Выполнила: Белоусова Е., ИП-911

Задача

Сравнительный анализ производительности программ, реализующих алгоритмы линейной алгебры с использованием библиотек Thrust, cuBLAS и «сырого» CUDA C кода.

Сравнение производительности программ на основе интерфейса CUDA и на основе OpenGL - вычислительных шейдеров.

Описание работы программы

Реализуем алгоритм saxpy с использованием библиотек Thrust, cuBLAS, CUDA C и на основе OpenGL – вычислительных шейдеров.

Thrust предоставляет всего два контейнера - thrust::host_vector и thrust::device_vector. Создадим вектора для x и y, заполним их индексами. Далее создадим вектора на девайсе и скопируем туда вектора с хоста. Создадим функтор, который с помощью метода transform применим к содержимому векторов. Скопируем полученные значения на хост, проверим, соответствует ли результат действительности, используя простой перебор и сравнение значений.

CuBLAS – библиотека, реализующая алгоритмы линейной алгебры. Воспользуемся методом cublasSaxpy. Но перед этим необходимо вызвать cublasInit() для инициализации CUBLAS. В конце необходимо вызвать cublasShutdown для освобождения ресурсов, используемых библиотекой. cublasMalloc позволяет создать объект в пространстве памяти GPU. С помощью функций cublasSetVector и cublasGetVector будем копировать векторы из пространства памяти хоста в пространство памяти девайса и наоборот. В конце проверим, проверим, соответствует ли результат действительности, используя простой перебор и сравнение значений.

Для реализации saxpy с помощью сырого CUDA C будем использовать при вызове ядра 256 нитей.

Далее реализуем алгоритм на основе OpenGL - вычислительных шейдеров.

В функции InitBuffers:

Используем glGenBuffers для генерации n имен объектов буфера, передадим количество необходимых буфферов и массива указателей на идентификаторы. С помощью glBindBuffer укажем назначение буфферов - GL_SHADER_STORAGE_BUFFER. glBufferData - создает и инициализирует

хранилище данных буферного объекта - GL_DYNAMIC_DRAW – содержимое будет меняться. Функция *glBindBufferBase* в качестве параметров ожидает идентификатор цели привязки буфера – индекс точки привязки и буфер. glUseProgram устанавливает программный объект, указанный программой, как часть текущего состояния рендеринга. Один или несколько исполняемых файлов создаются в программном объекте путем успешного присоединения объектов шейдера. Получает на вход дескриптор программы.

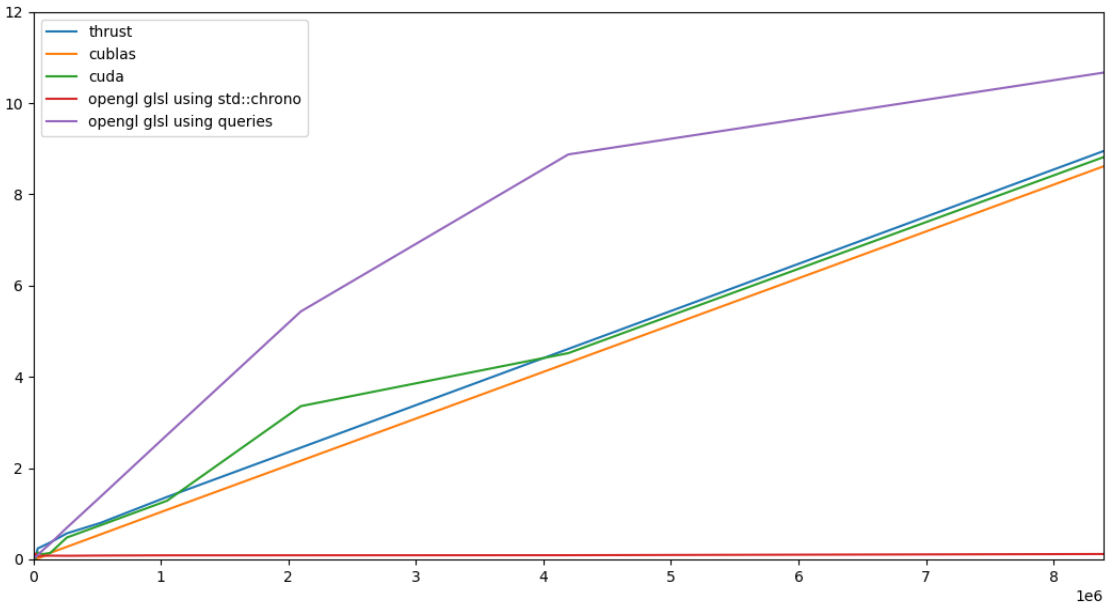
Сама программа создается в функциях genInitProg (инициализация векторов) и genTransformProg(saxpy). Внутри этих функций сначала создаем программу, получаем ее дескриптор. Далее создаем вычислительный шейдер. Записываем непосредственно текст программы на GLSL. Далее создаем исходник для компиляции и компоновки, компилируем программу, проверяем на ошибки. Далее цепляем шейдер к программе, компонуем и снова проверяем на ошибки. Возвращаем дескриптор. Далее запускаем ядро с помощью glDispatchCompute, указывая параметры вычисления. Ждем синхронизации с помощью glMemoryBarrier и освобождаем ресурсы.

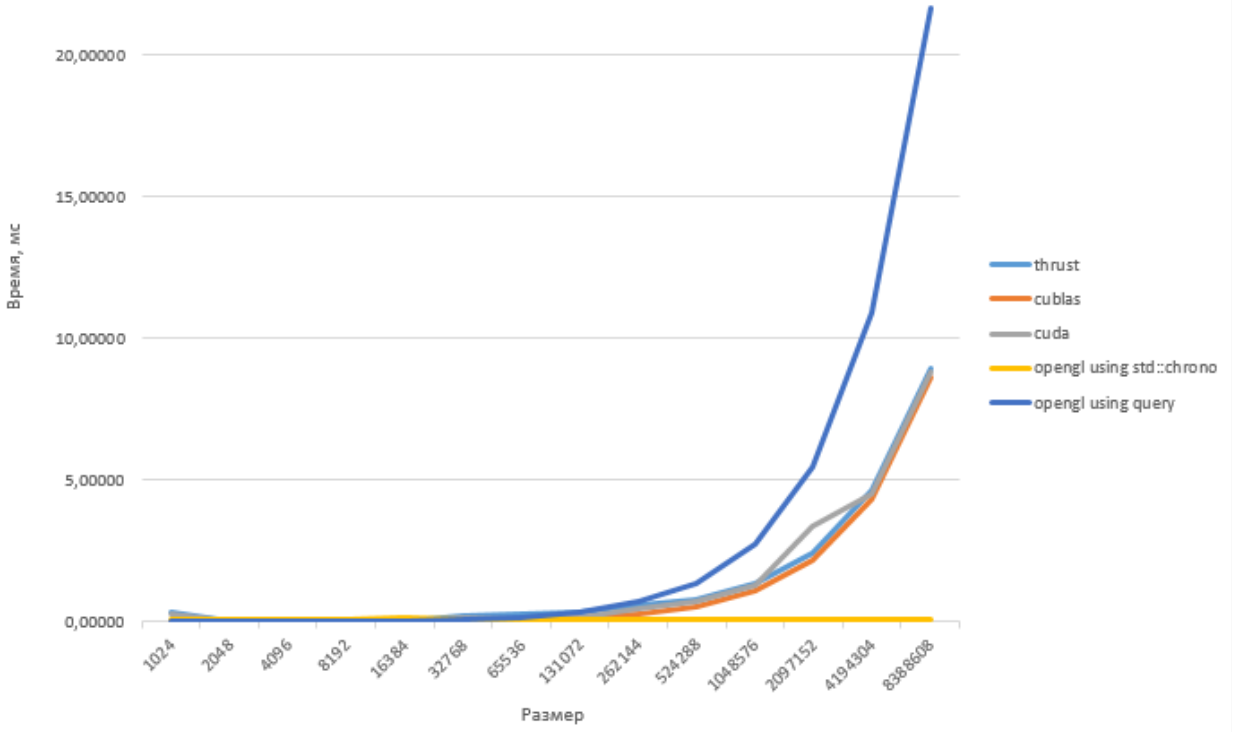
Для проверки результатов используем функцию glMapBuffer, которая возвращает указатель на память текущего связанного буфера, чтобы работать с ним. Копируем данные на хост и проверяем.

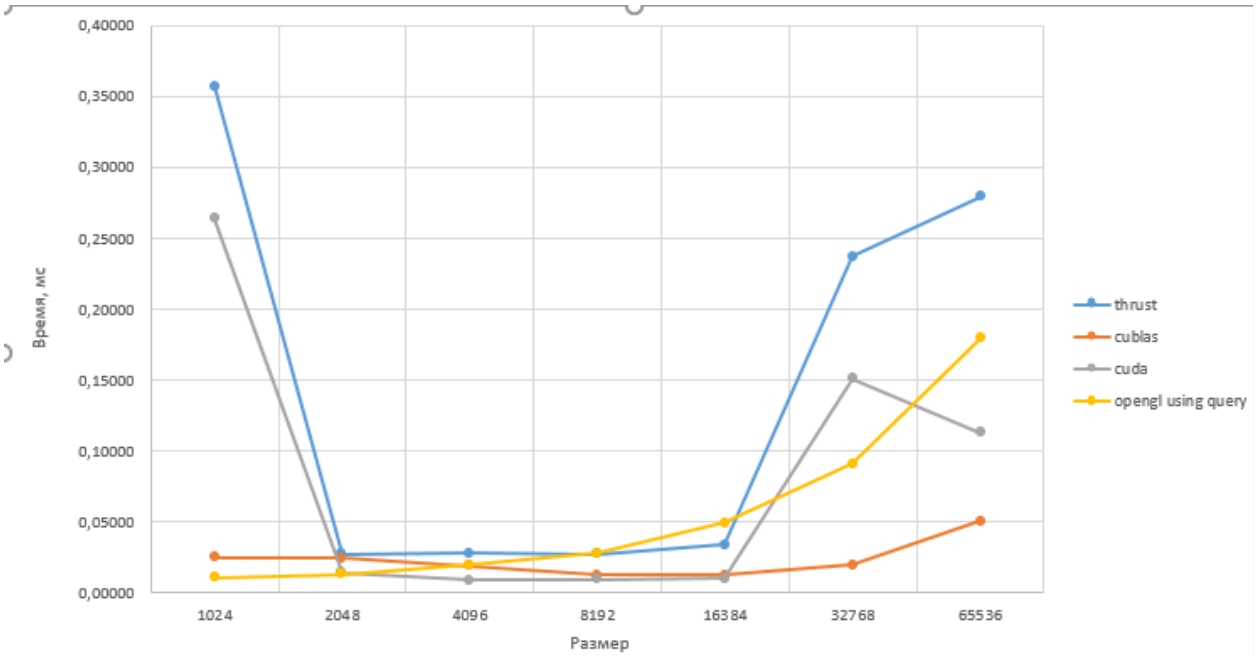
Для оценки времени воспользуемся std::chrono и асинхронные запросы.

Оценим результаты выполнения программ:

N	thrust	cublas	cuda	opengl using std::chrono	opengl using query
1024	0.35696	0.025152	0.263584	0,067	0.010944
2048	0.027712	0.02432	0.014304	0,068	0.013088
4096	0.02832	0.01888	0.009088	0,074	0.019456
8192	0.02752	0.01248	0.009632	0,069	0.028288
16384	0.034144	0.012672	0.010464	0,124	0.049440
32768	0.236992	0.01984	0.150816	0,069	0.091104
65536	0.2792	0.050624	0.112832	0,075	0.179360
131072	0.36784	0.140192	0.136992	0,079	0.342592
262144	0.56848	0.273184	0.477184	0,076	0.692416
524288	0.79568	0.543456	0.744128	0,081	1.366880
1048576	1.36774	1.08554	1.28432	0,086	2.728896
2097152	2.44816	2.16051	3.35789	0,087	5.434048
4194304	4.61187	4.30899	4.51968	0,089	8.876640
8388608	8.9465	8.61187	8.81411	0,115	10.669


thrust
cublas
cuda
opengl glsl using std::chrono
opengl glsl using queries
12
10
8
6
4
2
0
0
1
2
3
4
5
6
7
8
1e6


20,00000
15,00000
10,00000
5,00000
0,00000
Время, мс
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
Размер
thrust
cublas
cuda
opengl using std::chrono
opengl using query



Листинг

```
//main.cpp
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <stdio.h>
#include <malloc.h>
const unsigned int window_width = 512;
const unsigned int window_height = 512;
void initGL();
GLuint* bufferID;
void initBuffers(GLuint*&);
void transformBuffers(GLuint*);
void outputBuffers(GLuint* bufferID, int id);
void checkResult();

int main() {

	initGL();
	bufferID = (GLuint*)calloc(2, sizeof(GLuint));
	initBuffers(bufferID);
```

```
	outputBuffers(bufferID, 0);
	transformBuffers(bufferID);
	outputBuffers(bufferID, 1);
	checkResult();
	glDeleteBuffers(2, bufferID);
	free(bufferID);
	glfwTerminate();
	return 0;
}

void initGL() {
	GLFWwindow* window;
	if (!glfwInit()) {
		fprintf(stderr, "Failed to initialize GLFW\n");
		getchar();
		return;
	}
	glfwWindowHint(GLFW_VISIBLE, 0);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
	glfwWindowHint(GLFW_OPENGL_PROFILE,
		GLFW_OPENGL_COMPAT_PROFILE);
	window = glfwCreateWindow(window_width, window_height,
		"Template window", NULL, NULL);
	if (window == NULL) {
		fprintf(stderr, "Failed to open GLFW window. \n");
		getchar();
		glfwTerminate();
		return;
	}
	glfwMakeContextCurrent(window);
	glewExperimental = true;
	if (glewInit() != GLEW_OK) {
```

```
                fprintf(stderr, "Failed to initialize GLEW\n");
                getchar();
                glfwTerminate();
                return;
        }
        return;
}
//csh.cpp
#include <GL/glew.h>
#include <stdio.h>
#include <string>
#include <string.h>
#include <stdlib.h>
#include <chrono>
#include <iostream>

GLuint genTransformProg();
GLuint genInitProg();
void initBuffers(GLuint*& bufferID);
void transformBuffers(GLuint* bufferID);
const float alpha = 2.0f;
const int N = 1 << 10;

float* startData = new float[N];
float* result = new float[N];



void checkErrors(std::string desc) {
        GLenum e = glGetError();
        if (e != GL_NO_ERROR) {
                fprintf(stderr, "OpenGL error in \"%s\": %s (%d)\n", desc.c_str(),
                        gluErrorString(e), e);
                exit(20);
        }
```

```
}


void initBuffers(GLuint*& bufferID) {
	glGenBuffers(2, bufferID);

	glBindBuffer(GL_SHADER_STORAGE_BUFFER, bufferID[0]);
	glBufferData(GL_SHADER_STORAGE_BUFFER, N * sizeof(float), 0,
		GL_DYNAMIC_DRAW);

	glBindBuffer(GL_SHADER_STORAGE_BUFFER, bufferID[1]);
	glBufferData(GL_SHADER_STORAGE_BUFFER, N * sizeof(float), 0,
		GL_DYNAMIC_DRAW);

	glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, bufferID[0]);
	glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, bufferID[1]);
	GLuint csInitID = genInitProg();
	glUseProgram(csInitID);
	glDispatchCompute(N/128 , 1 , 1);
	//glDispatchCompute(16 , 1 , 1);
	glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT |
		GL_BUFFER_UPDATE_BARRIER_BIT);
	glDeleteProgram(csInitID);
}

GLuint genInitProg() {
	GLuint progHandle = glCreateProgram();
	GLuint cs = glCreateShader(GL_COMPUTE_SHADER);
	const char* cpSrc[] = {
	"#version 430\n",
	"layout (local_size_x = 128, local_size_y = 1, local_size_z = 1) in; \
	layout(std430, binding = 0) buffer BufferA{float A[];};\
	layout(std430, binding = 1) buffer BufferB{float B[];};\
	void main() {\
```

```
		uint idx = gl_GlobalInvocationID.x;\
		A[idx]=float(idx);\
		B[idx]=float(idx);\
	}"
	};
	int rvalue;
	glShaderSource(cs, 2, cpSrc, NULL);
	glCompileShader(cs);
	glGetShaderiv(cs, GL_COMPILE_STATUS, &rvalue);
	if (!rvalue) {
		fprintf(stderr, "Error in compiling fp\n");
		exit(31);
	}

	glAttachShader(progHandle, cs);
	glLinkProgram(progHandle);
	glGetProgramiv(progHandle, GL_LINK_STATUS, &rvalue);
	if (!rvalue) {
		fprintf(stderr, "Error in linking sp\n");
		exit(32);
	}
	checkErrors("Render shaders");
	return progHandle;
}

GLuint genTransformProg() {
	GLuint progHandle = glCreateProgram();
	GLuint cs = glCreateShader(GL_COMPUTE_SHADER);
	const char* cpSrc[] = {
	"#version 430\n",
	"layout (local_size_x = 128, local_size_y = 1, local_size_z = 1) in; \
	layout(std430, binding = 0) buffer BufferA{float A[];};\
	layout(std430, binding = 1) buffer BufferB{float B[];};\
	uniform float alpha = 2.0f;\
```

```
		void main() {\
			uint idx = gl_GlobalInvocationID.x;\
			B[idx] = alpha * A[idx] + B[idx];\
		}"

		};
		glShaderSource(cs, 2, cpSrc, NULL);
		int rvalue;
		glShaderSource(cs, 2, cpSrc, NULL);
		glCompileShader(cs);
		glGetShaderiv(cs, GL_COMPILE_STATUS, &rvalue);
		if (!rvalue) {
			fprintf(stderr, "Error in compiling fp\n");
			exit(31);
		}

		glAttachShader(progHandle, cs);
		glLinkProgram(progHandle);
		glGetProgramiv(progHandle, GL_LINK_STATUS, &rvalue);
		if (!rvalue) {
			fprintf(stderr, "Error in linking sp\n");
			exit(32);
		}
		checkErrors("Render shaders");
		return progHandle;
}

void transformBuffers(GLuint * bufferID) {
		glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, bufferID[0]);
		glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, bufferID[1]);
		GLuint csTransformID = genTransformProg();
		GLuint koef = glGetUniformLocation(csTransformID, "alpha");
		glUniform1f(koef, alpha);
		glUseProgram(csTransformID);
```

```
	//const auto start = std::chrono::high_resolution_clock::now();
	GLuint timeElapsed = 0;
	GLuint queries[1];
	glGenQueries(1, queries);
	glBeginQuery(GL_TIME_ELAPSED, queries[0]);
	//glDispatchCompute(16, 1, 1);
	std::chrono::steady_clock::time_point pr_StartTime;
	std::chrono::steady_clock::time_point pr_EndTime;
	pr_StartTime = std::chrono::steady_clock::now();
	glDispatchCompute(N/128, 1, 1);
	glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT |
		GL_BUFFER_UPDATE_BARRIER_BIT);
	pr_EndTime = std::chrono::steady_clock::now();
	glEndQuery(GL_TIME_ELAPSED);

	auto Duration = std::chrono::duration_cast<std::chrono::microseconds>(pr_EndTime - pr_StartTime);
	std::cout << "Time: " << Duration.count() << " mks." << std::endl;
	glGetQueryObjectuiv(queries[0], GL_QUERY_RESULT, &timeElapsed);
  //std::cout << "Took " << std::chrono::duration_cast<std::chrono::microseconds>(end - start).count() << " us\n";
  printf("Time: %lu nanoseconds, %f milliseconds\n", timeElapsed, float(timeElapsed)/1000000);
	glDeleteProgram(csTransformID);
}


void outputBuffers(GLuint* bufferID, int id) {
	glBindBuffer(GL_SHADER_STORAGE_BUFFER, bufferID[id]);
	float* data = (float*)glMapBuffer(GL_SHADER_STORAGE_BUFFER,
		GL_READ_ONLY);
	float* hdata = (float*)calloc(N, sizeof(float));
	memcpy(&hdata[0], data, sizeof(float) * N);
	glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
	if  (id == 0) {
		//printf("Data:     ");
```

```
                memcpy(&startData[0], hdata, sizeof(float) * N);
        } else if (id == 1) {
                //printf("Result:  ");
                memcpy(&result[0], hdata, sizeof(float) * N);
        }

        // for (int i = 0; i < N; i++) {
        //      printf("%g\t", hdata[i]);
        // }
        // printf("\n");
        free(hdata);
}

void checkResult() {
        printf("N = %d\n", N);
        for(int i = 0; i < N; i++) {
                if(startData[i] * alpha + startData[i] == result[i]) {
                        continue;
                }
                else {
                        printf("Wrong answer!\n");
                        return;
                }
        }
        printf("Correct answer!\n");
        return;
}
```

//thrustSaxpy.cu

```
#include <thrust/device_vector.h>
#include <thrust/fill.h>
#include <thrust/host_vector.h>
#include <thrust/sequence.h>
#include <thrust/transform.h>
#define ALPHA 2.0f
```

```
#define SZ (1<<23)

using namespace std;

struct functor {
	const float koef;
	functor(float _koef) : koef(_koef) {}
	__host__ __device__ float operator()(float x, float y) { return koef * x + y; }
};

void saxpy(float _koef, thrust::device_vector<float> &x, thrust::device_vector<float> &y)
{
	functor func(_koef);
	thrust::transform(x.begin(), x.end(), y.begin(), y.begin(), func);

}

int main(void)
{
		float *a = (float*)calloc(SZ, sizeof(float));
		ofstream os("N.dat");
		ofstream os1("dataThrust.dat");

		for (int i = 0; i < SZ; i++) {
			a[i] = i;
		}
		for(int i = 10; i <= 23; i++) {
			int N = 1 << i;
			printf("N = %d\n", N);
			os << N << endl;
			cudaEvent_t start, stop;
			float time;
			thrust::host_vector<float> h1(N);
			thrust::host_vector<float> h2(N);
```

```
        float alpha = ALPHA;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        for(int k = 0; k < N; k++){
      h1[k] = a[k];
      h2[k] = a[k];
        }

        thrust::device_vector<float> gpumem1 = h1;
        thrust::device_vector<float> gpumem2 = h2;
        cudaEventRecord(start, 0);
        saxpy(alpha, gpumem1, gpumem2);
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&time, start, stop);
        h1 = gpumem1;
        h2 = gpumem2;
        for(int k = 0; k < N; k++) {
        if(h2[k] == h1[k]*ALPHA + h1[k]) {
                continue;
        } else {
                printf("Thrust: wrong answer!\n");
                return -1;
        }
    printf("%g\t %g\n", h1[k], h2[k]);
        }
        printf("Thrust: correct answer!\n");
        printf("Thrust time: %g ms \n", time);
        os1 << time << endl;

  }
}

//cublasSaxpy.cu
```

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cctype>
#include <list>
#include <stdlib.h>
#include <ctime>
#include <cuda_runtime.h>

#include <cublas.h>
#include <cublas_v2.h>
#define NX 64
#define BATCH 1
#define pi 3.141592
#define SZ (1<<23)
#define ALPHA 2.0f
using namespace std;

int main() {
	float *a = (float*)calloc(SZ, sizeof(float));
	ofstream os1("dataCublas.dat");

	for (int i = 0; i < SZ; i++) {
		a[i] = i;
	}
	for(int i = 10; i <=23; i++) {
		int N = 1 << i;
		printf("N = %d\n", N);
		cudaEvent_t start, stop;
		float time;
		cublasHandle_t handle;
		cublasCreate(&handle);
```

```
    float *res = new float[N];
    float *dev_x, *dev_y;
    cudaMalloc(&dev_x, N * sizeof(float));
    cudaMalloc(&dev_y, N * sizeof(float));
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cublasInit();
    cublasSetVector(N, sizeof(a[0]), a, 1, dev_x, 1);
    cublasSetVector(N, sizeof(a[0]), a, 1, dev_y, 1);
    float alpha = ALPHA;
    cudaEventRecord(start, 0);
    cublasSaxpy(handle, N, &alpha, dev_x, 1, dev_y, 1);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);

    cublasGetVector(N, sizeof(res[0]), dev_y, 1, res, 1);
    cublasShutdown();
    for(int k = 0; k < N; k++) {
    if(res[k] == a[k]*ALPHA + a[k]) {
            continue;
    } else {
            printf("cuBLAS: wrong answer!\n");
            return -1;
    }
    printf("%g\t %g\n", a[k], res[k]);
}
     printf("cuBLAS: correct answer!\n");

     printf("cuBLAS time: %g ms \n", time);
     os1 << time << endl;
    cublasDestroy(handle);
    cudaFree(dev_x);
    cudaFree(dev_y);
```

```
                cudaFreeHost(res);
                cudaDeviceReset();
        }
}

//cuSaxpy.cu
#include <stdio.h>
#include <cuda.h>
#include <malloc.h>
#include <string.h>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cctype>
#include <list>
#include <stdlib.h>
#include <ctime>
#include <cuda_runtime.h>

#define ALPHA 2.0f
#define SZ (1<<23)
using namespace std;

__global__ void cusaxpy(int n, float a, float *x, float *y)
{
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < n) y[i] = a*x[i] + y[i];
}



int main() {
        float *a = (float*)calloc(SZ, sizeof(float));
```

```
ofstream os1("dataCuda.dat");


for (int i = 0; i < SZ; i++) {
        a[i] = i;
}
for(int i = 10; i <= 23; i++) {
        int N = 1 << i;
        printf("N = %d\n", N);
        cudaEvent_t start, stop;
        float time;
        float alpha = ALPHA;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
        float *x, *y, *d_x, *d_y;
        x = (float*)malloc(N*sizeof(float));
        y = (float*)malloc(N*sizeof(float));
        cudaMalloc(&d_x, N*sizeof(float));
        cudaMalloc(&d_y, N*sizeof(float));
        for (int k = 0; k < N; k++) {
                x[k] = k;
                y[k] = k;
        }
        cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
        cudaEventRecord(start,0);
        cusaxpy<<<(N)/256, 256>>>(N, alpha, d_x, d_y);
        cudaEventRecord(stop,0);
        cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
        cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);


        for(int k = 0; k < N; k++) {
        if(y[k] == a[k]*ALPHA + a[k]) {
                continue;
```

```
                } else {
                        printf("cuda C: wrong answer!\n");
                        return -1;
                }
                //printf("%g\t %g\n", a[k], y[k]);
            }
            cudaFree(d_x);
                cudaFree(d_y);
                free(x);
                free(y);
        printf("cuda C: correct answer!\n");
        printf("cuda C time: %g ms \n", time);
        os1 << time << endl;
    }
}
```