ниже:

```
DECL_SHUFFLERS(4)

__declspec(align(16)) struct vec4
{
union
{
    struct { float x, y, z, w; };
    struct { float s, t, u, v; };
    struct { float r, g, b, a; };
    float  array[4];
    __m128 hw;
    NEST_SHUFFLERS(4)
};

inline vec4() : hw(_mm_setzero_ps()) {}
inline vec4(const vec4& v) : hw(v.hw) {}
inline vec4(__m128 hw) : hw(hw) {}
inline vec4(const float f) : hw(_mm_set_ps1(f)) {}
inline vec4(const float x, const float y, const float z,
const float w) : hw(_mm_set_ps(w, z, y, x)) {}
inline vec4& operator=(const vec4& v) { hw = v.hw; return
*this; }
inline vec4& operator=(__m128 ohw) {hw = ohw; return *this;}

inline bool operator==(const vec4& v) const { return x ==
v.x && y == v.y && z == v.z && w == v.w; }
inline bool operator!=(const vec4& v) const { return x !=
v.x || y != v.y || z != v.z || w != v.w; }

inline vec4 operator+() const { return *this; }
```

```
    inline vec4 operator-() const { return
vec4(_mm_sub_ps(zero.hw, hw)); }

    inline vec4 operator+(const vec4& other) const { return
vec4(_mm_add_ps(hw, other.hw)); }
    inline vec4 operator-(const vec4& other) const { return
vec4(_mm_sub_ps(hw, other.hw)); }
    inline vec4 operator*(const vec4& other) const { return
vec4(_mm_mul_ps(hw, other.hw)); }
    inline vec4 operator/(const vec4& other) const { return
vec4(_mm_div_ps(hw, other.hw)); }
    inline vec4 operator*(const float f)       const { return
vec4(_mm_mul_ps(hw, _mm_set_ps1(f))); }

    inline vec4& operator+=(const vec4& other) { hw =
_mm_add_ps(hw, other.hw); return *this; }
    inline vec4& operator-=(const vec4& other) { hw =
_mm_sub_ps(hw, other.hw); return *this; }
    inline vec4& operator*=(const vec4& other) { hw =
_mm_mul_ps(hw, other.hw); return *this; }
    inline vec4& operator/=(const vec4& other) { hw =
_mm_div_ps(hw, other.hw); return *this; }
    inline vec4& operator*=(const float f)          { hw =
_mm_mul_ps(hw, _mm_set_ps1(f)); return *this; }
    };
    inline vec4 operator*(const float f, const vec4& v) { return
vec4(_mm_mul_ps(_mm_set_ps1(f), v.hw)); }

    IMPL_SHUFFLERS(4)
```

4.4.5. Реализация операции *swizzle*

Теперь необходимо выполнить последний пункт из разд. 4.4.1.: реализовать операцию *swizzle*. Для этого нам потребуется различные структуры, которые будут хранить в себе переменную типа *__m128* и для

которых будет определен оператор приведения к *vec4*. Число таких структур равно сумме числа слов из четырех символов в алфавитах *{x, y, z, w}*, *{s, t, u, v}*, *{r, g, b, a}*. Данные структуры будут храниться в объединении с именами соответствующих перестановок, например, `Shuffler_xzzy xzzy`. При чтении такой структуры будет произведено преобразование вектора при помощи *intrinsic*-функции *_mm_shuffle_ps*, принимающей два вектора (мы будем использовать один и тот же) и маску перестановки.

Для реализации специальных макросов `DECL_SHUFFLERS(4)`, `NEST_SHUFFLERS(4)` и `IMPL_SHUFFLERS(4)`, которые упоминались выше, необходимо завести вспомогательные константы, на основе которых будет строиться маска для *intrinsic*-функции *_mm_shuffle_ps*. Код построения маски приведен ниже:

```
struct ShuffleConst
{
static const unsigned int x = 0;
static const unsigned int y = 1;
static const unsigned int z = 2;
static const unsigned int w = 3;

static const unsigned int s = 0;
static const unsigned int t = 1;
static const unsigned int u = 2;
static const unsigned int v = 3;

static const unsigned int r = 0;
static const unsigned int g = 1;
static const unsigned int b = 2;
static const unsigned int a = 3;

static const unsigned int none = 0;
};
```

```
template <unsigned int x, unsigned int y, unsigned int z,
unsigned int w>
struct ShuffleMask
{
static const unsigned int value = x | (y << 2) | (z << 4) |
(w << 6);
};
```

Далее при помощи макроса перечислим все возможные перестановки (данный код целесообразно генерировать программно – число таких перестановок велико):

```
#define ENUM_SHUFFLERS_FOR_VEC4(PREFIX)      \
PREFIX##_SHUFFLER_TO_VEC4(x, x, x, x);       \
PREFIX##_SHUFFLER_TO_VEC4(x, x, x, y);       \
…
PREFIX##_SHUFFLER_TO_VEC4(w, w, w, z);       \
PREFIX##_SHUFFLER_TO_VEC4(w, w, w, w);
```

Воспользуемся этим списком в макросах `DECL_SHUFFLERS(4)`, `NEST_SHUFFLERS(4)` и `IMPL_SHUFFLERS(4)`:

```
#define DECL_SHUFFLERS(N)    ENUM_SHUFFLERS_FOR_VEC##N(DECL)
#define NEST_SHUFFLERS(N)    ENUM_SHUFFLERS_FOR_VEC##N(NEST)
#define IMPL_SHUFFLERS(N)    ENUM_SHUFFLERS_FOR_VEC##N(IMPL)
```

Осталось провести следующие операции:

- объявить необходимые структуры в макросе `DECL_SHUFFLER_TO_VEC4(x, y, z, w);`
- добавить переменные в объединение в макросе `NEST_SHUFFLER_TO_VEC4(x, y, z, w);`

- определить реализацию перестановки в макросе `IMPL_SHUFFLER_TO_VEC4_C(x, y, z, w)`.

Данные макросы принимают имена компонентов. Например, для перестановки *v.rbag* необходимо вызвать макросы `****_SHUFFLER_TO_VEC4(r, b, a, g)`.

В объявлении структуры мы опишем входящую в нее переменную типа *__m128* и объявим оператор преобразования в *vec4*:

```
#define DECL_SHUFFLER_TO_VEC4(x, y, z, w) struct Shuffler_##x##y##z##w {__m128 hw; inline operator vec4() const;}
```

Для добавления переменной в объединение укажем ее корректный тип и имя:

```
#define NEST_SHUFFLER_TO_VEC4(x, y, z, w) Shuffler_##x##y##z##w x##y##z##w;
```

Наконец, для реализации операции *swizzle* воспользуемся указанным выше кодом построения маски:

```
#define IMPL_SHUFFLER_TO_VEC4_C(x, y, z, w) inline Shuffler_##x##y##z##w::operator vec4() const { return vec4(hw); }
```

Таким образом, был реализован класс четырехкомпонентного вектора для *CPU*, аналогичный используемому на *GPU*. Другие встроенные типы реализуются аналогично.

4.4.6. Реализация стандартных функций

Для реализации стандартных функций мы также будем использовать *intrinsic*-функции и тип *__m128*. Реализуемые нами функции будут работать с

созданными структурами данных, например, *vec4*. Приведем два примера таких функций (скалярное произведение и нахождение минимума):

```
inline float dot(const vec4& v1, const vec4& v2)
{
float result;
_mm_store_ss(&result, _mm_dp_ps(v1.hw, v2.hw, 0xF1));
return result;
}

inline vec4 min(const vec4& v1, const vec4& v2)
{
return vec4(_mm_min_ps(v1.hw, v2.hw));
}
```

Отметим, что графический процессор работает с 32-битными вещественными числами, тогда как центральный процессор оперирует с 64-битными вещественными числами, даже при использовании типа *float*: при использовании этого типа результирующее число обрезается до 32 бит, но вычисления проводятся над 64-битными числами. Для того чтобы сделать поведение *CPU* схожим с *GPU*, а также для увеличения производительности необходимо перевести вещественный процессор (*FPU*) в режим одинарной точности. Это можно сделать при помощи *intrinsic*-функции *_controlfp*:

- `_controlfp( _PC_24, MCW_PC );` – перевод *FPU* в режим одинарной точности;
- `_controlfp( _CW_DEFAULT, 0xfffff );` – возврат *FPU* в режим двойной точности.

4.5. Произведение изменений в коде шейдера для его запуска на *CPU*

Код шейдера мы будем записывать в заголовочный файл (*.h*), предварительно проведя некоторые его модификации. Данный файл будет использоваться при компиляции проекта, выполняющего вычисления. Достоинства и недостатки данного подхода будут изложены ниже.

- В начале заголовочного файла подключим заголовочные файлы с реализацией стандартных структур данных и функций.
- Удалим директиву препроцессора *#extension*.

Далее необходимо научиться передавать входные данные в шейдер. Наиболее простым способом является оборачивание всего кода шейдера в структуру. Также необходимо избавиться от ключевого слова *uniform*, неподдерживаемого компилятором для *CPU*.

- В начале заголовочного файла добавим строку, тем самым отбросив ключевое слово *uniform* при компиляции:
  ```
  #define uniform
  ```
- Перед определением переменных добавим строку (вместо *«имя_шейдера»* необходимо подставить уникальное имя шейдера):
  ```
  struct имя_шейдера {
  ```
- После последней строки кода шейдера добавим строку:
  ```
  };
  ```

Для использования шейдера необходимо объявить переменную созданного типа, а для передачи входных параметров – произвести запись в член созданной структуры. Отметим, что запись производится до запуска вычислений, что избавляет от проблем с использованием нескольких потоков.

Далее требуется передавать в шейдер специальные переменные: *gl_TexCoord* и *gl_FragColor*. Эти переменные уникальны для каждого запуска шейдера, поэтому их нельзя делать членами структуры. Можно передавать их в функцию шейдера *main*.

- Заменим сигнатуру функции main, передав в нее *gl_TexCoord* и *gl_FragColor*:

```
void main(void)
```

↓

```
void main(const vec4* gl_TexCoord, vec4& gl_FragColor)
```

Осталось решить одну существенную проблему. Рассмотрим внешнюю функцию какого-либо шейдера, которая принимает один или несколько встроенных векторных типов данных, например:

```
vec4 f(vec4 v1, vec4 v2)
```

В языке *C++* функция с такой сигнатурой принимает свои аргументы по значению – они копируются на стек. К сожалению, при укладке на стек компилятор не может гарантировать кратность адреса 16 байтам, поэтому происходит ошибка компиляции. Необходимо заменить передачу по значению на передачу по ссылке. Это изменение также может увеличить производительность.

- Для всех определений функций шейдера, кроме *main*, при использовании векторных типов заменим эти типы константными ссылками на них:

```
vec4 f(vec4 v1, vec4 v2)
```

↓

```
vec4 f(const vec4& v1, const vec4& v2)
```

Дополнительно рекомендуется заменить все вещественные константы, чтобы они имели тип с одинарной точностью. Для этого после константы необходимо добавить символ *f*.

- Найдем все вещественные константы и при необходимости добавим символ *f*:

```
1.0 → 1.0f
```

После проведения всех вышеописанных изменений код шейдера будет компилироваться и будет готов к использованию в вычислениях.

4.6. Сравнение производительности *CPU* и *GPU*

Для сравнения производительности центрального и графического процессоров была написана тестовая программа, производящая вычисления для тестовой задачи из работы [4]. Использовались следующие вычислители:

- центральный процессор с отдельным кодом вычислений;
- центральный процессор с модифицированным кодом шейдера и использованием *SSE*;
- несколько ядер центрального процессора с модифицированным кодом шейдера и использованием *SSE*;
- графический процессор.

Замеры производились для различных объемов входных и выходных массивов с данными: от 0.5 до 4 миллионов четырехкомпонентных векторов. Производилось несколько предварительных вычислительных итераций для исключения влияния кеширования на результат. Далее выполнялось несколько вычислительных циклов и бралось среднее время выполнения одного цикла. Результаты представлены на рис. 15.

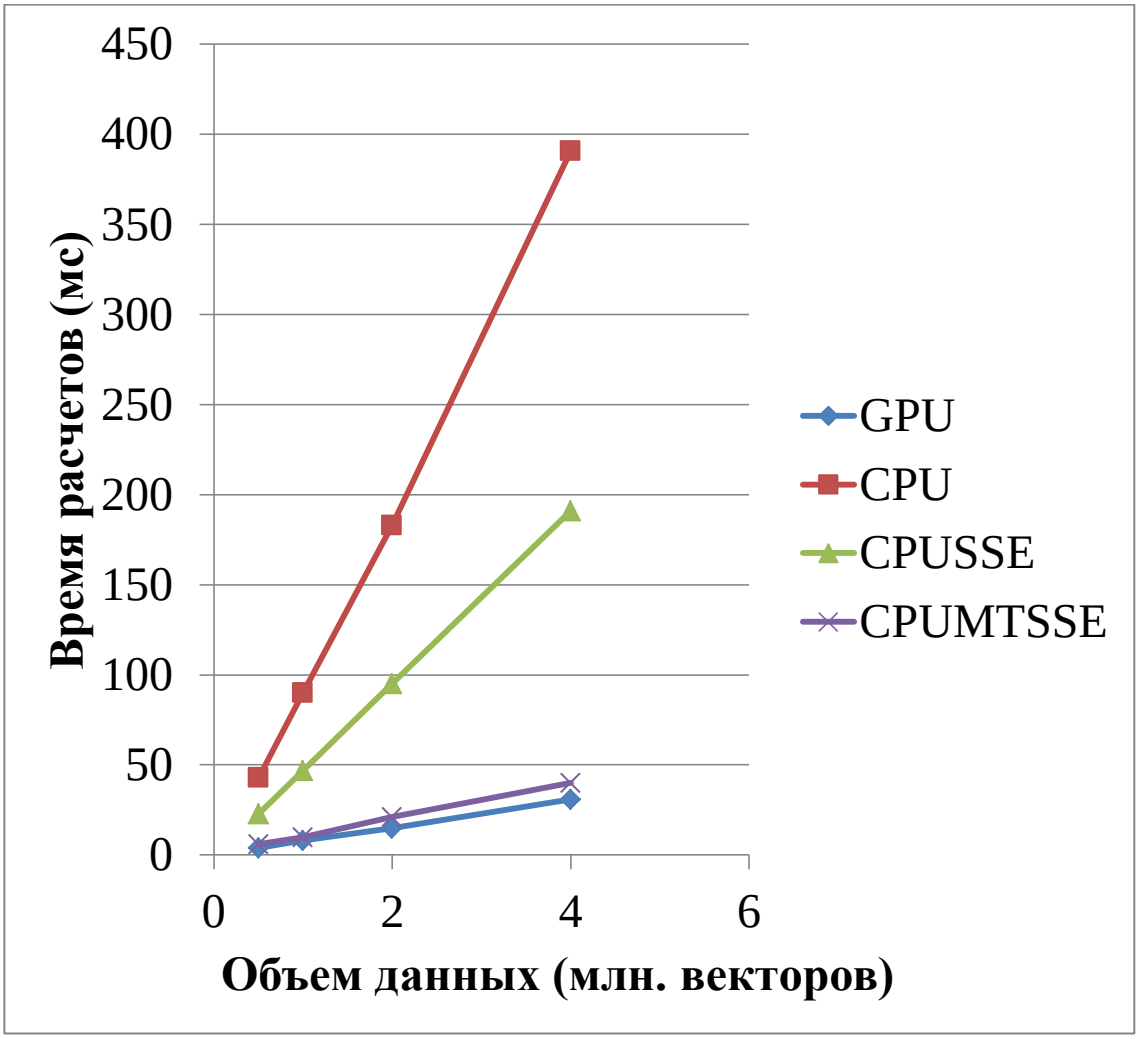

Рис. 15. Результаты замера производительности различных вычислителей

Как можно увидеть на графике, использование *SSE* дало для данной задачи двукратный прирост производительности. Использование четырех ядер центрального процессора также ускорило вычисления практически в четыре раза. Тем не менее, графический процессор все равно оказывается быстрее многоядерного центрального примерно на 25%. Отметим, что узким местом для графического процессора является передача данных между оперативной и видеопамятью. Существует ряд задач, в которых не требуется на каждой итерации выводить результат, для них время работы графического процессора будет значительно ниже, что можно увидеть на рис. 16. Примером такой задачи может быть расчет *N*-го состояния в игре «Жизнь» на основе заданного начального состояния.

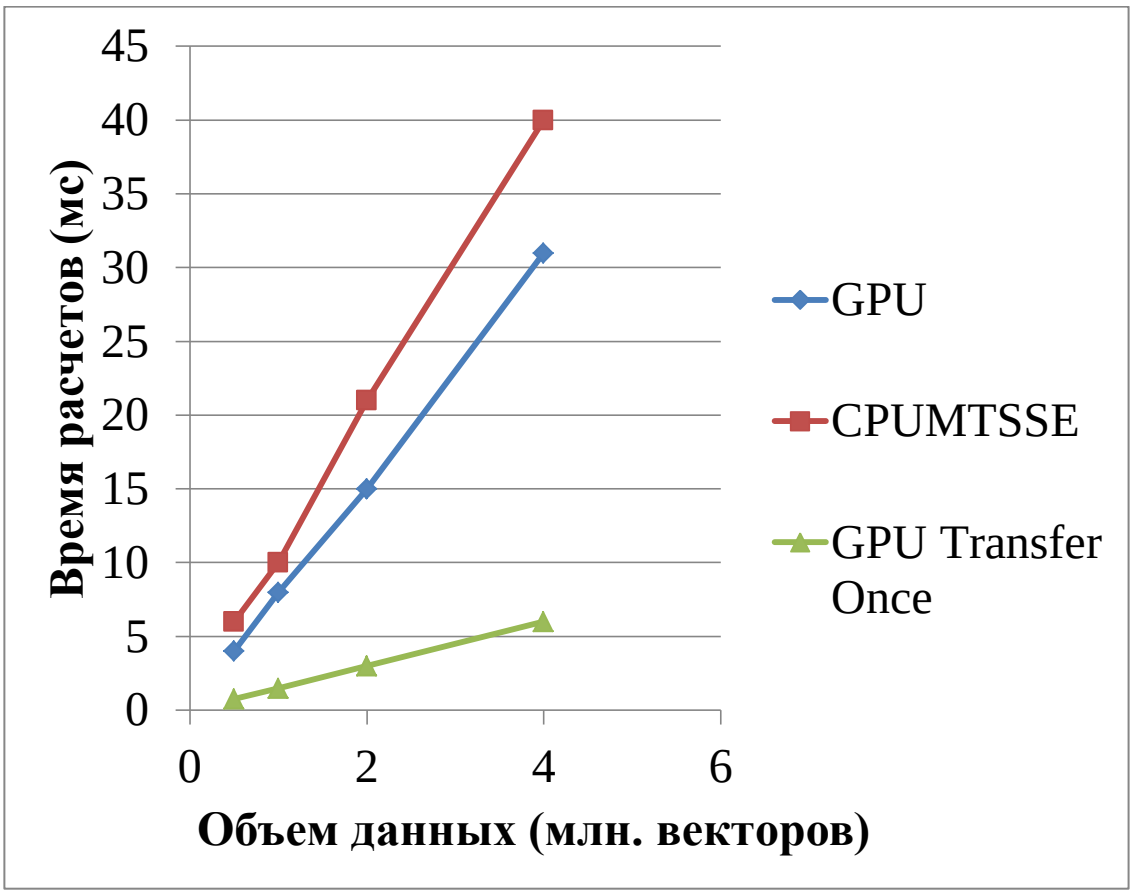

Рис. 16. Сравнение скорости выполнения вычислений на *GPU* для разных типов задач

В качестве примера было выполнено 8 вычислительных итераций для тестовой задачи, при этом операция записи в текстуру была применена только перед первой итерацией, а операция чтения из текстуры – только после последней. В результате графический процессор отработал в пять раз быстрее по сравнению с версией с постоянным выводом результатов и примерно в семь раз быстрее – по сравнению с многоядерным центральным процессором.

4.7. Балансировка нагрузки на *CPU* и *GPU*

Их графиков видно, что зависимость времени выполнения вычислений от объема данных имеет вид, близкий к линейному, и может быть описана формулами:

$$t_{cpu} \approx k_{cpu} \cdot x_{cpu}$$

$$t_{gpu} \approx k_{gpu} \cdot x_{gpu}$$

Здесь k обозначает линейный коэффициент, характерный для данной задачи и данного типа процессора, а x – объем данных, поставляемых на данный вычислитель.

Зафиксируем объем передаваемых данных x и введем коэффициент $\alpha \in [0; 1]$, который будет обозначать процент данных, отправляемых на *CPU*. Подберем коэффициент так, чтобы время выполнения вычислений на *CPU* и *GPU* было минимальным:

$$\mathrm{t} = \max(k_{cpu} \cdot \alpha \cdot x, k_{gpu} \cdot (1 - \alpha) \cdot x) \to \min$$

Покажем, что данное условие будет выполнено при равенстве времени выполнения вычислений на *CPU* и на *GPU*. Пусть при некотором коэффициенте α_1 выполняется условие:

$$t_1 = k_{cpu} \cdot \alpha_1 \cdot x = k_{gpu} \cdot (1 - \alpha_1) \cdot x = \max(t_{cpu_1}, t_{gpu_1})$$

Возьмем коэффициент $\alpha_2 > \alpha_1$. Тогда:

$$t_{cpu_2} = k_{cpu} \cdot \alpha_2 \cdot x > t_1 \Rightarrow t_2 > t_1$$

Возьмем коэффициент $\alpha_3 < \alpha_1$. Тогда:

$$t_{gpu_3} = k_{gpu} \cdot (1 - \alpha_3) \cdot x > t_1 \Rightarrow t_3 > t_1$$

Утверждение доказано. Найдем необходимый коэффициент α:

$$k_{cpu} \cdot \alpha \cdot x = k_{gpu} \cdot (1 - \alpha) \cdot x$$

$$\alpha = \frac{k_{gpu}}{k_{cpu} + k_{gpu}}$$

Таким образом, для нахождения нужного коэффициента α необходимо воспользоваться следующим алгоритмом.

1. Зафиксируем объем данных x, которые нам необходимо обработать. Определим объем данных для первой итерации как $x^{(1)} = x$. Зафиксируем номер текущей итерации $N = 1$ и количество итераций $N_{max} = const$. Достаточно провести три-четыре итерации.
2. Проведем вычисления на многоядерном *CPU* для объема данных $x^{(N)}$ и зафиксируем время их проведения как $t_{cpu}^{(N)}$.

3. Проведем вычисления на *GPU* для объема данных $x^{(N)}$ и зафиксируем время их проведения как $t_{gpu}^{(N)}$.
4. Если $N < N_{max}$, начнем новую итерацию: $x^{(N+1)} \leftarrow \frac{x^{(N)}}{2}$; $N \leftarrow N + 1$ – и перейдем к пункту 2.
5. Найдем усредненные линейные коэффициенты: $k_{cpu} = \frac{\sum_1^N 2^{(N-1)} \cdot t_{cpu}^{(N)}}{N \cdot x}$ и $k_{gpu} = \frac{\sum_1^N 2^{(N-1)} \cdot t_{gpu}^{(N)}}{N \cdot x}$.
6. Найдем $\alpha = \frac{k_{gpu}}{k_{cpu} + k_{gpu}}$.

Для тестовой задачи и массива с 4 миллионами четырехкомпонентных векторов для *CPU Intel Core i7-920* и *GPU AMD Radeon 5970* были найдены следующие коэффициенты:

$$k_{cpu} \approx 10; \quad k_{gpu} \approx 7; \quad \alpha \approx 0.4$$

Был произведен замер времени выполнения вычислений в зависимости коэффициента α. Как можно увидеть на рис. 17, именно при коэффициенте 0.4 достигается минимальное время работы.

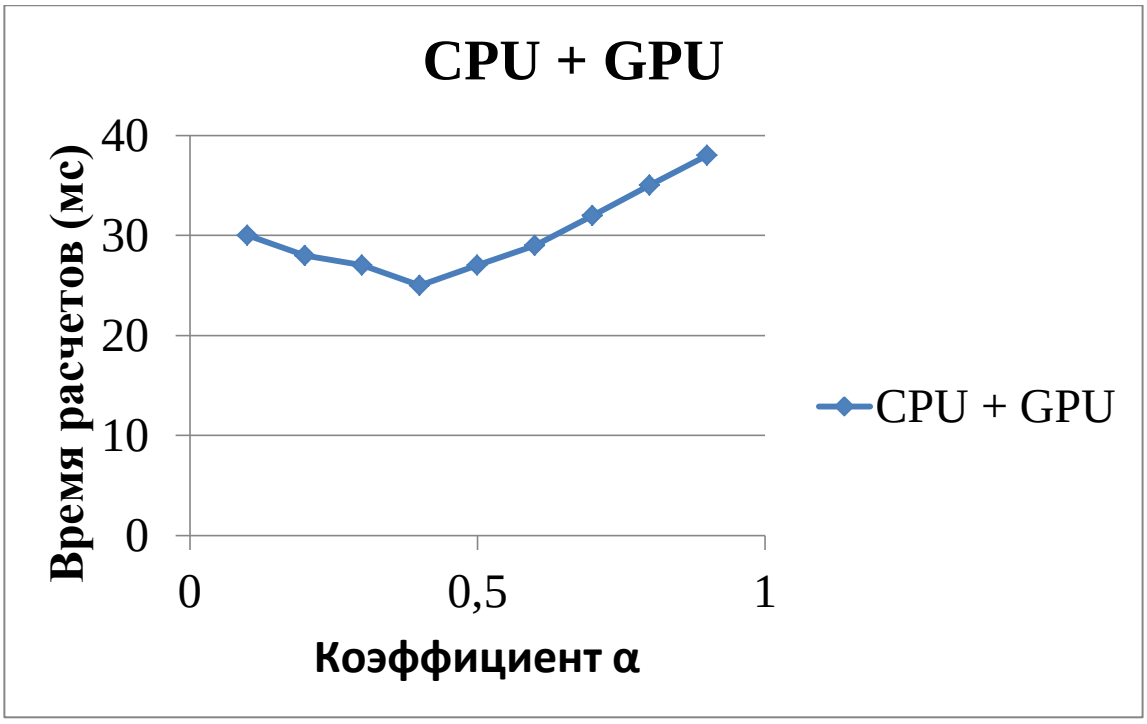

Рис. 17. Зависимость времени выполнения вычислений от коэффициента балансировки нагрузки

На рис. 18 приводятся графики с рис. 15 в сравнении с результатом, полученным при оптимальной балансировке нагрузки.

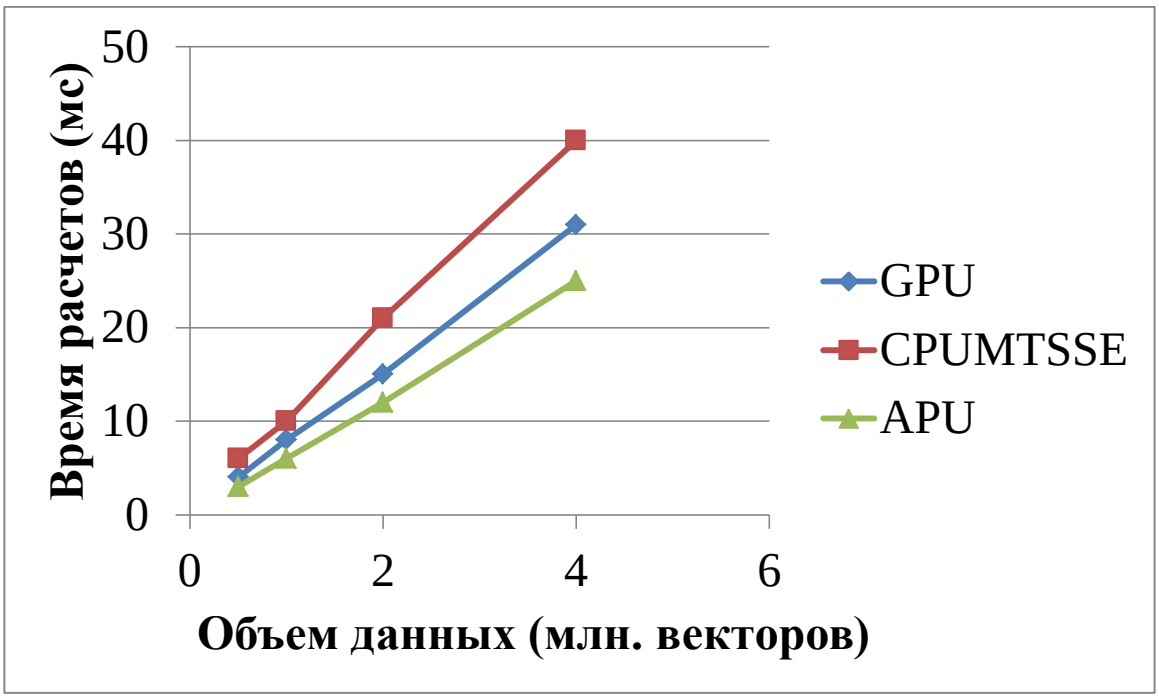

Рис. 18. Сравнение производительности *CPU*, *GPU* и их совместного использования

Как можно увидеть на графике, прирост от совместного использования *CPU* и *GPU* для данной задачи составляет около 20% по сравнению с *GPU* и около 40% по сравнению с *CPU*.

4.8. Достоинства и недостатки изложенного метода совместного использования связки из *CPU* и *GPU*

Отметим основные достоинства метода:

- отсутствие необходимости написания отдельного кода вычислений для центрального процессора избавляет от дополнительных трудозатрат и возможных ошибок;
- использование векторных инструкций *SSE* позволяет значительно ускорить работу кода;

- статическая компиляция сгенерированного кода позволяет компилятору провести локальные и глобальные оптимизации для ускорения вычислений;
- изложенный метод нахождения коэффициента балансировки прост и может быть использован адаптивно в процессе выполнения приложения.

Отметим основные недостатки метода:

- статическая компиляция кода требует перекомпиляции проекта в случае любых изменений, тогда как шейдеры компилируются динамически во время их инициализации в программе;
- необходимо вручную поддерживать многопоточность и распределение данных между вычислителями;
- метод балансировки может быть применен не для всех задач: взаимодействие между распределенными участками массива данных невозможно при обработке одного участка центральным процессором, а другого – графическим.

Заключение

1. Рассмотрены средства реализации *GPGPU*-вычислений и выявлены особенности таких вычислений, а также недостатки традиционного подхода.
2. Рассмотрен доработанный алгоритм оптимизации программ, состоящих из нескольких шейдеров, и внедрен в программное средство *GPGPU-Optimizer*.
3. Упрощено взаимодействие с графическим процессором за счет использования авторской *GPGPU*-библиотеки.
4. Выявлена возможность параллельного использования центрального и графического процессоров, разработан адаптивный алгоритм балансировки нагрузки на них.

Список источников

1. *Dean J., Ghemawat S.* MapReduce: Simplied Data Processing on Large Clusters. http://labs.google.com/papers/mapreduce-osdi04.pdf
2. Интернет-ресурс http://www.gpureview.com/show_cards.php
3. Интернет-ресурс http://www.gpgpu.org
4. *Селифонов Е. В.* Оптимизация работы программ для графического процессора. http://is.ifmo.ru/papers/_selifonov.pdf
5. Интернет-ресурс http://fusion.amd.com
6. Интернет-ресурс http://www.intel.com/technology/architecture-silicon/2ndgen/index.htm
7. Интернет-ресурс http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/
8. Интернет-ресурс http://www.opengl.org
9. Интернет-ресурс http://msdn.microsoft.com/ru-ru/directx/
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://en.wikipedia.org/wiki/High_Level_Shader_Language
15. Интернет-ресурс http://msdn.microsoft.com/en-us/library/ff471356(v=vs.85).aspx
16. Интернет-ресурс http://www.nvidia.com/cuda/
17. Интернет-ресурс http://www.khronos.org/opencl/

18. Интернет-ресурс http://www.opengl.org/resources/libraries/glut/
19. Интернет-ресурс http://glew.sourceforge.net