МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

“Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики”

(НИУ ИТМО)

_____

Факультет программной инженерии и компьютерной техники

Направление подготовки 09.04.04 Программная инженерия

Лабораторная работа № 6

“Изучение технологии OpenCL”

По дисциплине “Параллельные вычисления”

Студент группы P4114

Трофимова Полина Владимировна

Преподаватель:

Жданов Андрей Дмитриевич

Санкт-Петербург, 2024 г.

Оглавление

Описание решаемой задачи

1. Вам необходимо реализовать один (для оценки «3») или два (для оценки «4») этапа вашей программы из предыдущих лабораторных работ. При этом вычисления можно проводить как на CPU, так и на GPU (на своё усмотрение, но GPU предпочтительнее).

2. Необязательное задание №1 (для получения оценки «5»).

- Выполнение заданий для оценки «3» и «4».
- Расчёт доверительного интервала.
- Посчитать время двумя способами: с помощью profiling и с помощью обычного замера (как в предыдущих заданиях).
- Оценить накладные расходы, такие как доля времени, проводимого на каждом этапе вычисления («нормированная диаграмма с областями и накоплением»), число строк кода, добавленных при распараллеливании, а также грубая оценка времени, потраченного на распараллеливание (накладные расходы программиста), и т.п.

3. Необязательное задание №2 (для получения бонусов и лучшей итоговой оценки по итогам прохождения дисциплины). Провести вычисления совместно на GPU и CPU (т.е. итерации в некоторой обоснованной пропорции делятся между GPU и CPU, и параллельно на них выполняются).

Краткая характеристика системы

Операционная система: Windows 10 Домашняя

Тип системы: 64-разрядная операционная система

Процессор: AMD Ryzen 7 5700U

Оперативная память: 8ГБ

Количество физических ядер: 8

Количество логических ядер: 16

gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)

Имя устройства	LAPTOP-89IUHH64
Процессор	AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz
Оперативная память	8,00 ГБ (доступно: 7,34 ГБ)
Код устройства	7A026669-2754-4B7F-9181-DDB49100AAC8
Код продукта	00342-41448-15829-AAOEM
Тип системы	64-разрядная операционная система, процессор x64
Перо и сенсорный ввод	Для этого монитора недоступен ввод с помощью пера и сенсорный ввод

Программа lab1.c

```
//#define CL_TARGET_OPENCL_VERSION 120
#define BENCHMARK 1
#define SOURCE_NAME "C_block_form.cl"
#define _CRT_SECURE_NO_WARNINGS


//#include <sys/timeb.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/timeb.h>
#include <CL/cl.h>
//#include <matmul.h>
//#include <matrix_lib.h>
//#include <err_code.h>
//#include <device_picker.h>
//#include "matrix_lib.c"
//#include <cstdio>

#ifdef _OPENMP
#include <omp.h>
#else
#include <sys/time.h>
#endif


void check_error_code(cl_int* err) {
    switch (*err) {
    case CL_INVALID_PROGRAM:
        printf("CL_INVALID_PROGRAM\n");
        break;
    case CL_INVALID_PROGRAM_EXECUTABLE:
        printf("CL_INVALID_PROGRAM_EXECUTABLE\n");
        break;
    case CL_INVALID_KERNEL_NAME:
        printf("CL_INVALID_KERNEL_NAME\n");
        break;
    case CL_INVALID_KERNEL_DEFINITION:
        printf("CL_INVALID_KERNEL_DEFINITION\n");
        break;
    case CL_INVALID_VALUE:
        printf("CL_INVALID_VALUE\n");
        break;
    case CL_OUT_OF_HOST_MEMORY:
        printf("CL_OUT_OF_HOST_MEMORY\n");
        break;
    case CL_INVALID_ARG_INDEX:
        printf("CL_INVALID_ARG_INDEX\n");
        break;
    case CL_INVALID_ARG_VALUE:
        printf("CL_INVALID_ARG_VALUE\n");
        break;
    case CL_INVALID_MEM_OBJECT:
        printf("CL_INVALID_MEM_OBJECT\n");
        break;
    case CL_INVALID_SAMPLER:
        printf("CL_INVALID_SAMPLER\n");
        break;
    case CL_INVALID_ARG_SIZE:
        printf("CL_INVALID_ARG_SIZE\n");
        break;
    case CL_INVALID_COMMAND_QUEUE:
        printf("CL_INVALID_COMMAND_QUEUE\n");
```

```
        break;
    case CL_INVALID_CONTEXT:
        printf("CL_INVALID_CONTEXT\n");
        break;
    case CL_INVALID_KERNEL_ARGS:
        printf("CL_INVALID_KERNEL_ARGS\n");
        break;
    }
}

void check_error( cl_context* ctx, cl_int* err, const char* f_name, const char* subpart
) {
    if (*err != CL_SUCCESS) {
        if (subpart) printf("[%s] %s failed with %d\n", subpart, f_name, *err);
        else printf("%s failed with %d\n", f_name, *err);
        check_error_code(err);
        if (*ctx) {
            clReleaseContext(*ctx);
        }
        exit(1);
    }
}

double get_time()
{
    struct timeb result;
    ftime(&result);
    return 1000.0 * result.time + result.millitm;
    //#ifdef _OPENMP
    //    /* Use omp_get_wtime() if we can */
    //    return omp_get_wtime();
    //#else
    //    /* Use a generic timer */
    //    static int sec = -1;
    //    struct timeval tv;
    //    gettimeofday(&tv, NULL);
    //    if (sec < 0) sec = tv.tv_sec;
    //    return (tv.tv_sec - sec) + 1.0e-6 * tv.tv_usec;
    //#endif
}

//запуск ядра
void run_kernel(const char* kernel_name, cl_kernel kernel, cl_context* context, cl_program*
program, cl_command_queue* queue, int n, int n_args, ...)
{
    cl_int err = CL_SUCCESS;
    va_list valist;
    va_start(valist, n_args);
    err = CL_SUCCESS;
    for (int i = 0; i < n_args; ++i) {
        size_t arg_size = va_arg(valist, size_t);
        void* arg = va_arg(valist, void*);
        //err |= clSetKernelArg(kernel, i, arg_size, arg);
    }
    va_end(valist);
    check_error(context, &err, "clSetKernelArg()", kernel_name);

    //кол-во задач и запуск ядра
    size_t global_work_size = n;
    err = clEnqueueNDRangeKernel(*queue, kernel, 1, NULL, &global_work_size, NULL, 0, NULL,
NULL);
}



//хранить данные
```

```
void init_chunked_args(cl_context* context, cl_mem* src_offset, cl_mem* src_size, int*
src_offset_host, int* src_size_host, int sort_parts, int n) {
    int n_chunk = sort_parts < 2 ? n : ceil((double)n / sort_parts);
    int n_done = 0;
    for (int i = 0; i < sort_parts; ++i) { //разбиение на чанки
        int n_cur_chunk = fmax(fmin((n - n_done), n_chunk), 0);
        src_offset_host[i] = n_done;
        src_size_host[i] = n_cur_chunk;
        n_done += n_cur_chunk;
    }

    cl_int err = CL_SUCCESS;
    *src_offset = clCreateBuffer(*context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sort_parts
* sizeof(cl_int), src_offset_host, &err);
    *src_size = clCreateBuffer(*context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sort_parts *
sizeof(cl_int), src_size_host, &err);

}

//MERGE
void merge_sorted(cl_context* ctx, cl_program* program, cl_command_queue* queue, cl_kernel
merge_sorted_kernel, cl_mem* src, cl_mem* temp,
    int* src_offset_host, int* src_size_host, int sort_parts, int n) {
    cl_int err = CL_SUCCESS;
    for (int i = 1; i < sort_parts; ++i) {
        cl_int offset_1 = 0, offset_2 = src_offset_host[i], offset_dst = 0;
        cl_int n_src_1 = src_offset_host[i], n_src_2 = src_size_host[i];

        int n_will_done = src_offset_host[i] + src_size_host[i];
        run_kernel(
            "merge_sorted", merge_sorted_kernel, ctx, program, queue, 1, 7,
            sizeof(cl_mem*), src, sizeof(cl_mem*), temp,
            sizeof(cl_int), &offset_1, sizeof(cl_int), &offset_2, sizeof(cl_int), &offset_dst,
            sizeof(cl_int), &n_src_1, sizeof(cl_int), &n_src_2
        );
        err = clEnqueueCopyBuffer(*queue, *temp, *src, 0, 0, n_will_done * sizeof(cl_double),
0, NULL, NULL);
        //check_error(ctx, &err, "sort temp -> src clEnqueueCopyBuffer()", NULL);
    }
}

//SORT
void sort(cl_context* ctx, cl_program* program, cl_command_queue* queue, cl_kernel
sort_kernel, cl_kernel merge_sorted_kernel,
    int n_parts, int n, cl_mem* src, cl_mem* temp)
{
    cl_int err = CL_SUCCESS;
    int* src_offset_host = (int*)malloc(n * sizeof(int));
    int* src_size_host = (int*)malloc(n * sizeof(int));
    cl_mem src_offset, src_size;

    init_chunked_args(ctx, &src_offset, &src_size, src_offset_host, src_size_host, n_parts,
n);

    run_kernel("sort", sort_kernel, ctx, program, queue, n_parts, 3,
        sizeof(cl_mem*), &src_offset, sizeof(cl_mem*), &src_size, sizeof(cl_mem*), src);

    err = clEnqueueCopyBuffer(*queue, *src, *temp, 0, 0, n * sizeof(cl_double), 0, NULL,
NULL);

    merge_sorted(ctx, program, queue, merge_sorted_kernel, src, temp, src_offset_host,
src_size_host, n_parts, n);

    free(src_offset_host);
    free(src_size_host);
}
```

```
void reduce_sum(cl_context* ctx, cl_program* program, cl_command_queue* queue, cl_kernel
reduce_sum_kernel,
    int n_parts, int n, cl_mem* src, double* result)
{
    cl_int err = CL_SUCCESS;
    int* src_offset_host = (int*)malloc(n * sizeof(int));
    int* src_size_host = (int*)malloc(n * sizeof(int));
    cl_mem src_offset, src_size;

    init_chunked_args(ctx, &src_offset, &src_size, src_offset_host, src_size_host, n_parts,
n);
    cl_mem dst = clCreateBuffer(*ctx, CL_MEM_READ_WRITE, n * sizeof(cl_double), NULL, &err);



    run_kernel("reduce_sum", reduce_sum_kernel, ctx, program, queue, n_parts, 4,
        sizeof(cl_mem*), &src_offset, sizeof(cl_mem*), &src_size, sizeof(cl_mem*), src,
sizeof(cl_mem*), &dst);

    double* dst_host = (double* )malloc(n * sizeof(double));
    clEnqueueReadBuffer(*queue, dst, CL_TRUE, 0, n * sizeof(cl_double), dst_host, 0, NULL,
NULL);
    *result = 0;
    for (int i = 0; i < n_parts; ++i) {
        *result += dst_host[i];
    }

    free(dst_host);
    free(src_offset_host);
    free(src_size_host);
}

void generate(double* m1_h, double* m2_h, int n1, int n2, int i)
{
    const int A = 6;

    for (int j = 0; j < n1; ++j) {
        m1_h[j] = (rand() % (A * 100)) / 100.0 + 1;
    }
    for (int j = 0; j < n2; ++j) {
        m2_h[j] = A + rand() % (A * 9);
    }
}

void init_benchmarks(double* benchmarking_time, double* benchmarking_results, int n) {
#ifdef BENCHMARK
    for (int i = 0; i < n; ++i) {
        benchmarking_results[i] = 0;
    }
#endif
}

void start_benchmark(double* benchmarking_time, int idx) {
#ifdef BENCHMARK
    benchmarking_time[idx] = get_time();
#endif
}

void finish_benchmark(double* benchmarking_time, double* benchmarking_results, int idx) {
#ifdef BENCHMARK
    benchmarking_results[idx] += get_time() - benchmarking_time[idx];
#endif
}

void show_benchmark_results(double* benchmarking_time, double* benchmarking_results, int n) {
#ifdef BENCHMARK
```

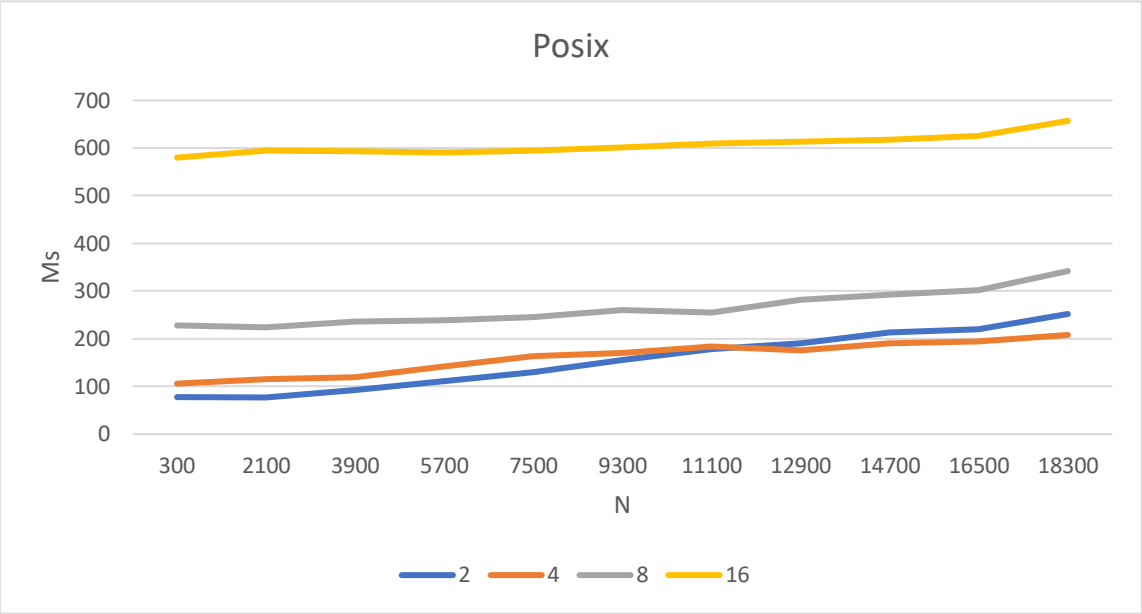
```
    printf("\n\nBENCHMARK\n");
    for (int i = 0; i < n; ++i) {
        printf("%f\n", benchmarking_results[i]);
    }
    printf("\n");
    free(benchmarking_time);
    free(benchmarking_results);
#endif
}

//иниц OpenCL окружения
void init_opencl_env(cl_context* ctx, cl_command_queue* queue,
    cl_program* program, const char** source)
{
    cl_int err;
    cl_platform_id platform = 0;
    cl_device_id device = 0;
    cl_context_properties props[3] = { CL_CONTEXT_PLATFORM, 0, 0 };
    //получение платформы
    err = clGetPlatformIDs(1, &platform, NULL);

    //получение устройства
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);

    //создание контектса (окружение д выполнения OpenCL)
    props[1] = (cl_context_properties)platform;
    *ctx = clCreateContext(props, 1, &device, NULL, NULL, &err);

    //очередь команд на выполнение
    //cl_properties queue_props[] = { CL_QUEUE_PROPERTIES,
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE | CL_QUEUE_PROFILING_ENABLE , 0 };
    *queue = clCreateCommandQueue(*ctx, device, 0, &err);

    //создание программы
    *program = clCreateProgramWithSource(*ctx, 1, source, NULL, &err);
    //построение программы
    clBuildProgram(*program, 1, &device, NULL, NULL, NULL);
}

//создание буферов
void init_buffers(cl_context* ctx, double* m1_h, double* m2_h, cl_mem* m1, cl_mem* m2,
    cl_mem* m2_cpy, int n1, int n2)
{
    cl_int err = CL_SUCCESS;
    *m1 = clCreateBuffer(*ctx, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, n1 *
sizeof(cl_double), m1_h, &err);

    *m2 = clCreateBuffer(*ctx, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, n2 *
sizeof(cl_double), m2_h, &err);

    *m2_cpy = clCreateBuffer(*ctx, CL_MEM_READ_WRITE, n2 * sizeof(cl_double), NULL, &err);

}

//инициализация и создание ядер
void init_kernels(
    cl_context* ctx,
    cl_program* program,
    cl_kernel* ctanh_sqrt,
    cl_kernel* sum_prev,
    cl_kernel* pow_log10,
    cl_kernel* max_2_src,
    cl_kernel* map_sin,
    cl_kernel* sort_kernel,
    cl_kernel* merge_sort_kernel,
    cl_kernel* reduce_sum_kernel
) {
```

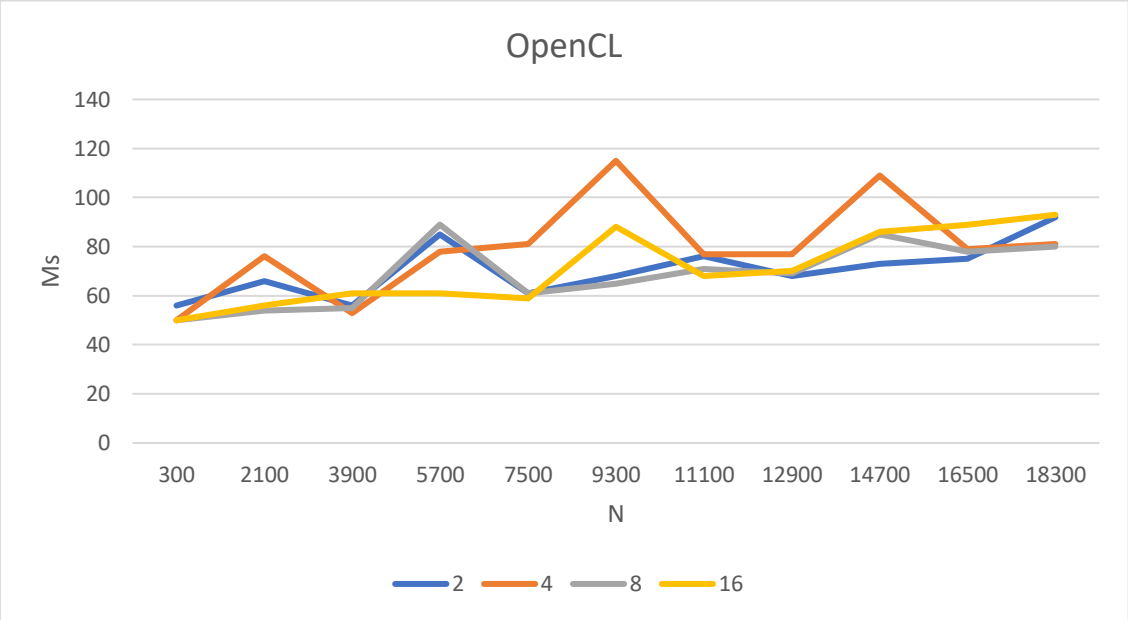
```
    cl_int err = CL_SUCCESS;
    *ctanh_sqrt = clCreateKernel(*program, "tanh_sqrt", &err);

    *sum_prev = clCreateKernel(*program, "sum_prev", &err);

    *pow_log10 = clCreateKernel(*program, "pow_log10", &err);

    *max_2_src = clCreateKernel(*program, "max_2_src", &err);

    *map_sin = clCreateKernel(*program, "map_sin", &err);


    *sort_kernel = clCreateKernel(*program, "sort", &err);

    *merge_sort_kernel = clCreateKernel(*program, "merge_sorted", &err);

    *reduce_sum_kernel = clCreateKernel(*program, "reduce_sum", &err);

}



int main(int argc, char* argv[])
{

    int N_benchmarks = 4;
    double* benchmarking_time = (double*)malloc(N_benchmarks * sizeof(double));
    double* benchmarking_results = (double*)malloc(N_benchmarks * sizeof(double));
    init_benchmarks(benchmarking_time, benchmarking_results, N_benchmarks);
    start_benchmark(benchmarking_time, 0);
    double time_start = get_time();

    /* C-block-form */
    FILE* f;
    long lSize;

    f = fopen(SOURCE_NAME, "rb");

    //определяем размер
    fseek(f, 0L, SEEK_END);
    lSize = ftell(f);
    rewind(f);

    //выделяем память
    const char* source = (const char*)calloc(1, lSize + 1);
    if (!source) fclose(f), fputs("memory alloc fails", stderr), exit(1);

    //читаем в выделеную память
    if (1 != fread((void*)source, lSize, 1, f)) {
        fclose(f), free((void*)source), fputs("entire read fails", stderr), exit(1);
    }
    fclose(f);

    cl_int err;               // error code returned from OpenCL calls
    cl_device_id     device;        // compute device id
    cl_context       context;       // compute context
    cl_command_queue commands;      // compute command queue
    cl_program       program;       // compute program
    //cl_kernel        kernel;        // compute kernel


    init_opencl_env(&context, &commands, &program, &source);

    const int N = atoi(argv[1]);
    const int N_2 = N / 2;
    const int N_separate = argc > 2 ? atoi(argv[2]) : 4;
```

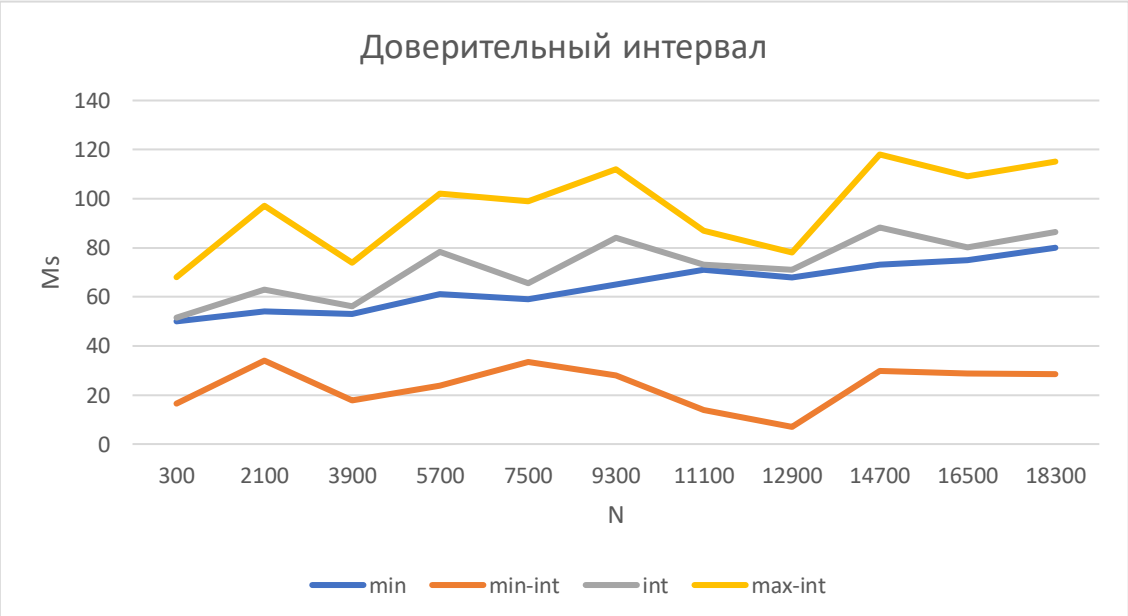
```
    double* m1_h = (double* )malloc(N * sizeof(double));
    double* m2_h = (double* )malloc(N_2 * sizeof(double));

    cl_mem m1, m2, m2_cpy;
    init_buffers(&context, (double*)m1, (double*)m2, &m1, &m2, &m2_cpy, N, N_2);

    cl_kernel tanh_sqrt, sum_prev, pow_log10, max_2_src, map_sin;
    cl_kernel sort_kernel, merge_sort_kernel, reduce_sum_kernel;

    init_kernels(&context, &program, &tanh_sqrt, &sum_prev, &pow_log10, &max_2_src, &map_sin,
        &sort_kernel, &merge_sort_kernel, &reduce_sum_kernel);
    finish_benchmark(benchmarking_time, benchmarking_results, 0);

    for (int i = 0; i < 100; i++) {
        start_benchmark(benchmarking_time, 0);
        generate(m1_h, m2_h, N, N_2, i);
        finish_benchmark(benchmarking_time, benchmarking_results, 0);


        // запуск ядер
        start_benchmark(benchmarking_time, 1);
        run_kernel("ctanh_sqrt", tanh_sqrt, &context, &program, &commands, N, 2,
            sizeof(cl_mem*), &m1, sizeof(cl_mem*), &m1);
        run_kernel("sum_prev", sum_prev, &context, &program, &commands, N_2, 3,
            sizeof(cl_mem*), &m2, sizeof(cl_mem*), &m2_cpy, sizeof(cl_mem*), &m2);
        run_kernel("pow_log10", pow_log10, &context, &program, &commands, N_2, 2,
            sizeof(cl_mem*), &m2, sizeof(cl_mem*), &m2);
        run_kernel("max_2_src", max_2_src, &context, &program, &commands, N_2, 3,
            sizeof(cl_mem*), &m2, sizeof(cl_mem*), &m1, sizeof(cl_mem*), &m2_cpy);
        finish_benchmark(benchmarking_time, benchmarking_results, 1);

        start_benchmark(benchmarking_time, 2);
        sort(&context, &program, &commands, sort_kernel, merge_sort_kernel, N_separate, N_2,
&m2_cpy, &m2);
        finish_benchmark(benchmarking_time, benchmarking_results, 2);

        start_benchmark(benchmarking_time, 3);
        clEnqueueReadBuffer(commands, m2_cpy, CL_TRUE, 0, N_2 * sizeof(cl_double), m2_h, 0,
NULL, NULL);
        int k = 0;
        while (m2_h[k] == 0 && k < N_2 - 1) k++;
        cl_double m2_min = m2_h[k];

        run_kernel("map_sin", map_sin, &context, &program, &commands, N_2, 3,
            sizeof(cl_mem*), &m2_cpy, sizeof(cl_mem*), &m2_cpy, sizeof(cl_double), &m2_min);

        double X = 0;
        reduce_sum(&context, &program, &commands, reduce_sum_kernel, N_separate, N_2, &m2_cpy,
&X);
        printf("%f ", X);
        finish_benchmark(benchmarking_time, benchmarking_results, 3);
    }

    clFinish(commands);
    free(m1_h);
    free(m2_h);
    show_benchmark_results(benchmarking_time, benchmarking_results, N_benchmarks);

    double time_end = get_time();
    printf("\n%f\n", time_end - time_start);
    return 0;
}
```

Результаты

Сравнение времени выполнения







Вывод: в случае с OpenCL время выполнения программы практически не зависит от количества элементом и количества блоков, на которые бьётся сортировка. Процесс вычислений крайне быстр и эффективен, но гораздо больше затрат на накладные расходы. Сама программа получилась сильно сложнее.

Накладные расходы программиста:

строки OpenCL ~650 Posix ~450 OpenMP~350

дни OpenCL 7 дней Posix 3 дня OpenMP 2дня