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"

ОТЧЕТ
по дисциплине «Программирование графических процессоров»
курсовая работа

Выполнил:
студент группы ИП-811
Разумов Д. Б.

Проверил:
преподаватель
Малков Е.А.

Новосибирск 2021

Оглавление

Задание

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

Выполнение задания

Я написал программу, которая сранивает CUDA C, Thrust, cuBLAS на функции SAXPY (Single Alpha X Plus Y — сложение двух массивов, один из которых умножается на некоторый коэффициент). Также я сравнил CUDA C, Thrust, cuBLAS на копирование массива с устройства на устройство и с устройства на хост.

Результаты представлены ниже (6 графиков). Зеленые столбцы — это CUDA C, оранжевые — Thrust, фиолетовые — cuBLAS.

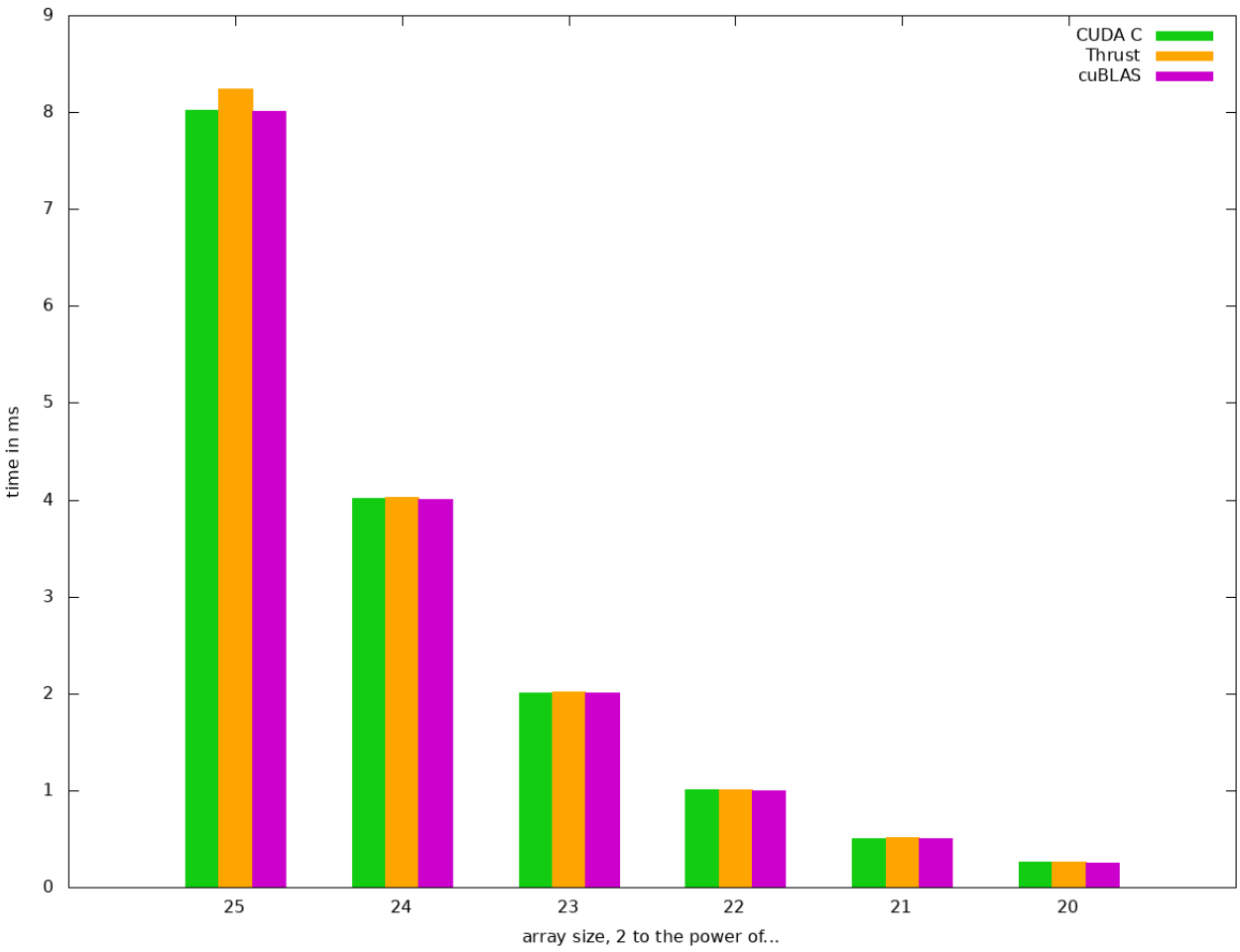

Рисунок 1 - выполнение SAXPY, выполнение в миллисекундах

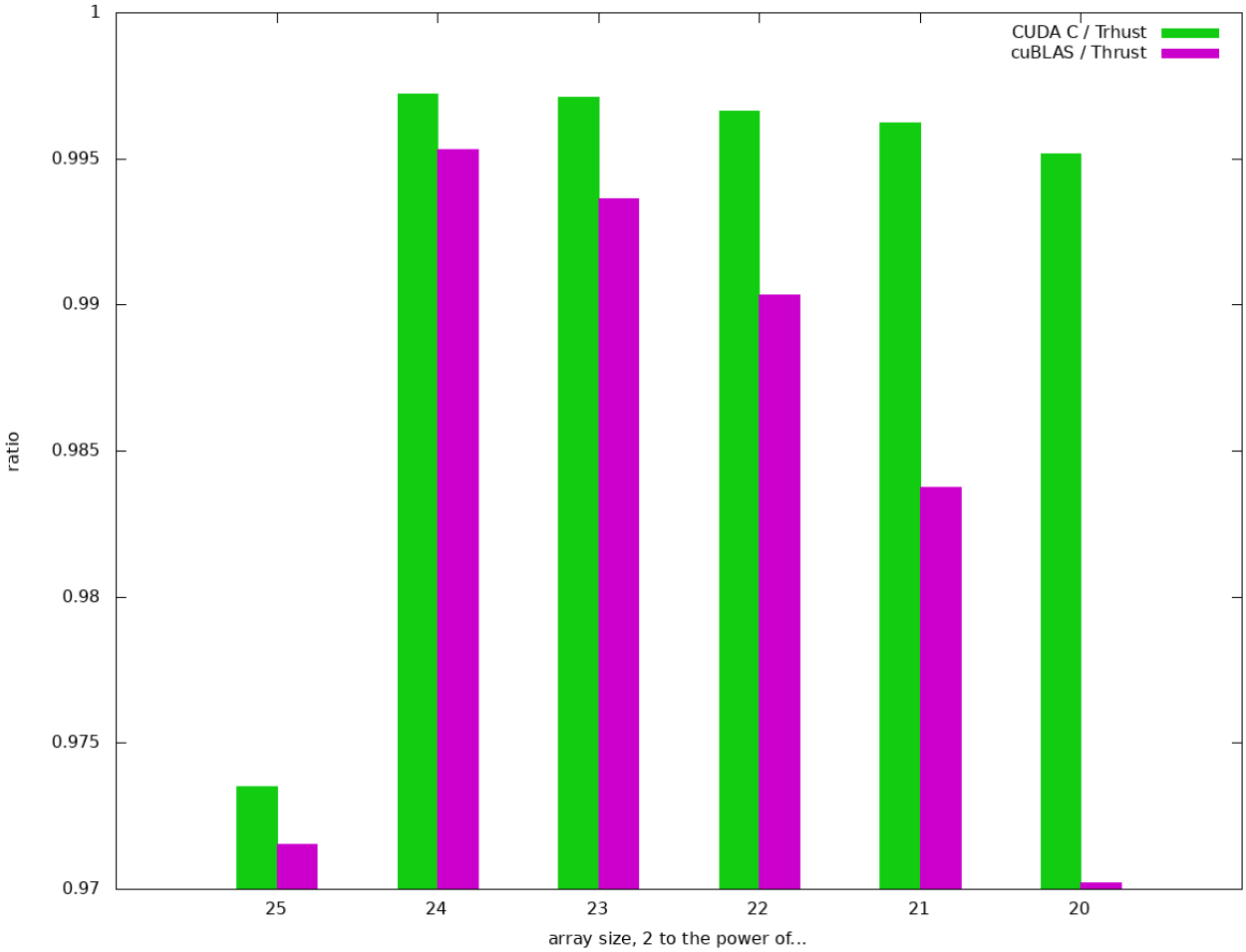

Рисунок 2 - выполнение SAXPY, относительно Thrust

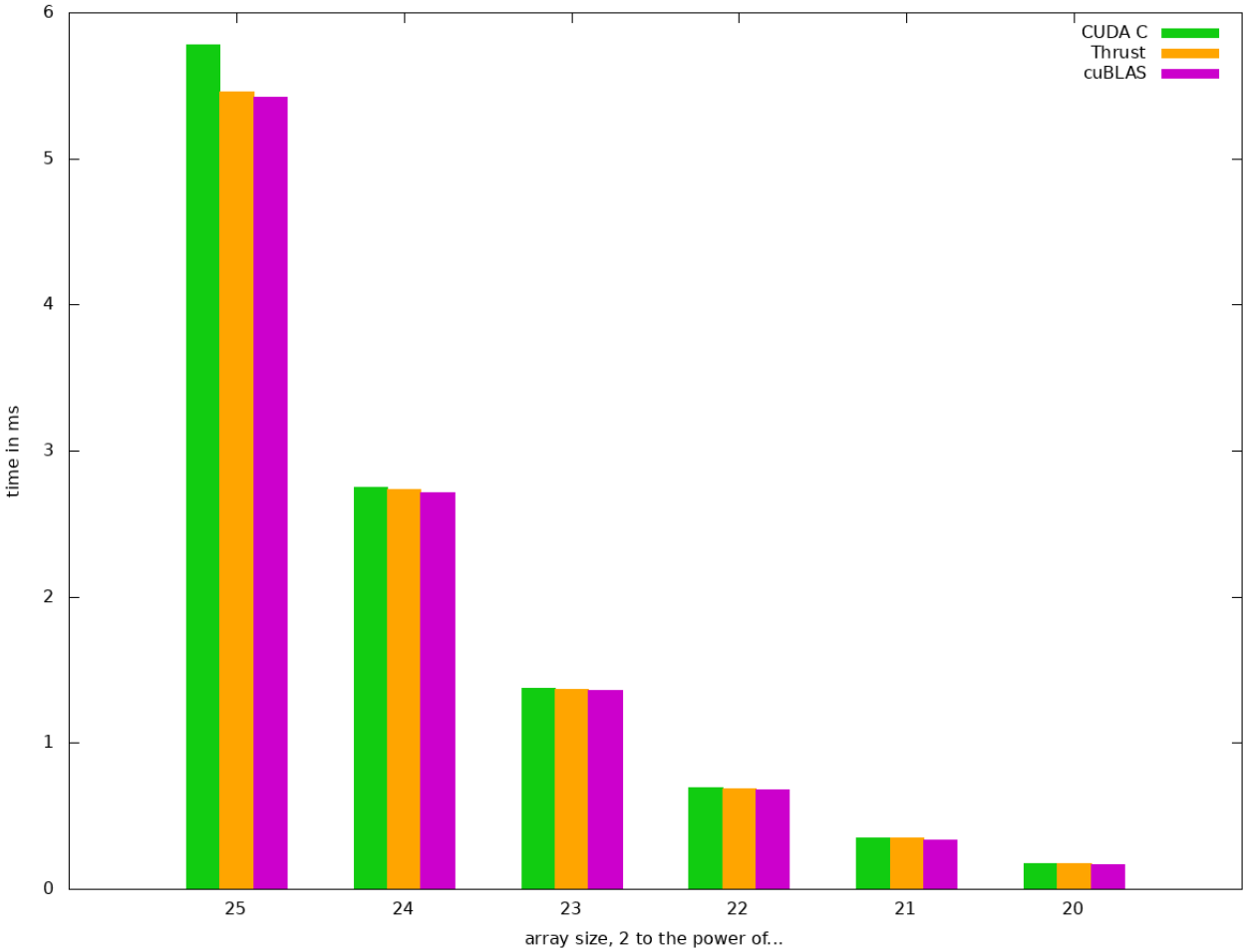

Рисунок 3 - копирование массива с устройства на устройство, время в мс

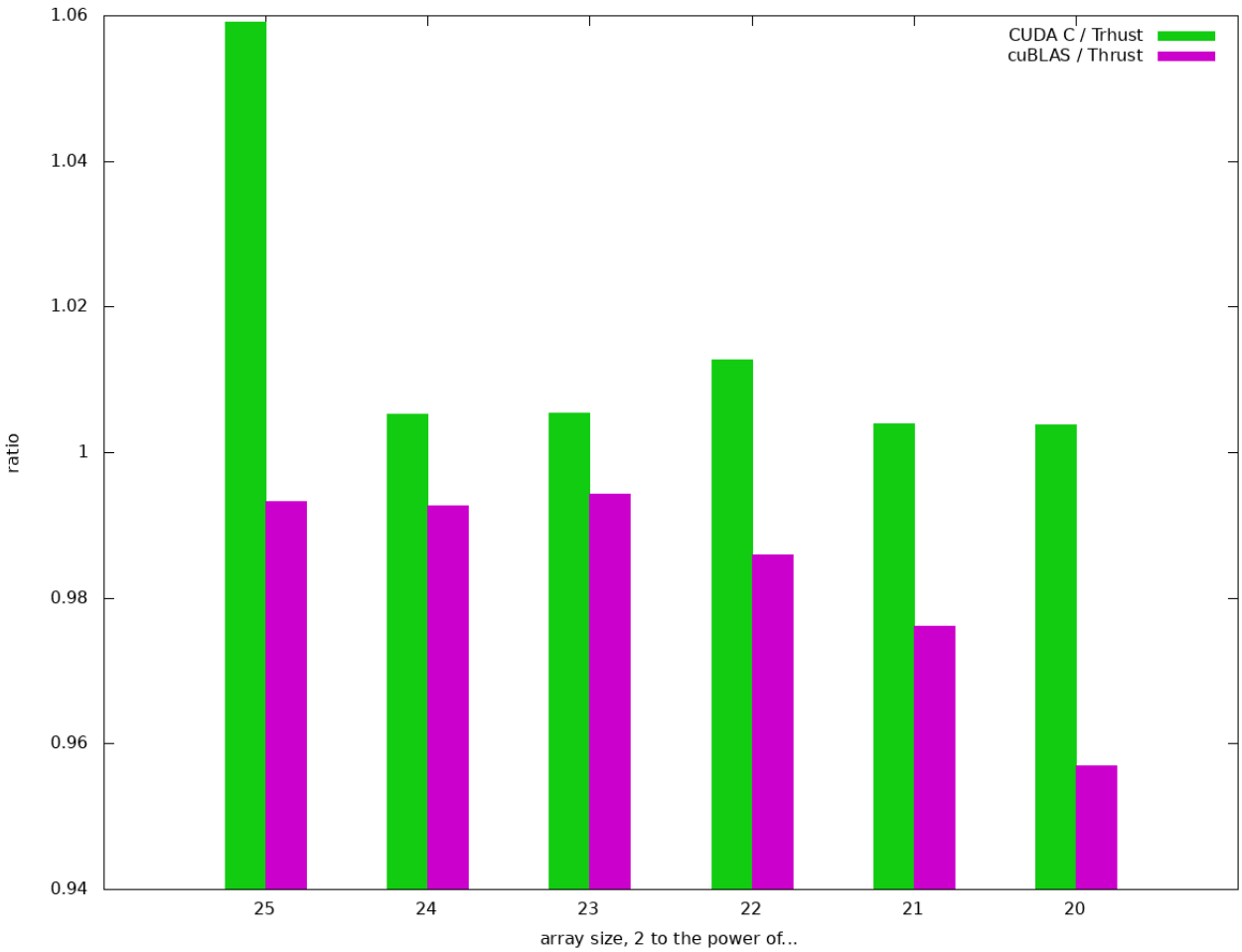

Рисунок 4 - копирование массива с устройства на устройство, относительно Thrust

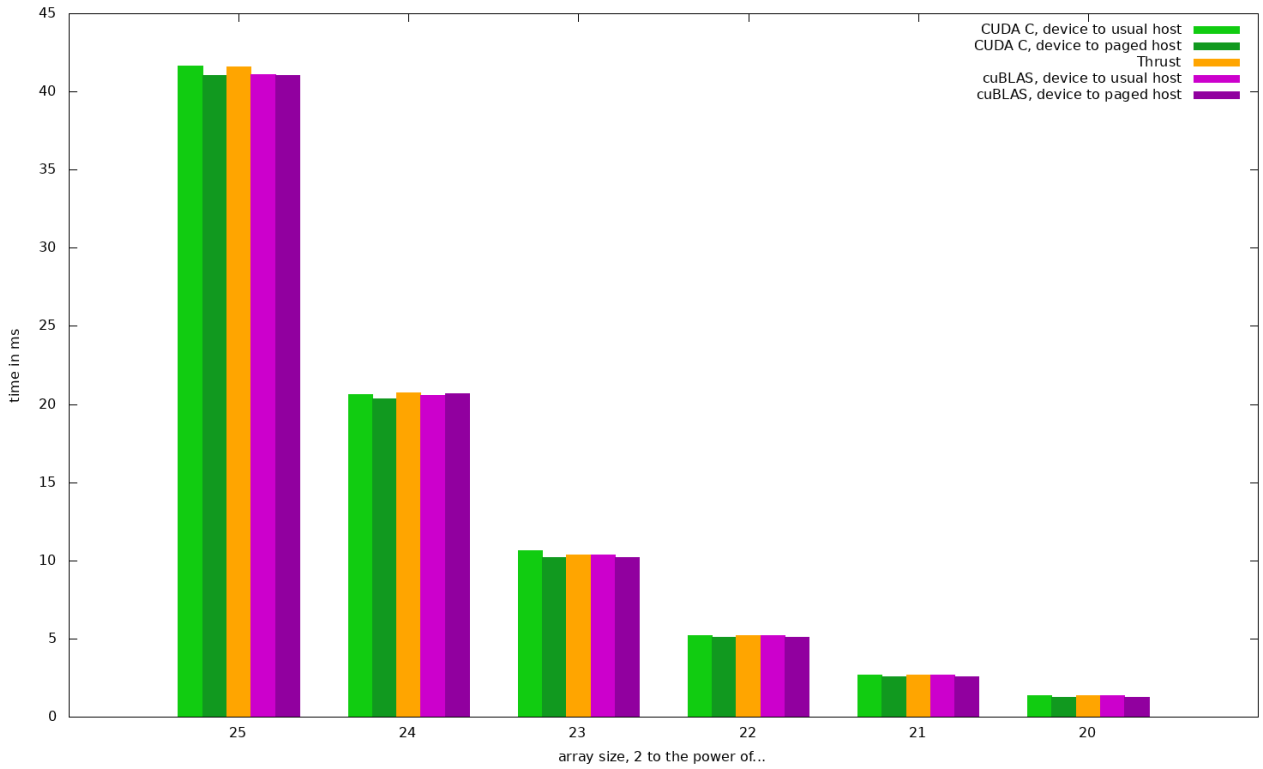

Рисунок 5 - копирование массива с устройства на хост, время в мс

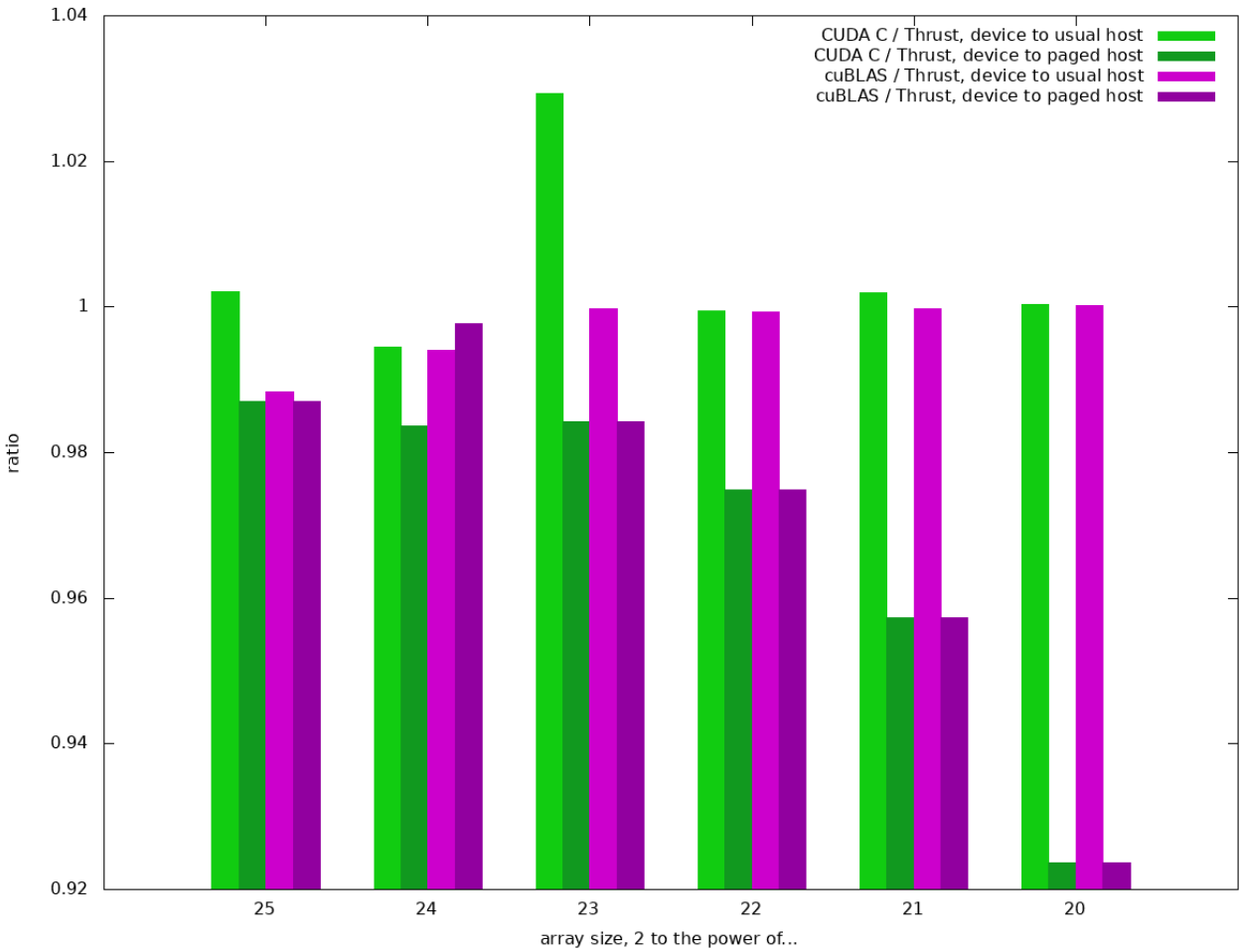

Рисунок 6 - копирование массива с устройства на хост, относительно Thrust

Листинг

Файл main.cu

```
#include "header.h"


int main(int argc, char *argv[]) {
	/// информация об используемом устройстве:
	cudaDeviceProp deviceProp;
	cudaGetDeviceProperties(&deviceProp, 0);
	printf("\ndevice: %s \n\n", deviceProp.name);

	/// Куда-события для замера времени:
	cudaEvent_t start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	/// параметры тестирования:
	int degree = 25; //степень двойки - размер массива
	float alpha = 2.5f; //коэффициент умножения для массива X
	int iterations = 6; //сколько раз запускать SAXPY
	int check_arrays = 0; //сколько элементов массива вывести на экран

	/// пользователь может изменить эти параметры, задав их при запуске:
	if (argc >= 2) {
		degree = atoi(argv[1]); //преобразование строки в long int
		if (degree <= 0) {
			fprintf(stderr, "error, degree should be > 0\n");
			degree = 25;
		}
	}
```

```
if (argc >= 3) {
	alpha = atof(argv[2]); //преобразование строки в float
}
if (argc >= 4) {
	int tmp = atoll(argv[3]);
	if (tmp > 0)
		iterations = tmp;
	else
		fprintf(stderr, "error, iterations should be > 0\n");
}
if (argc >= 5) {
	check_arrays = atoll(argv[4]);
}

/// массив для записи результатов тестирования:
long int arr_size = 1 << degree; //максимальный размер массивов
float *time_arr = (float*) malloc(iterations * TA_COLS * sizeof(float));
for (int i = 0; i < iterations; i++) {
	for (int j = 0; j < TA_COLS; j++)
		time_arr[i * TA_COLS + j] = 0.0f;
}

/// посчитать среднее время и вывести результаты:
printf("performing saxpy_Thrust...\n");
saxpy_thrust(arr_size, alpha, iterations, start, stop, check_arrays, time_arr);
printf("performing saxpy_cuBLAS...\n");
saxpy_cublas(arr_size, alpha, iterations, start, stop, check_arrays, time_arr);
printf("performing saxpy_CUDA...\n");
saxpy_cuda(arr_size, alpha, iterations, start, stop, check_arrays, time_arr);

printf("performing copying_Thrust...\n");
```

```
	copying_thrust(arr_size, iterations, check_arrays, start, stop, time_arr);
	printf("performing copying_cuBLAS...\n");
	copying_cublas(arr_size, iterations, check_arrays, start, stop, time_arr);
	printf("performing copying_CUDA...\n");
	copying_cuda(arr_size, iterations, check_arrays, start, stop, time_arr); //если Куду поместить выше Траста, то ломается...

	///открытие файла, чтобы записать туда time_arr[]
	FILE *fp;
	fp = fopen("graphs/time.csv", "w");
	if (fp == NULL) {
		fprintf(stderr, "error: can't open graphs/time.dat\n");
		exit(EXIT_FAILURE);
	}

	/// подписи к столбцам в .csv файле
	fprintf(fp, "arr_size;saxpy_CUDA;saxpy_Thrust;saxpy_cuBLAS;");
	fprintf(fp, "dev_to_dev_CUDA;dev_to_dev_Thrust;dev_to_dev_cuBLAS;");
	fprintf(fp, "dev_to_usual_host_CUDA;dev_to_paged_host_CUDA;");
	fprintf(fp, "dev_to_host_Thrust;");
	fprintf(fp, "dev_to_usual_host_cuBLAS;dev_to_paged_host_cuBLAS\n");

	/// запись полученного времени из массива в файл:
	for (int i = 0; i < iterations; i++) {
		fprintf(fp, "%d;", degree - i);
		for (int j = 0; j < TA_COLS; j++) {
			fprintf(fp, "%g;", time_arr[i * TA_COLS + j]);
		}
		fprintf(fp, "\n");
	}
```

```
fclose(fp);
fp = fopen("graphs/ratio.csv", "w");
if (fp == NULL) {
	fprintf(stderr, "error: can't open graphs/ratio.dat\n");
	exit(EXIT_FAILURE);
}

/// подписи к столбцам в новом .csv файле
fprintf(fp, "arr_size;saxpy CUDA / Thrust;saxpy cuBLAS / Thrust;");
fprintf(fp, "DevToDev CUDA / Thrust; DevToDev cuBLAS / Thrust;");
fprintf(fp, "DevToHostUsual CUDA / Thrust; DevToHostPaged CUDA / Thrust;");
fprintf(fp, "DevToHostUsual cuBLAS / Thrust; DevToHostPaged cuBLAS / Thrust\n");

for (int i = 0; i < iterations; i++) {
	fprintf(fp, "%d;", degree - i);
	//время saxpy-CUDA поделить на время saxpy-Thrust:
	fprintf(fp, "%g;", time_arr[i * TA_COLS] / time_arr[i * TA_COLS + 1]);
	//время saxpy-cuBLAS поделить на время saxpy-Thrust:
	fprintf(fp, "%g;", time_arr[i * TA_COLS + 2] / time_arr[i * TA_COLS + 1]);
	//время копирования DevToDev, CUDA / Thrust:
	fprintf(fp, "%g;", time_arr[i * TA_COLS + 3] / time_arr[i * TA_COLS + 4]);
	//время копирования DevToDev, cuBLAS / Thrust:
	fprintf(fp, "%g;", time_arr[i * TA_COLS + 5] / time_arr[i * TA_COLS + 4]);
	//время копирования DevToHostUsual, CUDA / Thrust:
	fprintf(fp, "%g;", time_arr[i * TA_COLS + 6] / time_arr[i * TA_COLS + 8]);
	//время копирования DevToHostPaged, CUDA / Thrust:
	fprintf(fp, "%g;", time_arr[i * TA_COLS + 7] / time_arr[i * TA_COLS + 8]);
	//время копирования DevToHostUsual, cuBLAS / Thrust:
	fprintf(fp, "%g;", time_arr[i * TA_COLS + 9] / time_arr[i * TA_COLS + 8]);
	//время копирования DevToHostPaged, cuBLAS / Thrust:
	fprintf(fp, "%g;", time_arr[i * TA_COLS + 10] / time_arr[i * TA_COLS + 8]);
```

```
            fprintf(fp, "\n");
        }

        cudaEventDestroy(start);
        cudaEventDestroy(stop);
        printf("finish successful\n");
        exit(EXIT_SUCCESS);
}
```

Файл header.h

```
#ifndef HEADER_H
#define HEADER_H

#include <stdio.h>
#include <stdlib.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/fill.h>
#include <thrust/sequence.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"

#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);\
        }\
```

```
} //макрос для обработки ошибок CUDA

#define CUBLAS_CHECK_RETURN(value) {\
	cublasStatus_t stat = value;\
	if (stat != CUBLAS_STATUS_SUCCESS) {\
		fprintf(stderr, "Error at line %d in file %s\n",\
			__LINE__, __FILE__);\
		exit(1);\
	}\
} //макрос для обработки ошибок CUBLAS

#define TA_COLS 11 //количество столбцов в массиве time_arr
#define NOKR 1 //number of kernel runs

void saxpy_cuda(long int arr_size, float alpha, int iterations,
		cudaEvent_t start, cudaEvent_t stop, int check_arrays, float *time_arr);
void copying_cuda(long int arr_size, int iterations, int check_arrays,
		cudaEvent_t start, cudaEvent_t stop, float *time_arr);

void saxpy_thrust(long int arr_size, float alpha, int iterations,
		cudaEvent_t start, cudaEvent_t stop, int check_arrays, float *time_arr);
void copying_thrust(long int arr_size, int iterations, int check_arrays,
		cudaEvent_t start, cudaEvent_t stop, float *time_arr);

void saxpy_cublas(long int arr_size, float alpha, int iterations,
		cudaEvent_t start, cudaEvent_t stop, int check_arrays, float *time_arr);
void copying_cublas(long int arr_size, int iterations, int check_arrays,
		cudaEvent_t start, cudaEvent_t stop, float *time_arr);

#endif
```

Файл test_cuda.cu

```
#include "header.h"


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


void saxpy_cuda(long int arr_size, float alpha, int iterations,
	cudaEvent_t start, cudaEvent_t stop, int check_arrays, float *time_arr)
{
	/// создание массивов:
	long int size_in_bytes = arr_size * sizeof(float);
	float *X_dev;
	cudaMalloc((void **) &X_dev, size_in_bytes);
	float *Y_dev;
	cudaMalloc((void **) &Y_dev, size_in_bytes);
	float *X_hos;
	cudaMallocHost((void **) &X_hos, size_in_bytes);
	float *Y_hos;
	cudaMallocHost((void **) &Y_hos, size_in_bytes);

	/// заполнение массивов:
	for (int i = 0; i < arr_size; i++){
		X_hos[i] = (float)i;
	}
	memset(Y_hos, 0, size_in_bytes);
```

```
/// копирование на массивы устройства:
cudaMemcpy(X_dev, X_hos, size_in_bytes, cudaMemcpyHostToDevice);
cudaMemcpy(Y_dev, Y_hos, size_in_bytes, cudaMemcpyHostToDevice);

/// запуск SAXPY на разных размерах массивов
float _time; //затраченное время на SAXPY
long int tmp_size = arr_size; //размер массива, который на каждой итерации уменьшаться вдвое
for (int i = 0; i < iterations; tmp_size = tmp_size >> 1, i++) {
	cudaEventRecord(start, 0);
	for (int j = 0; j < NOKR; j++) //saxpy вызывается несколько раз для большей точности по времени
	{
		saxpy <<< tmp_size / 256, 256 >>> (tmp_size, alpha, X_dev, Y_dev);
		cudaDeviceSynchronize(); //синхронизация потоков
	}
	cudaEventRecord(stop, 0);
	cudaEventSynchronize(stop);
	cudaEventElapsedTime(&_time, start, stop);

	_time /= NOKR; //посчитать среднее время выполнения saxpy
	time_arr[i * TA_COLS] = _time; //записать время в общий массив

	if (check_arrays)
		printf("size of arrays = %ld, CUDA time = %f ms\n", tmp_size, _time);
}

/// проверка:
if (check_arrays > 0) {
	cudaMemcpy(X_hos, X_dev, size_in_bytes, cudaMemcpyDeviceToHost);
	cudaMemcpy(Y_hos, Y_dev, size_in_bytes, cudaMemcpyDeviceToHost);
```

```
		for (int i = 0; i < check_arrays; i++) {
			printf("i = %d;\t X[i] = %g;\t Y[i] = %g\n", i, X_hos[i], Y_hos[i]);
		}
	}
	if (check_arrays)
		printf("\n");

	/// освобождение ресурсов:
	cudaFree(X_dev);
	cudaFree(Y_dev);
	cudaFreeHost(X_hos);
	cudaFreeHost(Y_hos);
}

__global__ void gInitArray(long int arr_size, float* arr) {
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   if (i >= arr_size)
      return;
   arr[i] = (float)i;
}

void copying_cuda(long int arr_size, int iterations, int check_arrays,
	cudaEvent_t start, cudaEvent_t stop, float *time_arr)
{
	/// создание массивов:
   float *host_usual_arr, *host_paged_arr, *dev1_arr, *dev2_arr;
	long int size_in_bytes = arr_size * sizeof(float);
   host_usual_arr = (float*)malloc(size_in_bytes); //выделение обычной памяти на хосте
   cudaHostAlloc((void**)&host_paged_arr, size_in_bytes, cudaHostAllocDefault); //выделение закрепленной (paged-locked) памяти на хосте
   cudaMalloc((void**)&dev1_arr, size_in_bytes); //выделение памяти на девайсе
```

```
cudaMalloc((void**)&dev2_arr, size_in_bytes);


	/// заполнение массива:
gInitArray <<< arr_size / 256, 256 >>> (arr_size, dev1_arr);
cudaDeviceSynchronize();


/// запуск на разных размерах массивов:
	float _time; //затраченное время
	long int tmp_size = arr_size; //размер массива, который на каждой итерации уменьшаться вдвое
	for (int i = 0; i < iterations; tmp_size = tmp_size >> 1, i++) {
		cudaEventRecord(start, 0);
		for (int j = 0; j < NOKR; j++) { //копирование вызывается несколько раз для большей точности по времени
			cudaMemcpy(dev2_arr, dev1_arr, tmp_size * sizeof(float), cudaMemcpyDeviceToDevice);
			cudaDeviceSynchronize(); //синхронизация потоков
		}
		cudaEventRecord(stop, 0);
		cudaEventSynchronize(stop);
		cudaEventElapsedTime(&_time, start, stop);

		_time /= NOKR;
		time_arr[i * TA_COLS + 3] = _time;

		if (check_arrays) {
			printf("size of arrays = %ld\n", tmp_size);
			printf("copying device to device, CUDA time = %f ms\n", _time);
		}

		cudaEventRecord(start, 0);
```

```
		for (int j = 0; j < NOKR; j++) { //копирование вызывается несколько раз для большей точности по времени
			cudaMemcpy(host_usual_arr, dev1_arr, tmp_size * sizeof(float), cudaMemcpyDeviceToHost);
			cudaDeviceSynchronize(); //синхронизация потоков
		}
		cudaEventRecord(stop, 0);
		cudaEventSynchronize(stop);
		cudaEventElapsedTime(&_time, start, stop);

		_time /= NOKR;
		time_arr[i * TA_COLS + 6] = _time;

		if (check_arrays) {
			printf("size of arrays = %ld\n", tmp_size);
			printf("copying device to host usual, CUDA time = %f ms\n", _time);
		}

		cudaEventRecord(start, 0);
		for (int j = 0; j < NOKR; j++) //копирование вызывается несколько раз для большей точности по времени
			cudaMemcpy(host_paged_arr, dev1_arr, tmp_size * sizeof(float), cudaMemcpyDeviceToHost);
		cudaEventRecord(stop, 0);
		cudaEventSynchronize(stop);
		cudaEventElapsedTime(&_time, start, stop);

		_time /= NOKR;
		time_arr[i * TA_COLS + 7] = _time;

		if (check_arrays) {
			printf("size of arrays = %ld\n", tmp_size);
```

```
                printf("copying device to host paged, CUDA time = %f ms\n", _time);
            }
        }

        /// освобождение ресурсов:
        cudaFree(dev1_arr);
        cudaFree(dev2_arr);
        cudaFreeHost(host_usual_arr);
        cudaFreeHost(host_paged_arr);
}
```

Файл test_thrust.cu

```
#include "header.h"

struct saxpy_functor
{
        const float a;
        saxpy_functor(float _a) : a(_a) {}
        __host__ __device__
        float operator()(float x, float y) {
            return a * x + y;
        }
};

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);
}
```

```
void saxpy_thrust(long int arr_size, float alpha, int iterations,
		cudaEvent_t start, cudaEvent_t stop, int check_arrays, float *time_arr)
{
	/// создание и заполнение векторов векторов:
	thrust::host_vector<float> X_hos(arr_size);
	thrust::host_vector<float> Y_hos(arr_size);
	thrust::sequence(X_hos.begin(), X_hos.end());
	//thrust::fill(h2.begin(), h2.end(), 0.0);
	thrust::device_vector<float> X_dev = X_hos;
	thrust::device_vector<float> Y_dev = Y_hos;

	/// запуск SAXPY на разных размерах массивов
	float _time; //затраченное время на SAXPY
	long int tmp_size = arr_size; //размер массива, который на каждой итерации уменьшаться вдвое
	for (int i = 0; i < iterations; tmp_size = tmp_size >> 1, i++) {
		X_dev.resize(tmp_size);
		Y_dev.resize(tmp_size);

		cudaEventRecord(start, 0);
		for (int j = 0; j < NOKR; j++) //saxpy вызывается несколько раз для большей точности по времени
			saxpy(alpha, X_dev, Y_dev);
		cudaEventRecord(stop, 0);
		cudaEventSynchronize(stop);
		cudaEventElapsedTime(&_time, start, stop);

		_time /= NOKR;
		time_arr[i * TA_COLS + 1] = _time;
```

```
		if (check_arrays)
			printf("size of arrays = %ld, Thrust time = %f ms\n", tmp_size, _time);
	}

	/// проверка:
	if (check_arrays > 0) {
		X_hos = X_dev;
		Y_hos = Y_dev;
		for (int i = 0; i < check_arrays; i++) {
			printf("i = %d;\t X[i] = %g;\t Y[i] = %g\n", i, X_hos[i], Y_hos[i]);
		}
	}
	if (check_arrays)
		printf("\n");
}

void copying_thrust(long int arr_size, int iterations, int check_arrays,
		cudaEvent_t start, cudaEvent_t stop, float *time_arr)
{
	/// создание и заполнение векторов векторов:
	thrust::host_vector<float> X_hos(arr_size);
	thrust::sequence(X_hos.begin(), X_hos.end());
	thrust::device_vector<float> X_dev = X_hos;
	thrust::device_vector<float> Y_dev = X_hos;

	/// запуск на разных размерах массивов
	float _time; //затраченное время
	long int tmp_size = arr_size; //размер массива, который на каждой итерации уменьшаться вдвое
	for (int i = 0; i < iterations; tmp_size = tmp_size >> 1, i++) {
		X_hos.resize(tmp_size);
```

```
X_dev.resize(tmp_size);
Y_dev.resize(tmp_size);

cudaEventRecord(start, 0);
for (int j = 0; j < NOKR; j++) //копирование вызывается несколько раз для большей точности по времени
    Y_dev = X_dev;
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&_time, start, stop);

_time /= NOKR;
time_arr[i * TA_COLS + 4] = _time;

if (check_arrays) {
    printf("size of arrays = %ld\n", tmp_size);
    printf("copying device to device, Thrust time = %f ms\n", _time);
}

cudaEventRecord(start, 0);
for (int j = 0; j < NOKR; j++) //копирование вызывается несколько раз для большей точности по времени
    X_hos = X_dev;
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&_time, start, stop);

_time /= NOKR;
time_arr[i * TA_COLS + 8] = _time;

if (check_arrays) {
```

```
                printf("size of arrays = %ld\n", tmp_size);
                printf("copying device to host, Thrust time = %f ms\n", _time);
            }
        }
}
```

Файл test_cublas.cu

```
#include "header.h"

void saxpy_cublas(long int arr_size, float alpha, int iterations,
        cudaEvent_t start, cudaEvent_t stop, int check_arrays, float *time_arr)
{
    /// создание массивов:
    long int size_in_bytes = arr_size * sizeof(float);
    float *X_dev;
    cudaMalloc((void **) &X_dev, size_in_bytes);
    float *Y_dev;
    cudaMalloc((void **) &Y_dev, size_in_bytes);
    float *X_hos;
    cudaMallocHost((void **) &X_hos, size_in_bytes);
    float *Y_hos;
    cudaMallocHost((void **) &Y_hos, size_in_bytes);

    /// инициализация библиотеки CUBLAS:
    cublasHandle_t cublas_handle;
    CUBLAS_CHECK_RETURN(cublasCreate(&cublas_handle));

    /// заполнение массивов:
    for (int i=0; i < arr_size; i++){
        X_hos[i] = (float)i;
```

```
	}
	memset(Y_hos, 0, size_in_bytes);

	const int num_rows = arr_size; //arr_size
	const int num_cols = 1; //1
	const size_t elem_size = sizeof(float);

	//Копирование матрицы с числом строк arr_size и одним столбцом с
	//хоста на устройство
	cublasSetMatrix(num_rows, num_cols, elem_size, X_hos,
		num_rows, X_dev, num_rows); //leading dimension
	cudaMemset(Y_dev, 0, size_in_bytes);

	/// запуск SAXPY на разных размерах массивов
	const int stride = 1; //шаг (каждый stride элемент берется из массива)
	float _time; //затраченное время на SAXPY
	long int tmp_size = arr_size; //размер массива, который на каждой итерации уменьшаться вдвое
	for (int i = 0; i < iterations; tmp_size = tmp_size >> 1, i++) {
		cudaEventRecord(start, 0);
		for (int j = 0; j < NOKR; j++) //saxpy вызывается несколько раз для большей точности по времени
			cublasSaxpy(cublas_handle, tmp_size, &alpha, X_dev, stride, Y_dev, stride);
		cudaEventRecord(stop, 0);
		cudaEventSynchronize(stop);
		cudaEventElapsedTime(&_time, start, stop);

		_time /= NOKR;
		time_arr[i * TA_COLS + 2] = _time;

		if (check_arrays)
```

```
                printf("size of arrays = %ld, cuBLAS time = %f ms\n", tmp_size, _time);
        }

        /// проверка:
        if (check_arrays > 0) {
                cudaMemcpy(X_hos, X_dev, size_in_bytes, cudaMemcpyDeviceToHost);
                cudaMemcpy(Y_hos, Y_dev, size_in_bytes, cudaMemcpyDeviceToHost);
                for (int i = 0; i < check_arrays; i++) {
                        printf("i = %d;\t X[i] = %g;\t Y[i] = %g\n", i, X_hos[i], Y_hos[i]);
                }
        }
        if (check_arrays)
                printf("\n");

        /// освобождение ресурсов:
        cublasDestroy(cublas_handle);
        cudaFree(X_dev);
        cudaFree(Y_dev);
        cudaFreeHost(X_hos);
        cudaFreeHost(Y_hos);
}

void copying_cublas(long int arr_size, int iterations, int check_arrays,
                cudaEvent_t start, cudaEvent_t stop, float *time_arr)
{
        /// выделение памяти:
        float *host_usual_arr, *host_paged_arr, *dev1_arr, *dev2_arr;
    //выделение обычной памяти на хосте:
        long int size_in_bytes = arr_size * sizeof(float);
    host_usual_arr = (float*)malloc(size_in_bytes);
    //выделение закрепленной (paged-locked) памяти на хосте:
```

```
cudaHostAlloc((void**)&host_paged_arr, size_in_bytes, cudaHostAllocDefault);
//выделение памяти на девайсе:
cudaMalloc((void**)&dev1_arr, size_in_bytes);
cudaMalloc((void**)&dev2_arr, size_in_bytes);

        /// инициализация библиотеки CUBLAS:
        cublasHandle_t cublas_handle;
        CUBLAS_CHECK_RETURN(cublasCreate(&cublas_handle));

        /// заполнение массивов:
        for (int i=0; i < arr_size; i++) { //заполнить массив последовательностью
                host_usual_arr[i] = (float)i;
        }
        const int num_rows = arr_size; //arr_size
        const int num_cols = 1; //1
        const size_t elem_size = sizeof(float);
        cublasSetMatrix(num_rows, num_cols, elem_size, host_usual_arr,
                num_rows, dev1_arr, num_rows); //leading dimension
        memset(host_usual_arr, 0, size_in_bytes); //убрать последовательность из массива, занулить
        cudaMemset(dev2_arr, 0, size_in_bytes);

    /// копирование массива с разными размерностями
        const int stride = 1; //шаг (каждый stride элемент берется из массива)
        float _time; //затраченное время
        long int tmp_size = arr_size; //размер массива, который на каждой итерации уменьшаться вдвое
        for (int i = 0; i < iterations; tmp_size = tmp_size >> 1, i++) {
                cudaEventRecord(start, 0);
        for (int j = 0; j < NOKR; j++) //копирование вызывается несколько раз для большей точности по времени
```

```
            cublasScopy(cublas_handle, tmp_size, dev1_arr, stride, dev1_arr, stride);
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&_time, start, stop);

        _time /= NOKR;
        time_arr[i * TA_COLS + 5] = _time;

        if (check_arrays) {
            printf("size of arrays = %ld\n", tmp_size);
            printf("copying device to device, cuBLAS time = %f ms\n", _time);
        }

        cudaEventRecord(start, 0);
        for (int j = 0; j < NOKR; j++) //копирование вызывается несколько раз для большей точности по времени
            cublasGetMatrix(tmp_size, num_cols, elem_size, dev1_arr, tmp_size, host_usual_arr, tmp_size);
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&_time, start, stop);

        _time /= NOKR;
        time_arr[i * TA_COLS + 9] = _time;

        if (check_arrays) {
            printf("size of arrays = %ld\n", tmp_size);
            printf("copying device to host usual, cuBLAS time = %f ms\n", _time);
        }

        cudaEventRecord(start, 0);
```

```
		for (int j = 0; j < NOKR; j++) //копирование вызывается несколько раз для большей точности по времени
			cublasGetMatrix(tmp_size, num_cols, elem_size, dev1_arr, tmp_size, host_paged_arr, tmp_size);
		cudaEventRecord(stop, 0);
		cudaEventSynchronize(stop);
		cudaEventElapsedTime(&_time, start, stop);

		_time /= NOKR;
		time_arr[i * TA_COLS + 10] = _time;

		if (check_arrays) {
			printf("size of arrays = %ld\n", tmp_size);
			printf("copying device to host paged, cuBLAS time = %f ms\n", _time);
		}
	}

	/// освобождение ресурсов:
	cublasDestroy(cublas_handle);
	cudaFree(dev1_arr);
	cudaFree(dev2_arr);
	cudaFreeHost(host_usual_arr);
	cudaFreeHost(host_paged_arr);
}
```

Файл makefile

```
all: main.exe

main.exe: obj/main.o obj/test_cuda.o obj/test_thrust.o obj/test_cublas.o
	nvcc -o $@ $^ -lcublas
```

```
obj/main.o: main.cu
	nvcc -c -o $@ $<
obj/test_cuda.o: test_cuda.cu
	nvcc -c -o $@ $<
obj/test_thrust.o: test_thrust.cu
	nvcc -c -o $@ $<
obj/test_cublas.o: test_cublas.cu
	nvcc -c -o $@ $<

.PHONY: all clean run

run:
	./main.exe && gnuplot graphs.gpi

clean:
	rm -rf *.o *.exe

# $@ - автоматическая переменная, вставить файл из цели (то, что до ':')
# $< - автоматическая переменная, вставить имя первой зависимости (после ':')
# $^ - автоматическая переменная, вставить все зависимости
```

Об используемом GPU

Device name: GeForce MX350

Global memory available on device: 2099904512 (2002 MByte)

Shared memory available per block: 49152

Count of 32-bit registers available per block: 65536

Warp size in threads: 32

Maximum pitch in bytes allowed by memory copies: 2147483647

Maximum number of threads per block: 1024

Maximum size of each dimension of a block[0]: 1024

Maximum size of each dimension of a block[1]: 1024

Maximum size of each dimension of a block[2]: 64

Maximum size of each dimension of a grid[0]: 2147483647

Maximum size of each dimension of a grid[1]: 65535

Maximum size of each dimension of a grid[2]: 65535

Clock frequency in kilohertz: 1468000

totalConstMem: 65536

Major compute capability: 6

Minor compute capability: 1

Number of multiprocessors on device: 5

Count of cores: 640

Id current device: 0

Id nearest device to 1.3: 0

Заключение

Функции, написанные на CUDA C, Thrust и cuBLAS работают примерно одинаковое время. Однако cuBLAS все же немного быстрее Thurst и сырого CUDA в вычислениях и копировании данных.