Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

09.03.01 Информатика и вычислительная техника

Профиль: Программное обеспечение средств вычислительной техники и автоматизированных систем

**Отчёт по расчётно-графическому заданию**

**по дисциплине «Программирование графических процессоров»**

**по теме**

**«Сравнение вычислений общего назначения**

**на основе OpenGL - вычислительных шейдеров и интерфейса CUDA»**

по направлению 09.03.01 «Информатика и вычислительная техника»,

направленность (профиль) – «Программное обеспечение средств вычислительной техники и автоматизированных систем», квалификация – бакалавр,

программа академического бакалавриата,

форма обучения – очная, год начала подготовки (по учебному плану) – 2016

Выполнил: студент 3 курса гр. ИП-611 / Макаревич А.А. /

Проверил: профессор кафедры ПМиК / Малков Е.А. /

Новосибирск, 2019

Содержание

[Введение 3](#_Toc9327794)

[1. Постановка задачи 4](#_Toc9327795)

[2. Анализ предметной области 5](#_Toc9327796)

[2.1. Цель создания 5](#_Toc9327797)

[2.2. Перечень решаемых задач 5](#_Toc9327798)

[2.3. Специальное и общесистемное программное обеспечение 5](#_Toc9327799)

[3. Программное проектирование 7](#_Toc9327800)

[3.1. Пользовательский интерфейс 7](#_Toc9327801)

[3.2. Функциональное обеспечение 7](#_Toc9327802)

[3.3. Алгоритмическое обеспечение 8](#_Toc9327803)

[3.4. Архитектурное обеспечение 8](#_Toc9327804)

[3.5. Информационное обеспечение 9](#_Toc9327805)

[4. Программная реализация 10](#_Toc9327806)

[4.1. Описание программы-клиента 10](#_Toc9327807)

[4.2. Результаты выполнения 16](#_Toc9327808)

[5. Сопроводительная документация программного обеспечения 17](#_Toc9327809)

[5.1. Разработка описания программы 17](#_Toc9327810)

[5.2. Разработка руководства пользователя 17](#_Toc9327811)

[5.3. Анализ программного обеспечения 18](#_Toc9327812)

[6. Заключение 18](#_Toc9327813)

[7. Список использованных источников 19](#_Toc9327814)

# Введение

Как известно, современная видеокарта — это устройство, которое позволяет быстро обрабатывать огромное количество данных за счёт параллельных вычислений. Расчётами в ней занимаются сотни (а в новых моделях и тысячи) процессоров. То, что они работают одновременно, позволяет получить огромное быстродействие. Разумеется, эти процессоры не настолько мощные и универсальные, как центральный процессор компьютера (CPU). Но для обработки изображений часто требуется лишь набор только самых базовых команд и операций из линейной алгебры. Так что, видеокарты берут вопреки завету Суворова не умением, а числом. Уже давно существуют такие библиотеки, как CUDA от Nvidia или универсальная OpenCL.

На данный момент реализация OpenGL включает в себя несколько библиотек (описание базовых функций OpenGL, GLU, GLUT, GLFW, GLEW и другие).

Характерными особенностями OpenGL, которые обеспечили распространение и развитие этого графического стандарта, являются:

* ***Стабильность***. Дополнения и изменения в стандарте реализуются таким образом, чтобы сохранить совместимость с разработанным ранее программным обеспечением.
* ***Надежность и переносимость***. Приложения, использующие OpenGL, гарантируют одинаковый визуальный результат вне зависимости от типа используемой операционной системы и организации отображения информации. Кроме того, эти приложения могут выполняться как на персональных компьютерах, так и на рабочих станциях и суперкомпьютерах.
* ***Легкость применения***. Стандарт OpenGL имеет продуманную структуру и интуитивно понятный интерфейс, что позволяет с меньшими затратами создавать эффективные приложения, содержащие меньше строк кода, чем с использованием других графических библиотек. Необходимые функции для обеспечения совместимости с различным оборудованием реализованы на уровне библиотеки и значительно упрощают разработку приложений.

Все данные, которые мы будем обрабатывать, мы будем хранить в виде текстур. Ведь что такое текстура? Это изображение, то есть двумерный массив пикселей. По сути это матрица, а с ними нам и нужно работать.

Программу для поэлементных вычислений над матрицами в GPGPU называют ядром. Ядра применяются к элементам матрицы независимо и параллельно. То есть, в идеале время обработки всей матрицы будет равно времени работы одного ядра. Ядра — это фрагментные шейдеры. Мы можем запускать шейдеры с некоторыми параметрами, делая их более универсальными, но надо помнить, что параметры будут одинаковыми для всех шейдеров на время обработки.

Процесс вычислений — это процесс отрисовки фрагментов. То есть, чтобы запустить шейдеры, нам сначала нужно нарисовать что-то. Обычно это прямоугольник, на который наложена наша текстура. Но так как основной примитив у нас треугольник, то вместо прямоугольника рисуем два прямоугольных треугольника.

# 1. Постановка задачи

В качестве расчётно-графического задания было выбрана тема – сравнение вычислений общего назначения на основе OpenGL - вычислительных шейдеров и интерфейса CUDA, включающая в себя следующие пункты, которые нам следовало реализовать в нашем приложении:

* Настроить среду для разработки OpenGL в Visual Studio 2017 (установить и настроить нужные библиотеки);
* Реализовать SAXPY с помощью OpenGL – вычислительных щейдеров;
* Реализовать SAXPY на основе библиотек thrust и cuBLAS;
* Сравнить время выполнения saxpy при разных входных данных;
* Сравнить производительности программ на основе интерфейса CUDA и на основе OpenGL - вычислительных шейдеров.

# 2. Анализ предметной области

Предметная область проекта – реализация saxpy, технологии CUDA и OpenGL. SAXPY – векторный алгоритм, смысл которого заключается в вычислении скалярного произведения константы на один вектор и прибавление другого вектора, и который очень часто используется в линейной алгебре.

Требуется разработать проект, который включает в себя разработку следующих функциональных частей: разработка реализации saxpy на основе OpenGL и разработка saxpy на основе библиотек cuBLAS и thrust.

## 2.1. Цель создания

Главная цель расчётно-графического задания по теме «Сравнение вычислений общего назначения на основе OpenGL - вычислительных шейдеров и интерфейса CUDA» – реализация алгоритма линейной алгебры (saxpy) с помощью технологии OpenGL.

## 2.2. Перечень решаемых задач

Функционал данной программы предусматривает выполнение следующих задач:

* Реализация алгоритма saxpy с помощью технологии OpenGL;
* Реализация алгоритма saxpy с помощью библиотек cuBLAS и thrust;
* Сравнение времени выполнения программы при разных входных данных;
* Сравнение времени выполнения saxpy при реализации данного алгоритма на OpenGL, cuBLAS или thrust.
* Вывод результатов сравнений в консоль.

## 2.3. Специальное и общесистемное программное обеспечение

При написании данного расчётно-графического задания планируется использовать следующее стороннее программное обеспечение и инструментарий:

**Microsoft Visual Studio 2017** — это набор инструментов для создания программного обеспечения: от планирования до разработки пользовательского интерфейса, написания кода, тестирования, отладки, анализа качества кода и производительности, развертывания в средах клиентов и сбора данных телеметрии по использованию. Эти инструменты предназначены для максимально эффективной совместной работы; все они доступны в интегрированной среде разработки (IDE) Visual Studio.  
 **OpenGL (Open Graphics Library)** — спецификация, определяющая платформонезависимый (независимый от языка программирования) программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику. Включает более 300 функций для рисования сложных трёхмерных сцен из простых примитивов. Используется при создании компьютерных игр, САПР, виртуальной реальности, визуализации в научных исследованиях. На платформе Windows конкурирует с Direct3D. Производители оборудования на основе этой спецификации создают реализации — библиотеки функций, соответствующих набору функций спецификации. Реализация призвана эффективно использовать возможности оборудования.

**CUDA (Compute Unified Device Architecture)** — программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia. CUDA SDK позволяет программистам реализовывать на специальных упрощённых диалектах языков программирования Си, C++ и Фортран алгоритмы, выполнимые на графических и тензорных процессорах Nvidia. Архитектура CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического или тензорного ускорителя и управлять его памятью.

**CUBLAS** — реализация интерфейса программирования приложений для создания библиотек, выполняющих основные операции линейной алгебры BLAS (Basic Linear Algebra Subprograms) для CUDA. Он позволяет получить доступ к вычислительным ресурсам графических процессоров NVIDIA. Библиотека является самодостаточной на уровне API, то есть, прямого взаимодействия с драйвером CUDA не происходит. CUBLAS прикрепляется к одному GPU и автоматически не распараллеливается между несколькими GPU. Основные функции библиотеки CUBLAS: создание матриц и векторных объектов в пространстве памяти GPU, заполнение их данными, вызов последовательных функций CUBLAS, и загрузка результатов из области памяти GPU обратно к хосту. Чтобы достичь этого, CUBLAS предоставляет вспомогательные функции для создания и уничтожения объектов в памяти GPU, и для записи данных и извлечения информации из этих объектов.

**Thrust** является библиотекой шаблонов C++ для CUDA на основе стандартной библиотеки шаблонов (STL). Thrust позволяет реализовать высокий производительность параллельные приложения с минимальными усилиями программирования через интерфейс высокого уровня, который является полностью совместимым с CUDA C.

Разработка будет осуществляться в операционной системе Windows 10.

# 3. Программное проектирование

## 3.1. Пользовательский интерфейс

При загрузке приложений, в первую очередь пользователь видит интерфейс продукта, поэтому особое внимание требуется уделить на разработку интерфейса. В данной работе интерфейс максимально прост, так как суть проекта состоит в реализации алгоритма, а не в его демонстрации.

Программа представляет собой консольное приложения, в котором выводится время выполнения алгоритма saxpy при разных реализациях (по необходимости можно вывести результат работы saxpy).

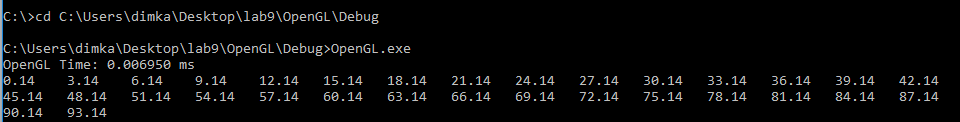


Рисунок 1. Макет окна вывода OpenGL.

Также на домашнем компьютере, где установлена видеокарта Nvidia, были протестированы библиотеки CUDA.

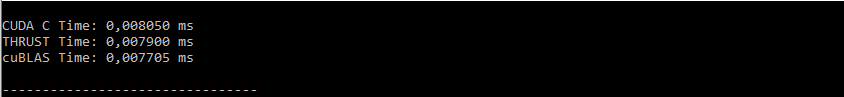


Рисунок 2. Макет окна вывода CUDA.

## 3.2. Функциональное обеспечение

Прежде, чем начинать разработку необходимо заранее определить требуемый функционал нашего программного обеспечения.

Пользователь должен иметь возможность выполнить следующие действия:

* Применить реализованный алгоритм saxpy с помощью технологии OpenGL;
* Применить реализованный алгоритм saxpy с помощью библиотек cuBLAS и thrust;
* Сравнить время выполнения программы при разных входных данных;
* Сравнить время выполнения saxpy при реализации данного алгоритма на OpenGL, cuBLAS или thrust.
* Просмотреть результаты сравнений в консоли.

## 3.3. Алгоритмическое обеспечение

В данной работе реализован векторный алгоритм линейной алгебры saxpy, смысл которого заключается в вычислении скалярного произведения константы на один вектор и прибавление другого вектора, и который очень часто используется в линейной алгебре.

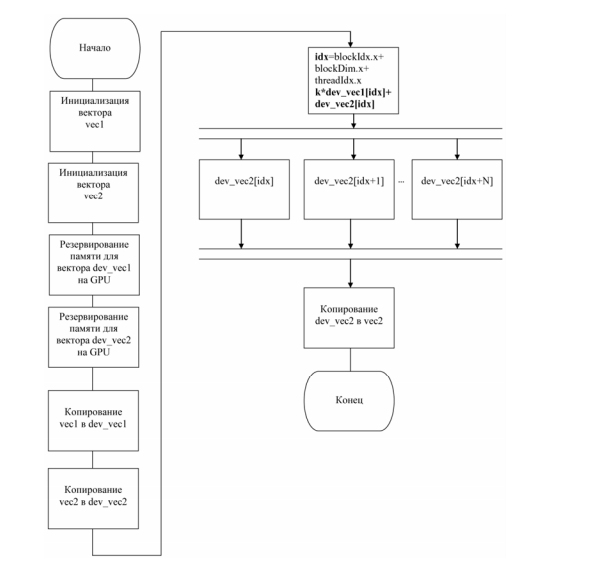


Рисунок 3. Блок-схема векторной реализации алгоритма SAXPY.

## 3.4. Архитектурное обеспечение

В ходе выполнения данного задания использовалась видеокарта производства Nvidia ***GIGABYTE GeForce GT 730*** со следующими основными характеристиками:

* Объем видеопамяти - 2 ГБ;
* Тип памяти - GDDR3;
* Разрядность шины памяти - 64 бит;
* Графический процессор - GeForce GT 730;
* Количество видеочипов - 1;
* Штатная частота работы видеочипа - 902 МГц;
* Число текстурных блоков - 16;
* Версия шейдеров - 5.0;
* Число блоков растеризации - 8.

OpenGL ориентируется на следующие две задачи:

* Скрыть сложности адаптации различных 3D-ускорителей, предоставляя разработчику единый API.
* Скрыть различия в возможностях аппаратных платформ, требуя реализации недостающей функциональности с помощью программной эмуляции.

Основным принципом работы OpenGL является получение наборов векторных графических примитивов в виде точек, линий и треугольников с последующей математической обработкой полученных данных и построением растровой картинки на экране и/или в памяти. Векторные трансформации и растеризация выполняются графическим конвейером (graphics pipeline), который по сути представляет собой дискретный автомат. Абсолютное большинство команд OpenGL попадает в одну из двух групп: либо они добавляют графические примитивы на вход в конвейер, либо конфигурируют конвейер на различное исполнение трансформаций. OpenGL является низкоуровневым процедурным API, что вынуждает программиста диктовать точную последовательность шагов, чтобы построить результирующую растровую графику (императивный подход). Это является основным отличием от дескрипторных подходов, когда вся сцена передается в виде структуры данных (чаще всего дерева), которое обрабатывается и строится на экране. С одной стороны, императивный подход требует от программиста глубокого знания законов трёхмерной графики и математических моделей, с другой стороны — даёт свободу внедрения различных инноваций.

В архитектуре CUDA используется модель памяти грид, кластерное моделирование потоков и SIMD-инструкции. Применима не только для высокопроизводительных графических вычислений, но и для различных научных вычислений с использованием видеокарт nVidia. Учёные и исследователи широко используют CUDA в различных областях, включая астрофизику, вычислительную биологию и химию, моделирование динамики жидкостей, электромагнитных взаимодействий, компьютерную томографию, сейсмический анализ и многое другое. В CUDA имеется возможность подключения к приложениям, использующим OpenGL и Direct3D. CUDA — кроссплатформенное программное обеспечение для таких операционных систем, как Linux, Mac OS X и Windows.

## 3.5. Информационное обеспечение

Информация в программу поступает как массив данных, которыми в дальнейшем оперирует алгоритм saxpy.

# 4. Программная реализация

Так как работа над данным продуктом осуществлялась бригадой, то наши обязанности были равномерно распределены.

Моя ответственность легла на реализацию алгоритма saxpy с помощью технологии OpenGL. Было сравнено время работы программы при различных входных данных. Также результаты данной реализации были сравнены с результатами, полученными с помощью библиотек CUDA.

## 4.1. Описание программы-клиента

//Подключение библиотек Thrust и CuBLAS:

#include <thrust/host\_vector.h>

#include <thrust/device\_vector.h>

#include <thrust/transform.h>

#include <thrust/fill.h>

#include <thrust/sequence.h>

#include <stdio.h>

#include <cublas\_v2.h>

//Макрос для определения ошибки:

//cudaGetErrorString – возвращает сообщение с кодом ошибки

//\_\_LINE\_\_, \_\_FILE\_\_ - предопределённыые макросы препроцессора для определения местоположения в коде программы – номер строки и имени файла.

#define CUDA\_CHECK\_RETURN(value) {\

cudaError\_t \_m\_cudaStat = value;\

if(\_m\_cudaStat != cudaSuccess) {\

fprintf(stderr, "Error %s at line %d in file %s\n", cudaGetErrorString(\_m\_cudaStat), \_\_LINE\_\_, \_\_FILE\_\_);\

exit(1);\

}\

}

//Реализация saxpy

//Функтор Saxpy:

struct saxpy\_functor {

const float a;

saxpy\_functor(float \_a) : a(\_a) {}

\_\_host\_\_ \_\_device\_\_ float operator()(float x, float y) {

return a \* x + y;

}

};

//Алгоритм transform:

void saxpy(float a, thrust::device\_vector<float>& x, thrust::device\_vector<float>& y) {

saxpy\_functor func(a);

thrust::transform(x.begin(), x.end(), y.begin(), y.begin(), func);

}

\_\_global\_\_ void gSaxpy(float alpha, float \*x, float \*y) {

int i = threadIdx.x + blockIdx.x \* blockDim.x;

y[i] = alpha \* x[i] + y[i];

}

//Реализация «сырого» CUDA C кода:

float saxpyCUDAC() {

cudaEvent\_t start, stop;

float \*x\_d, \*x\_h, \*y\_h, \*y\_d;

float elapsedTime;

//Создание объекта событий start и stop:

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

//Выделяем память на устройствах:

//cudaMalloc(указатель на выделенную память устройства, запрошенный размер выделения в байтах)

CUDA\_CHECK\_RETURN(cudaMalloc((void\*\*)&x\_d, N \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void\*\*)&y\_d, N \* sizeof(float)));

x\_h = (float\*)calloc(N, sizeof(float));

y\_h = (float\*)calloc(N, sizeof(float));

for(int i = 0; i < N; i++) {

x\_h[i] = i;

y\_h[i] = 0.87;

}

//Копируем данные между хостом и устройством:

//cudaMemcpy(адрес памяти назначения, адрес источника памяти, размер в байтах для копирования, тип перевода)

CUDA\_CHECK\_RETURN(cudaMemcpy(x\_d, x\_h, N \* sizeof(float), cudaMemcpyHostToDevice));

CUDA\_CHECK\_RETURN(cudaMemcpy(y\_d, y\_h, N \* sizeof(float), cudaMemcpyHostToDevice));

//Записывает событие start:

cudaEventRecord(start, 0);

gSaxpy <<< N / 256, 256 >>> (3.0, x\_d, y\_d);

//Ожидание окончания работы вычислительного устройства, возвращает ошибку, если одна из предыдущих задач не выполнена:

CUDA\_CHECK\_RETURN(cudaDeviceSynchronize());

//Записывает событие stop:

cudaEventRecord(stop, 0);

//Ожидает завершения события stop:

cudaEventSynchronize(stop);

//Вычисляет прошедшее время между событиями:

cudaEventElapsedTime(&elapsedTime, start, stop);

//Копируем данные с устройства на хост:

CUDA\_CHECK\_RETURN(cudaMemcpy(y\_h, y\_d, N \* sizeof(float), cudaMemcpyDeviceToHost));

cudaEventDestroy(start);

cudaEventDestroy(stop);

//Освобождает память на устройстве:

cudaFree(x\_d);

cudaFree(y\_d);

free(x\_h);

free(y\_h);

return elapsedTime;

}

//Реализация алгоритма линейной алгебры с использование библиотеки CuBLAS:

float saxpyCuBLAS() {

cudaEvent\_t start, stop;

float \*x\_h, \*y\_h, \*x\_d, \*y\_d;

float elapsedTime;

//Создание объекта событий start и stop:

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

//Выделяем заблокированную страницу памяти на хосте:

//cudaMallocHost(указатель на выделенную память хоста, запрошенный размер выделения в байтах)

CUDA\_CHECK\_RETURN(cudaMallocHost((void\*\*)&x\_h, N \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMallocHost((void\*\*)&y\_h, N \* sizeof(float)));

//Выделяем память на устройствах:

//cudaMalloc(указатель на выделенную память устройства, запрошенный размер выделения в байтах)

CUDA\_CHECK\_RETURN(cudaMalloc((void\*\*)&x\_d, N \* sizeof(float)));

CUDA\_CHECK\_RETURN(cudaMalloc((void\*\*)&y\_d, N \* sizeof(float)));

for(int i = 0; i < N; i++) {

x\_h[i] = (float) i;

y\_h[i] = 0.87f;

}

//Инициализация библиотеки CUBLAS:

cublasHandle\_t cublas\_handle;

cublasCreate(&cublas\_handle);

const int num\_rows = N;

const int num\_cols = 1;

const size\_t elem\_size = sizeof(float);

//Копирование матрицы с числом строк num\_elem и одним столбцом с хоста на устройство:

cublasSetMatrix(num\_rows, num\_cols, elem\_size, x\_h, num\_rows, x\_d, num\_rows);

cublasSetMatrix(num\_rows, num\_cols, elem\_size, y\_h, num\_rows, y\_d, num\_rows);

const int stride = 1;

float alpha = 3.0f;

//Записывает событие start:

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

//Выполнение SingleAlphaXPlusY:

cublasSaxpy(cublas\_handle, N, &alpha, x\_d, stride, y\_d, stride);

//Записывает событие stop:

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

//Ожидает завершения события stop:

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

//Вычисляет прошедшее время между событиями:

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&elapsedTime, start, stop));

//Копирование матриц с числом строк num\_elem и одним столбцом с устройства на хост:

cublasGetMatrix(num\_rows, num\_cols, elem\_size, x\_d, num\_rows, x\_h, num\_rows);

cublasGetMatrix(num\_rows, num\_cols, elem\_size, y\_d, num\_rows, y\_h, num\_rows);

cudaEventDestroy(start);

cudaEventDestroy(stop);

//Освобождаем ресурсы на устройстве:

cublasDestroy(cublas\_handle);

CUDA\_CHECK\_RETURN(cudaFree(x\_d));

CUDA\_CHECK\_RETURN(cudaFree(y\_d));

//Освобождаем ресурсы на хосте:

CUDA\_CHECK\_RETURN(cudaFreeHost(x\_h));

CUDA\_CHECK\_RETURN(cudaFreeHost(y\_h));

return elapsedTime;

}

//Реализация алгоритма линейной алгебры с использование библиотеки Thrust:

float saxpyTHRUST() {

cudaEvent\_t start, stop;

float elapsedTime;

//Создание объекта событий start и stop:

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

//Контейнеры host-vector:

//Инициализируем host\_vector N числом:

thrust::host\_vector<float> h1(N);

thrust::host\_vector<float> h2(N);

//Устанавливаем все элементы h1 в 0, 1, 2, 3:

thrust::sequence(h1.begin(), h1.end());

//Устанавливаем элементы вектора в 0.87:

thrust::fill(h2.begin(), h2.end(), 0.87);

//Контейнеры device\_vector:

thrust::device\_vector<float> d1 = h1;

thrust::device\_vector<float> d2 = h2;

//Записывает события start и stop:

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

saxpy(3.0, d1, d2);

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

//Ожидает завершения события stop:

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

//Вычисляет прошедшее время между событиями:

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&elapsedTime, start, stop));

h2 = d2;

h1 = d1;

cudaEventDestroy(start);

cudaEventDestroy(stop);

return elapsedTime;

}

int main() {

float elapsedTime;

//Вычисление времени выполнения «сырого» CUDA C кода при реализации алгоритма линейной алгебры:

elapsedTime = saxpyCUDAC();

printf("CUDA C Time: %f ms\n", elapsedTime);

//Вычисление времени выполнения при реализации алгоритма линейной алгебры c с использованием библиотеки Thrust:

elapsedTime = saxpyTHRUST();

printf("THRUST Time: %f ms\n", elapsedTime);

//Вычисление времени выполнения при реализации алгоритма линейной алгебры c с использованием библиотеки CuBLAS:

elapsedTime = saxpyCuBLAS();

printf("cuBLAS Time: %f ms\n", elapsedTime);

return 0;

}

## 4.2. Результаты выполнения

Сравнение времени выполнения saxpy при реализации алгоритма на OpenGL, cuBLAS и thrust.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Название метода** | **Время на обработку вектора, размером , мс** | **Время на обработку вектора, размером , мс** | **Время на обработку вектора, размером , мс** | **Время на обработку вектора, размером , мс** | **Время на обработку вектора, размером , мс** |
| CUDA C | 0,035808 | 0,038000 | 0,051800 | 0,492000 | 2,000560 |
| Thrust | 0,003120 | 0,003780 | 0,007905 | 0,209004 | 1,730567 |
| cuBLAS | 0,003016 | 0,003700 | 0,007790 | 0,206735 | 1,715980 |
| OpenGL | 0,006300 | 0,006900 | 0,016950 | 0,535500 | 4,841800 |

Сравнение времени выполнения программы при разных входных данных на основе реализации по средством OpenGL:

|  |  |
| --- | --- |
| **Размер вектора** | **Время выполнения, мс** |
|  | 0,000150 |
|  | 0,000200 |
|  | 0,000200 |
|  | 0,000150 |
|  | 0,000200 |
|  | 0,000150 |
|  | 0,000150 |
|  | 0,000200 |
|  | 0,006050 |
|  | 0,006100 |
|  | 0,006200 |
|  | 0,006300 |
|  | 0,006950 |
|  | 0,008250 |
|  | 0,010600 |
|  | 0,016950 |
|  | 0,031300 |
|  | 0,065850 |
|  | 0,134200 |
|  | 0,272050 |
|  | 0,535500 |
|  | 1,153050 |
|  | 2,211150 |
|  | 4,841800 |

# 5. Сопроводительная документация программного обеспечения

## 5.1. Разработка описания программы

Общее описание:

«Данный программный продукт предназначен для реализации алгоритма saxpy по средствам OpenGL и CUDA и для сравнения и оценки производительности этих двух технологий. CUDA реализация требует предварительной установки ПО для разработки программ с кодом для GPU. OpenGL реализация требует предварительной настройки среды для разработки OpenGL приложений».

## 5.2. Разработка руководства пользователя

1. Для настройки OpenGL в VS 2017 необходимо скачать и настроить библиотеки GLEW и GLFW, для наглядности весь процесс детально описан по ссылке *https://www.youtube.com/watch?v=gCkcP0GcCe0*

2. Для запуска программы с CUDA необходимо скомпилировать программу через командную строку командой *«nvcc saxpy.cu –o saxpy»,* а затем командой *«./saxpy.exe»* открыть скомпилированный файл.

3. Для запуска программы с OpenGL необходимо произвести отладку проекта через VS, а затем открыть файл OpenGL.exe.

Для запуска приложения необходимым и достаточным условием являются следующие требования:

* Операционная система: Windows 7 или выше;
* Объём оперативной памяти: 1,00 ГБ;
* Свободное место на диске: 1,00 МБ;
* Наличие видеокарты (для CUDA – Nvidia, для OpenGL - любой).

## 5.3. Анализ программного обеспечения

Анализ исходного кода проводился с помощью встроенной утилиты в среду разработки Microsoft Visual Studio 2017.

Результат анализа исходного кода:

* Предупреждения о переменных, содержащих излишнюю инициализацию – исправлено;
* Предупреждение о неиспользуемых переменных – исправлено;
* Критических ошибок в коде не обнаружено.

# 6. Заключение

На основе полученных знаний по курсу «Программирование графических процессоров» нам в результате удалось выполнить расчётно-графическое задание по теме «Сравнение вычислений общего назначения на основе OpenGL - вычислительных шейдеров и интерфейса CUDA».

В ходе данного расчетно-графического задания была проверена оптимальность использования и скорость работы различных методов вычислений на графическом устройстве. Проанализировав зависимости времени от количества элементов можно сделать вывод, что на малом их количестве нет смысла использовать мощности графического устройства, так как перемещение многих элементов в память и остальные предварительные операции требуют времени и ресурсов.

В заключении хотелось бы отметить, что нам удалось выполнить цель, ставившуюся перед началом создания программного продукта.

# 7. Список использованных источников

1. Курс лекций Малкова Е.А. «Программирование графических процессоров».

2. Документация OpenGL API. http://docs.gl/

3. OpenGL. <https://ru.wikipedia.org/wiki/OpenGL>

4. CUDA. https://ru.wikipedia.org/wiki/CUDA

5. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учебное пособие. А. В. Боресков и др. Предисл.: В. А. Садовничий. Издательство Московского университета, 2012