Кафедра прикладной математики и кибернетики

Выполнили:

Студенты 3 курса группы ИП-111  
Корнилов А.А.,  
Попов М.И.,

Толкач А.А.

Проверил:

Профессор кафедры ПМиК  
Малков Е.А.

Расчетно-графическое задание

По дисциплине: «Программирование графических процессоров»

Сибирский государственный университет  
телекоммуникаций и информатики

Новосибирск, 2024

Министерство цифрового развития, связи  
и массовых коммуникаций Российской Федерации

**Задание:**

1. Сравнительный анализ производительности программ, реализующих алгоритмы линейной алгебры с использованием библиотек Thrust, cuBLAS и «сырого» CUDA C кода (оценка “хорошо”).
2. Сравнение производительности программ, выполняемых на нескольких GPU, с использованием CUDA Streams или потоков CPU (оценка “отлично”).
3. Сравнение производительности программ на основе интерфейса CUDA (пункт 1) и на основе вычислительных шейдеров OpenGL (оценка “отлично”).

**Цель:** разработать программу выполняющую алгоритм линейной алгебры умножения матриц с использованием интерфейса OpenGL

**Оборудование**: Видеокарта GTX 1050TI (Pascal)

**Выполнение работы:**

В группе было проведено распределение работы по написанию программы:

* Попов Мирон – настройка работы кроссплатформенной сборки CMake под Win32 и Linux, подключение библиотек для OpenGL в порядок сборки, частичная работа над кодом, отладка работы программы;
* Корнилов Андрей – Работа над основным кодом программы, выполнения отчета;
* Толкач Илья – Написание кода для шейдера и функции для файлов OpenGL.cpp и Matrix.cpp.

Для выполнения РГЗ была написана программа, используя OpenGL API и библиотеки расширяющие ей возможности, а именно GLFW (Graphics Library Framework) для создания окна в графическом режиме и библиотека GLEW (OpenGL Extension Wrangler Library) для использование вычисления с использованием вычислительных шейдеров. Также были использованы STD библиотеки:

* Iostream – для ввода вывода
* Random – для генерации начальных значений для матриц
* Fstream – для чтения вычислительного шейдера OpenGL из файла
* Chrono – для замера времени

|  |
| --- |
| #ifndef **RGZ\_H** #define **RGZ\_H** #include <GL/glew.h> #include <GLFW/glfw3.h> #include <iostream> #include <random> #include <fstream> #include <chrono>  const int windowHeight = 640; const int windowWidth = 480; const int matrixSize = 1 << 11;  inline GLchar infoLog[512];  // Matrix.cpp float\* generateRandomMatrix(int width, int height); void printMatrix(float\* matrix, int width, int height);  //OpenGL.cpp Glchar \*loadShader();  #endif //RGZ\_H |

Листинг 1 – Заголовочный файл RGZ.H

|  |
| --- |
| #include “RGZ.h”  Glchar \*loadShader() {  // Указываем файл и выбираем режим чтение  std::ifstream file(“Multiplication.comp.glsl”, std::ifstream::ate);  const int len = file.tellg();   auto \*computeSource = new Glchar[len + 1];   file.seekg(0, file.beg);  for (int I = 0; I < len; i++) file.get(computeSource[i]);   computeSource[len] = ‘\0’;  file.close();  return computeSource; } |

Листинг 2 – файл OpenGL.cpp

Функция для загрузки шейдера из файла, для OpenGL требуется чтобы содержимое char\* для загрузки был «сырым», без содержания управляющих символов.

|  |
| --- |
| #include “RGZ.h”  float\* generateRandomMatrix(const int width,const int height) {  std::random\_device rd;  // Выбор генератора для рандома, в данном случае Вихрь Мерсенна  std::mt19937 gen(rd());  // Генерация чисел в диапазоне 0.0, 1.0  std::uniform\_real\_distribution<float> dis(0.0f, 1.0f);   auto\* matrix = new float[width \* height];  for (int I = 0; I < height; ++i) {  for (int j = 0; j < width; ++j) {  matrix[I \* width + j] = dis(gen);  }  }  return matrix; } void printMatrix(float\* matrix,const int width,const int height) {  for (int I = 0; I < height; ++i) {  for (int j = 0; j < width; ++j) {  std::cout << matrix[I \* width + j] << “ “;  }  std::cout << std::endl;  } } |

Листинг 3 – файл Matrix.cpp

Функции для генерации значений для матриц, установка генератора рандома из chrono и непосредственно заполнение матриц. Функция для вывода на экран

|  |
| --- |
| #version 430 core layout(local\_size\_x = 1, local\_size\_y = 1) in; // Указываем размер локальной рабочей группы  layout(std430, binding = 0) buffer matrixA\_buffer{  float matrixA[]; }; // Буфер для матрицы A  layout(std430, binding = 1) buffer atrix\_buffer{  float atrix[]; }; // Буфер для матрицы B  layout(std430, binding = 2) buffer matrixResult\_buffer{  float matrixResult[]; }; // Буфер для результирующей матрицы  uniform int matrixSize; // Размер матриц  void main(){  // Получаем индексы элемента матрицы  uint i = gl\_GlobalInvocationID.x;  uint j = gl\_GlobalInvocationID.y;   // Вычисляем произведение матриц  float sum = 0.0;  for (int k = 0; k < matrixSize; ++k) {  sum += matrixA[i \* matrixSize + k] \* atrix[k \* matrixSize + j];  }  matrixResult[i \* matrixSize + j] = sum; } |

Листинг 4 – файл вычислительного шейдера Multiplication.comp.glsl

Шейдерный файл для выполнения в программе, выполняет разметку рабочей группы, получает выделенные буферы, в функции main размечает границы элементов матрицы и умножает элементы.

|  |
| --- |
| #include “RGZ.h”  typedef std::chrono::minutes min; typedef std::chrono::seconds sec;   int main() {  std::chrono::time\_point<std::chrono::system\_clock> start, end;   // Проверка на инициализацию GLFW (графическая часть OpenGL)  if (!glfwInit()) {  std::cerr << “Failed to initialize GLFW” << std::endl;  return -1;  }   // Устанавливаем параметры окна GLFW  glfwWindowHint(**GLFW\_CONTEXT\_VERSION\_MAJOR**, 4);  glfwWindowHint(**GLFW\_CONTEXT\_VERSION\_MINOR**, 3);  glfwWindowHint(**GLFW\_OPENGL\_PROFILE**, **GLFW\_OPENGL\_CORE\_PROFILE**);   // Создаем GLFW окно  GLFWwindow\* window = glfwCreateWindow(windowWidth, windowHeight, “Matrix atrixication”, nullptr, nullptr);  if (!window) {  std::cerr << “Failed to create GLFW window” << std::endl;  glfwTerminate();  return -1;  }   // Загружаем вычислительный шейдер из файла “Multiplication.comp.glsl”  char\* computeShaderSource = loadShader();   // Выбираем наше созданное окно  glfwMakeContextCurrent(window);   // Проверка на инициализацию GLEW (расширение для OpenGL)  glewExperimental = **GL\_TRUE**;  if (glewInit() != **GLEW\_OK**) {  std::cerr << “Failed to initialize GLEW” << std::endl;  glfwTerminate();  return -1;  }   // Создаем и загружаем вычислительный шейдер  Gluint computeShader = **glCreateShader**(**GL\_COMPUTE\_SHADER**);  **glShaderSource**(computeShader, 1, &computeShaderSource, nullptr);  **glCompileShader**(computeShader);   // Компилируем шейдер и проверяем его  Glint success;  **glGetShaderiv**(computeShader, **GL\_COMPILE\_STATUS**, &success);  if (!success) {  **glGetShaderInfoLog**(computeShader, 512, nullptr, infoLog);  std::cerr << «Compute shader compilation failed\n» << infoLog << std::endl;  glfwTerminate();  return -1;  }   // Создаем программу и приклепляем к ней шейдер  Gluint computeProgram = **glCreateProgram**();  **glAttachShader**(computeProgram, computeShader);  **glLinkProgram**(computeProgram);   **glGetProgramiv**(computeProgram, **GL\_LINK\_STATUS**, &success);  if (!success) {  **glGetProgramInfoLog**(computeProgram, 512, nullptr, infoLog);  std::cerr << «Compute program linking failed\n» << infoLog << std::endl;  glfwTerminate();  return -1;  }   // Создаем и генерируем матрицы  float\* matrixA = generateRandomMatrix(matrixSize, matrixSize);  float\* atrix = generateRandomMatrix(matrixSize, matrixSize);   // Инициализируем под каждую матрицу буфер  Gluint matrixBufferA, matrixBufferB, matrixBufferResult;  **glGenBuffers**(1, &matrixBufferA); // генерация буфера  **glBindBuffer**(**GL\_SHADER\_STORAGE\_BUFFER**, matrixBufferA); // привязка буфера, указываем что это шейдерный буфер  // Указываем размеры буфера, указываем что он статический  **glBufferData**(**GL\_SHADER\_STORAGE\_BUFFER**, matrixSize \* matrixSize \* sizeof(float), nullptr, **GL\_STATIC\_DRAW**);   **glGenBuffers**(1, &matrixBufferB);  **glBindBuffer**(**GL\_SHADER\_STORAGE\_BUFFER**, matrixBufferB);  **glBufferData**(**GL\_SHADER\_STORAGE\_BUFFER**, matrixSize \* matrixSize \* sizeof(float), nullptr, **GL\_STATIC\_DRAW**);   **glGenBuffers**(1, &matrixBufferResult);  **glBindBuffer**(**GL\_SHADER\_STORAGE\_BUFFER**, matrixBufferResult);  **glBufferData**(**GL\_SHADER\_STORAGE\_BUFFER**, matrixSize \* matrixSize \* sizeof(float), nullptr, **GL\_STATIC\_DRAW**);   // Копируем матрицы с хоста на девайс  **glBindBuffer**(**GL\_SHADER\_STORAGE\_BUFFER**, matrixBufferA);  **glBufferSubData**(**GL\_SHADER\_STORAGE\_BUFFER**, 0, matrixSize \* matrixSize \* sizeof(float), matrixA);  **glBindBuffer**(**GL\_SHADER\_STORAGE\_BUFFER**, matrixBufferB);  **glBufferSubData**(**GL\_SHADER\_STORAGE\_BUFFER**, 0, matrixSize \* matrixSize \* sizeof(float), atrix);    start = std::chrono::system\_clock::now();  // Выбираем программу  **glUseProgram**(computeProgram);  // Привязываем буферный шейдер  **glBindBufferBase**(**GL\_SHADER\_STORAGE\_BUFFER**, 0, matrixBufferA);  **glBindBufferBase**(**GL\_SHADER\_STORAGE\_BUFFER**, 1, matrixBufferB);  **glBindBufferBase**(**GL\_SHADER\_STORAGE\_BUFFER**, 2, matrixBufferResult);   //загружаем программу и подаем в неё обьект программы и указываем входной параметр  **glUniform1i**(**glGetUniformLocation**(computeProgram, “matrixSize”), matrixSize);  **glDispatchCompute**(matrixSize, matrixSize, 1);   // Ждем завершение работы программы  // для этого ставим барьер типо cudaDeviceSynchronize();  **glMemoryBarrier**(**GL\_SHADER\_STORAGE\_BARRIER\_BIT**);   // Копируем с девайса на хост результат  **glBindBuffer**(**GL\_SHADER\_STORAGE\_BUFFER**, matrixBufferResult);  auto\* matrixResult = (float\*)**glMapBuffer**(**GL\_SHADER\_STORAGE\_BUFFER**, **GL\_READ\_ONLY**);  end = std::chrono::system\_clock::now();   if (matrixResult) {  std::cout << “Result Matrix:” << std::endl;  //printMatrix(matrixResult, matrixSize, matrixSize);  printMatrix(matrixResult, 32, 32);  **glUnmapBuffer**(**GL\_SHADER\_STORAGE\_BUFFER**);  }   std::cout << “Wasted time:\n\t”  << std::chrono::duration\_cast<min>(end – start).count() << “min\n\t”  << std::chrono::duration\_cast<sec>(end – start).count() << “sec”  << std::endl;   delete[] matrixA;  delete[] atrix;  **glDeleteBuffers**(1, &matrixBufferA);  **glDeleteBuffers**(1, &matrixBufferB);  **glDeleteBuffers**(1, &matrixBufferResult);  **glDeleteProgram**(computeProgram);  **glDeleteShader**(computeShader);  glfwTerminate();  return 0; } |

Листинг 5 – файл main.cpp

В основном коде программы инициируется окно GLFW, инициируется GLEW библиотека и создаться основное окно программы, далее загружается шейдер из файла, добавляем шейдер в программу, генерируем матрицы и выделяем под каждую матрицу буфер, далее идет запуск программы и замер времени работы.

Результат работы программы:

|  |
| --- |
| D:\Projects\CUDA\_CMake\cmake-build-release\RGZ\_MatrixMultiplication.exe  Result Matrix:  517.409 512.922 532.699 515.532 512.514 518.55 508.873 515.035 519.805 527.02 516.658 512.357 516.005 524.787 518.735 51 …. 09.856 513.207 520.702  Wasted time:  0min  4sec  Process finished with exit code 0 |

|  |  |
| --- | --- |
| Размер матрицы | Время выполнения |
| 1<<10 | >1 сек |
| 1<<11 | 4 сек |
| 1<<12 | 33 сек |
| 1<<13 | 4м 27сек |
| 1<<14 | 16м 12сек |

В результате программа выводит корректные значения, и большим плюсом является что программа может работать с супербольшими значениями матриц (>2^13) и не выпадать в ошибку, на подобном CUDA коде такое не удавалось.