Федеральное государственное автономное образовательное учреждение высшего образования

«Университет ИТМО»

Факультет ПИиКТ

Дисциплина: Параллельные вычисления

Лабораторная работа 6

OpenCL

Выполнил: Гурин Евгений Иванович

Преподаватель: Жданов Андрей Дмитриевич

Группа: P4116

Санкт-Петербург 2023г.

Задача

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. Дополнительной задание (оценка 5).

- Выполнение заданий для оценки 3 и 4.
- Расчёт доверительного интервала.
- Посчитать время 2 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.
- Необязательное задание для магистрантов с большим количеством свободного времени: проводить вычисления совместно на GPU и CPU (т.е. итерации в некоторой обоснованной пропорции делятся между GPU и CPU, и параллельно на них выполняются).

3. При желании данную лабораторную работу можно написать на CUDA.

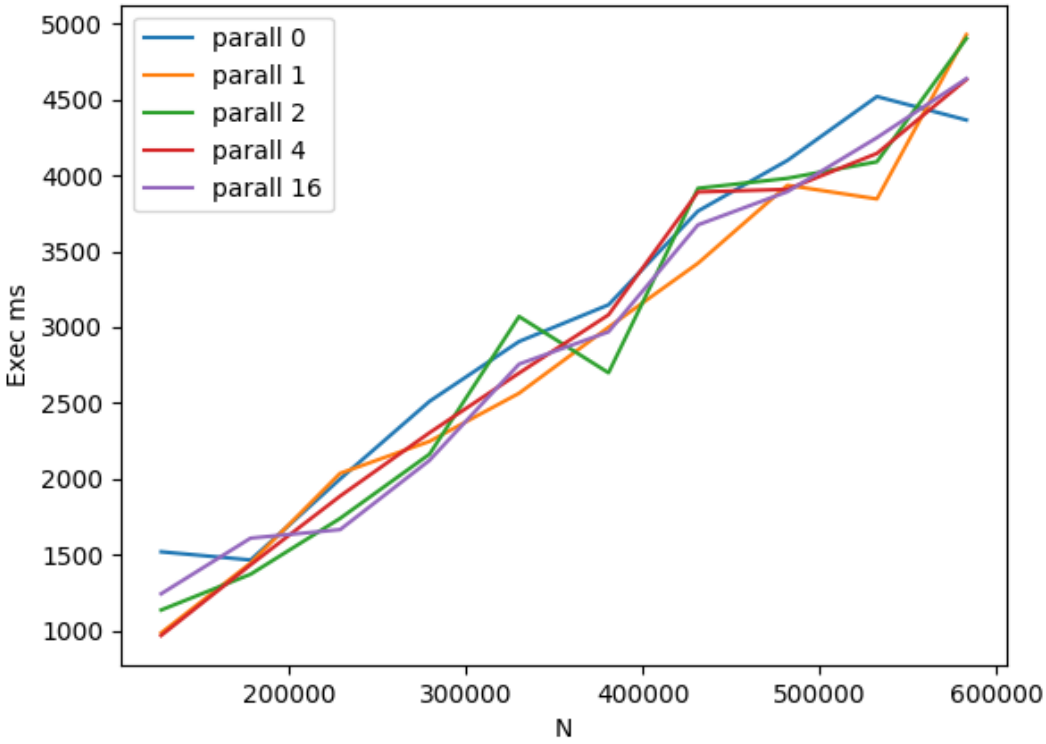
Конфигурация

```
Host Name:                 EGURIN-PC
OS Name:                   Microsoft Windows 11 Pro
OS Version:                10.0.22000 N/A Build 22000
OS Manufacturer:           Microsoft Corporation
OS Configuration:          Standalone Workstation
OS Build Type:             Multiprocessor Free
Registered Owner:          user
Registered Organization:   N/A
Product ID:                00331-10000-00001-AA539
Original Install Date:     02.10.2022, 21:59:41
System Boot Time:          20.03.2023, 2:46:00
System Manufacturer:       ASUS
System Model:              System Product Name
System Type:               x64-based PC
Processor(s):              1 Processor(s) Installed.
                           [01]: AMD64 Family 23 Model 113 Stepping 0 AuthenticAMD ~3600
Mhz
BIOS Version:              American Megatrends Inc. 2803, 27.04.2022
Windows Directory:         C:\Windows
System Directory:          C:\Windows\system32
Boot Device:               \Device\HarddiskVolume2
System Locale:             en-us;English (United States)
Input Locale:              en-us;English (United States)
Time Zone:                 (UTC+03:00) Moscow, St. Petersburg
Total Physical Memory:     32 679 MB
Available Physical Memory: 20 506 MB
Virtual Memory: Max Size:  87 975 MB
Virtual Memory: Available: 19 470 MB
```

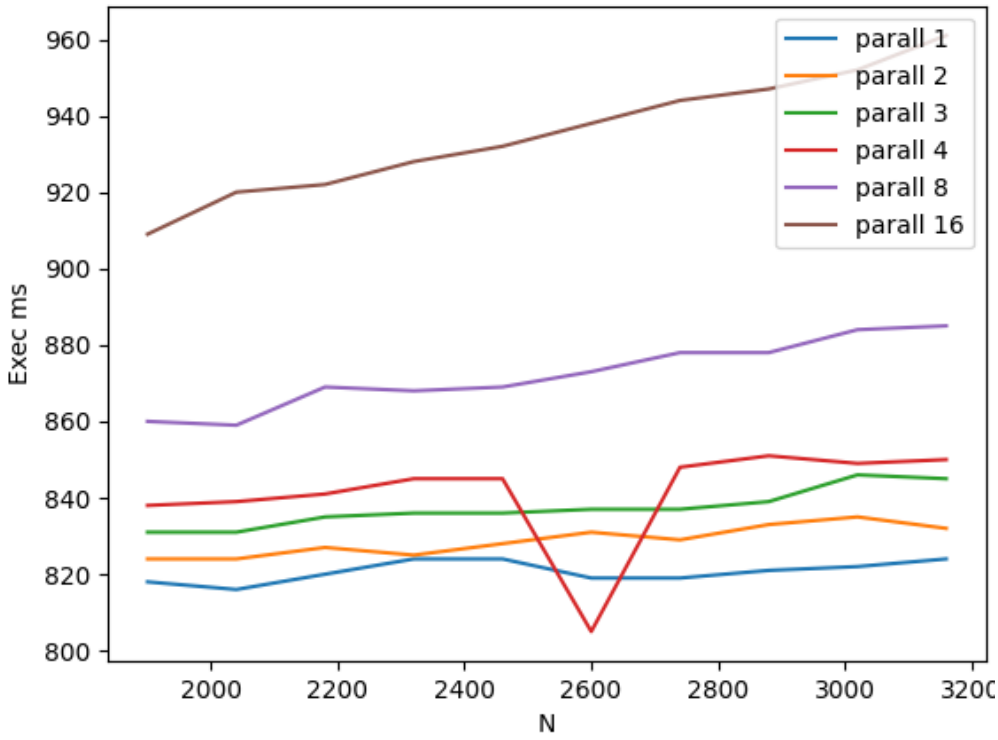
```
Virtual Memory: In Use:    68 505 MB
Page File Location(s):     D:\pagefile.sys
Domain:                    WORKGROUP
Logon Server:              \\EGURIN-PC
Hotfix(s):                 5 Hotfix(s) Installed.
                           [01]: KB5022505
                           [02]: KB5012170
                           [03]: KB5023698
                           [04]: KB5022369
                           [05]: KB5022925
Network Card(s):           4 NIC(s) Installed.
                           [01]: Realtek PCIe 2.5GbE Family Controller
                                 Connection Name: Ethernet
                                 Status:          Media disconnected
                           [02]: Intel(R) Wi-Fi 6 AX200 160MHz
                                 Connection Name: Wi-Fi
                                 DHCP Enabled:    Yes
                                 DHCP Server:     192.168.1.1
                                 IP address(es)
                                 [01]: 192.168.1.47
                                 [02]: fe80::933b:210e:a9a7:2c6e
                           [03]: Bluetooth Device (Personal Area Network)
                                 Connection Name: Bluetooth Network Connection
                                 Status:          Media disconnected
                           [04]: VirtualBox Host-Only Ethernet Adapter
                                 Connection Name: Ethernet 2
                                 DHCP Enabled:    No
                                 IP address(es)
                                 [01]: 192.168.56.1
                                 [02]: fe80::527e:5766:393d:acc6
Hyper-V Requirements:      A hypervisor has been detected. Features required for Hyper-V
will not be displayed.
```

Результаты работы

CLANG (автоматизированное распараллеливание)

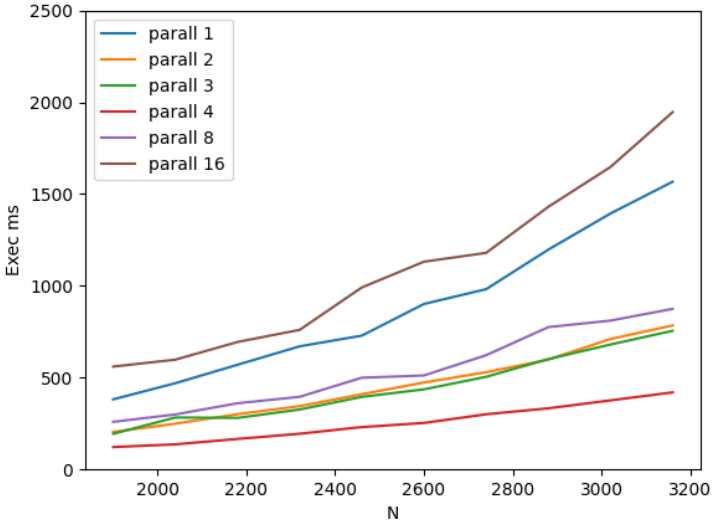


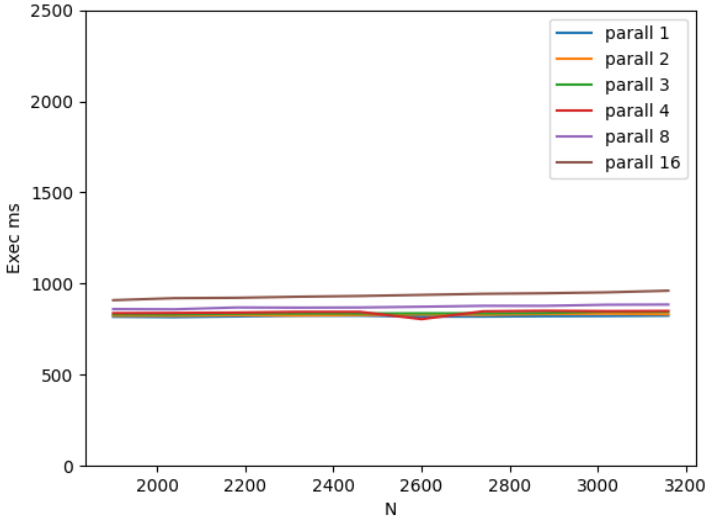
OpenCL



OpenMP **OpenCL**

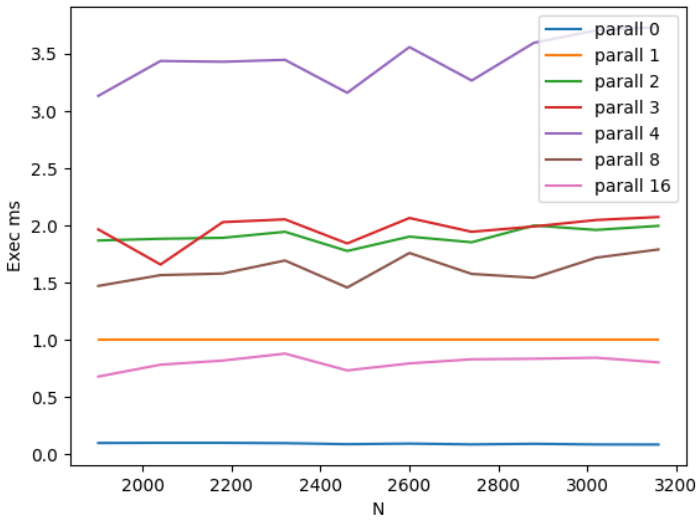
Время выполнения





По графикам можно сделать вывод, что при использовании OpenCL время выполнения для различного количества элементов массива практически совпадает, из чего можно сделать вывод, что сами вычисления занимают малое количество времени, сравнительно с накладными расходами.

Параллельное ускорение



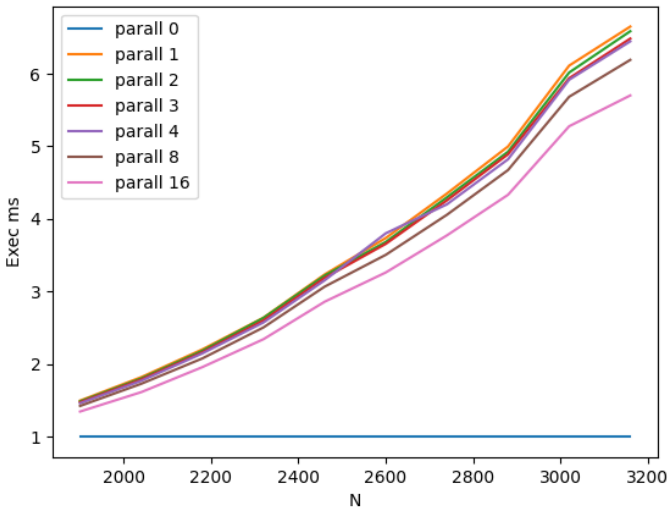
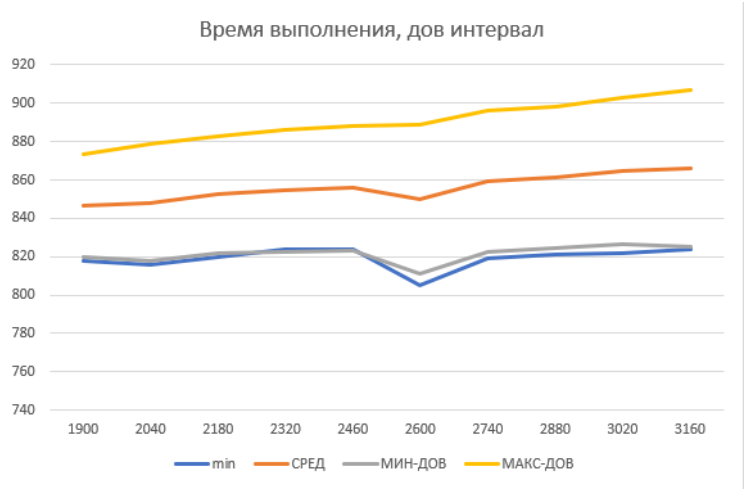
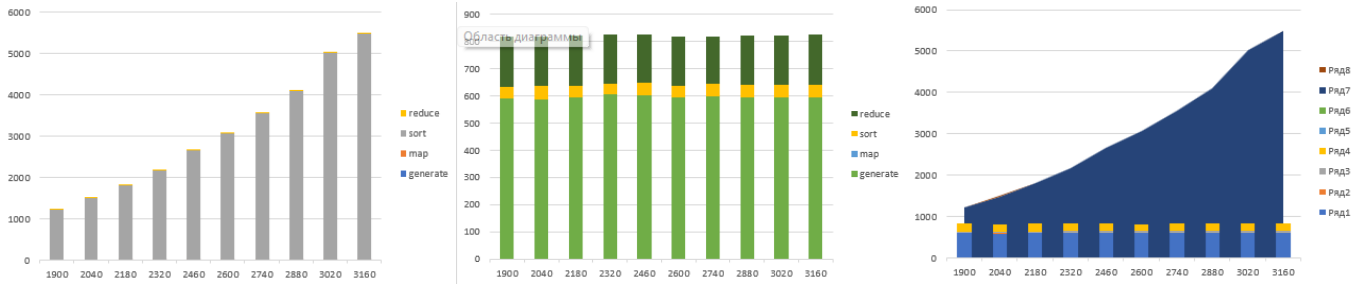


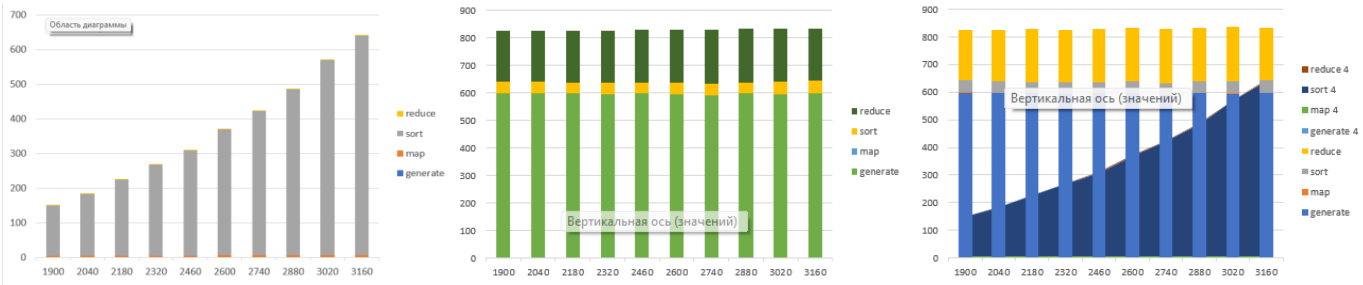
График параллельного ускорения при использовании OpenCL имеет такой сильный прирост в зависимости от количества элементов, так как для однопоточной реализации суммарное время выполнения очень сильно зависит от количества элементов и занимает существенное время. Тогда как при использовании OpenCL основное время занимают накладные расходы на инициирование kernel функций и копирование массивов в память видеокарты. При этом время, затрачиваемое на вычисления сравнимо мало, а накладные расходы не зависят от количества элементов.



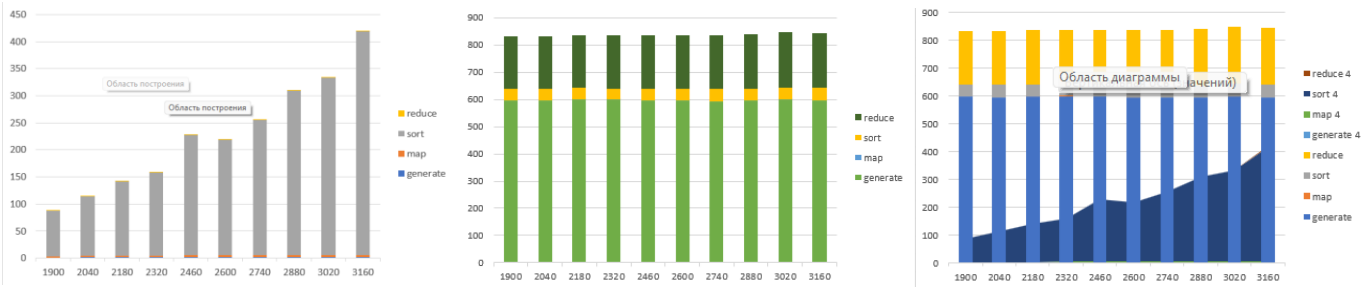
Сравнение времени выполнения на разных участках программы



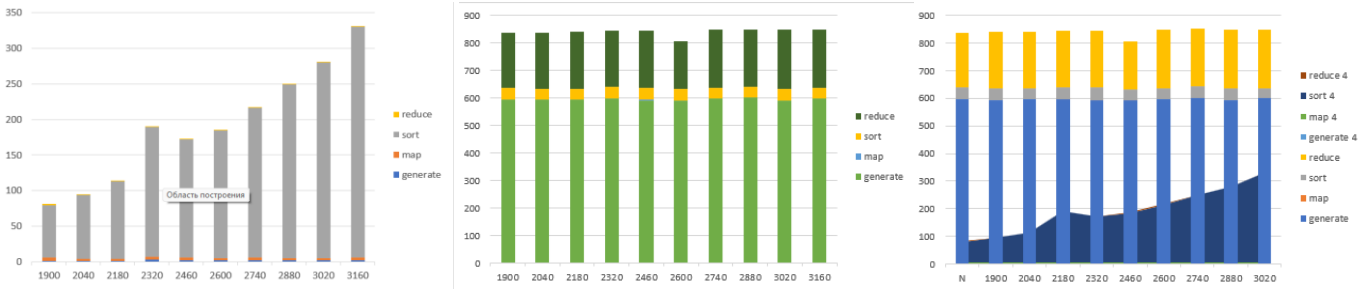
1 поток



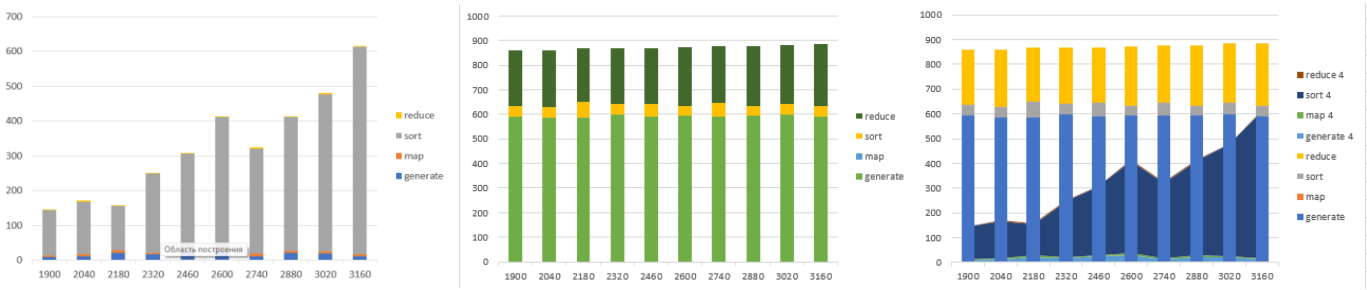
2 потока



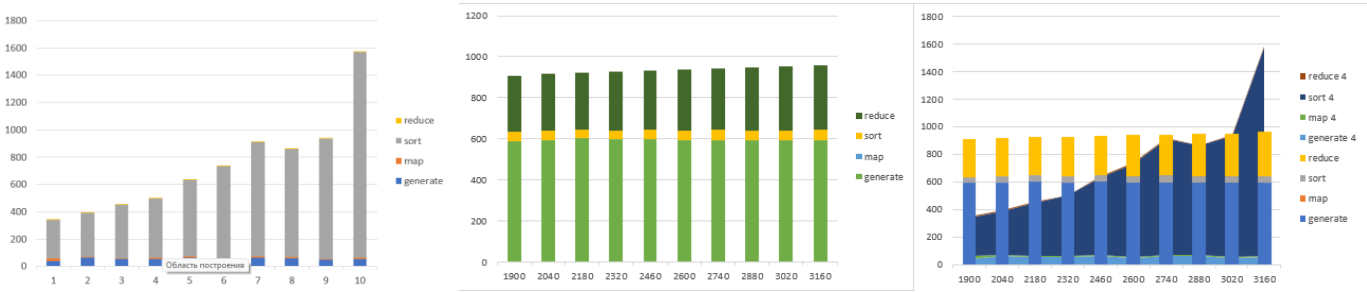
3 потока



4 потока



8 пот.



16 пот.

При сравнении времени, затраченного на разных этапах, можно заметить огромную разницу между использованием OpenCL и OpenMP. В случае с OpenCL время выполнения программы практически не зависит от количества элементом и количества блоков, на которые бьётся сортировка. Наибольшее время занимают накладные расходы использования вычислений на видеокарте.

Сравнение по остальным параметрам

- количество строк кода, добавленных при распараллеливании
  - OpenMP – около 150
  - POSIX – около 250
  - POSIX – около 400
- накладные расходы программиста
  - OpenMP – 1 рабочий день
  - POSIX – 1.5 рабочего дня
  - OpenCL – 2 рабочих дня
- Максимальная вложенность кода
  - OpenMP – 6 уровней
  - POSIX – 3 уровня
  - OpenCL – 3 уровня
- Сложность реализации дополнительного потока, работающего параллельно остальной части программы
  - OpenMP – средняя\высокая (неудобная реализация и управление)
  - POSIX – тривиально (запуск потоков не отличается от того, как они будут работать)

Листинг main.c

```
#define CL_TARGET_OPENCL_VERSION 120
#define CL_USE_DEPRECATED_OPENCL_1_2_APIS
#include <CL/cl.h>
#pragma warning (disable : 4996)
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <math.h>
#include <sys/timeb.h>

// #define DEBUG 1
#define BENCHMARK 1
#define SOURCE_NAME "compute.cl"


void print_err_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 print_err(
  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 );
        print_err_code(err);
        if (*ctx) {
            clReleaseContext(*ctx);
        }
        exit(1);
    }
}


void print_arr(double *array, int n) {
    #ifdef DEBUG
        for (int i = 0; i < n; ++i) {
            printf("%f ", array[i]);
        }
        printf("\n");
    #endif
}


void print_buffer(
    cl_program *program,
    cl_command_queue *queue,
    cl_mem *dst,
    int n,
    const char * buffer_name
) {
    #ifdef DEBUG
        double * dst_host = malloc(n * sizeof(double));
        clEnqueueReadBuffer(*queue, *dst, CL_TRUE, 0, n * sizeof(cl_double), dst_host, 0,
NULL, NULL);
        if (buffer_name) printf("%s ", buffer_name);
        for (int i = 0; i < n; i++) printf("%f ", dst_host[i]);
        printf("\n");
        free(dst_host);
    #endif
}
```

```
double get_time() {
    struct timeb result;
    ftime(&result);
    return 1000.0 * result.time + result.millitm;
}

// --------------- PRINTS END

void run_kernel(
    const char * kernel_name,
    cl_kernel kernel,
    cl_context *ctx,
    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);
    print_err(ctx, &err, "clSetKernelArg()", kernel_name);

    size_t global_work_size = n;
    err = clEnqueueNDRangeKernel(*queue, kernel, 1, NULL, &global_work_size, NULL, 0, NULL,
NULL);
    print_err(ctx, &err, "clEnqueueNDRangeKernel()", kernel_name);
}


// --------------- SORT & REDUCE

void init_chunked_args(
    cl_context *ctx,
    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) {
        int n_cur_chunk = max(min((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(*ctx, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sort_parts *
sizeof(cl_int), src_offset_host, &err );
    print_err(ctx, &err, "sort src_offset clCreateBuffer()", NULL);
    *src_size = clCreateBuffer(*ctx, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sort_parts *
sizeof(cl_int), src_size_host, &err );
    print_err(ctx, &err, "sort src_size clCreateBuffer()", NULL);
}


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);
        print_err(ctx, &err, "sort temp -> src clEnqueueCopyBuffer()", NULL);
    }
}


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 = malloc(n * sizeof(int));
    int * src_size_host = 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);
    print_err(ctx, &err, "sort src -> temp clEnqueueCopyBuffer()", 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 = malloc(n * sizeof(int));
    int * src_size_host = 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);
    print_err(ctx, &err, "reduce_sum dst clCreateBuffer()", NULL);

    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 = 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 * restrict m1_host,
    double * restrict m2_host,
    int n1,
    int n2,
    int i
) {
    const int A = 280;
    srand(i);

    for (int j = 0; j < n1; ++j) {
        m1_host[j] = (rand() % (A * 100)) / 100.0 + 1;
    }
    for (int j = 0; j < n2; ++j) {
        m2_host[j] = A + rand() % (A * 9);
    }
}

// --------------- BENCHMARK

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
}

// --------------- INIT METHODS

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 };

    err = clGetPlatformIDs(1, &platform, NULL);
    print_err(ctx, &err, "clGetPlatformIDs()", NULL);

    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    print_err(ctx, &err, "clGetDeviceIDs()", NULL);

    props[1] = (cl_context_properties)platform;
    *ctx = clCreateContext(props, 1, &device, NULL, NULL, &err);
    print_err(ctx, &err, "clCreateContext()", NULL);

    *queue = clCreateCommandQueue(*ctx, device, 0, &err);
    print_err(ctx, &err, "clCreateCommandQueue()", NULL);

    // Perform runtime source compilation, and obtain kernel entry point.
    *program = clCreateProgramWithSource(*ctx, 1, source, NULL, &err );
    clBuildProgram( *program, 1, &device, NULL, NULL, NULL );
    if (err != CL_SUCCESS) {
        size_t len;
        char buffer[2048];
        clGetProgramBuildInfo(
            *program, device, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len
        );
        printf("clBuildProgram() failed with %s\n", buffer);
    }
}


void init_buffers(
    cl_context * ctx,
```

```
    double * m1_host,
    double * m2_host,
    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_host, &err );
    print_err(ctx, &err, "M1 clCreateBuffer()", NULL);
    *m2 = clCreateBuffer(*ctx, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, n2 *
sizeof(cl_double), m2_host, &err );
    print_err(ctx, &err, "M2 clCreateBuffer()", NULL);
    *m2_cpy = clCreateBuffer(*ctx, CL_MEM_READ_WRITE , n2 * sizeof(cl_double), NULL, &err );
    print_err(ctx, &err, "M2_CPY clCreateBuffer()", NULL);
}


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, "ctanh_sqrt", &err);
    print_err(ctx, &err, "[ctanh_sqrt] clCreateKernel", NULL);
    *sum_prev = clCreateKernel(*program, "sum_prev", &err);
    print_err(ctx, &err, "[sum_prev] clCreateKernel", NULL);
    *pow_log10 = clCreateKernel(*program, "pow_log10", &err);
    print_err(ctx, &err, "[pow_log10] clCreateKernel", NULL);
    *max_2_src = clCreateKernel(*program, "max_2_src", &err);
    print_err(ctx, &err, "[max_2_src] clCreateKernel", NULL);
    *map_sin = clCreateKernel(*program, "map_sin", &err);
    print_err(ctx, &err, "[map_sin] clCreateKernel", NULL);

    *sort_kernel = clCreateKernel(*program, "sort", &err);
    print_err(ctx, &err, "[sort] clCreateKernel", NULL);
    *merge_sort_kernel = clCreateKernel(*program, "merge_sorted", &err);
    print_err(ctx, &err, "[merge_sorted] clCreateKernel", NULL);
    *reduce_sum_kernel = clCreateKernel(*program, "reduce_sum", &err);
    print_err(ctx, &err, "[reduce_sum] clCreateKernel", NULL);
}

// ---------------

int main(int argc, char ** argv) {
```

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

/* Read source to char buffer */
FILE *fp;
long lSize;

fp = fopen(SOURCE_NAME, "rb");

fseek( fp , 0L , SEEK_END);
lSize = ftell(fp);
rewind(fp);

/* allocate memory for entire content */
const char * source = calloc(1, lSize + 1);
if( !source ) fclose(fp), fputs("memory alloc fails", stderr), exit(1);

/* copy the file into the source */
if( 1 != fread((void *)source, lSize, 1, fp) ) {
    fclose(fp), free((void *)source), fputs("entire read fails", stderr), exit(1);
}
fclose(fp);

cl_int err;
cl_context ctx = 0;
cl_command_queue queue = 0;
cl_program program = NULL;

init_opencl_env(&ctx, &queue, &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 * restrict m1_host = malloc(N * sizeof(double));
double * restrict m2_host = malloc(N_2 * sizeof(double));
cl_mem m1, m2, m2_cpy;
init_buffers(&ctx, m1_host, m2_host, &m1, &m2, &m2_cpy, N, N_2);

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

init_kernels(
    &ctx, &program, &ctanh_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_host, m2_host, N, N_2, i);
```

```
        print_arr(m1_host, N);
        print_arr(m2_host, N_2);

        err = clEnqueueWriteBuffer(queue, m1, CL_TRUE, 0, N_2 * sizeof(cl_double), m1_host,
0, NULL, NULL);
        err |= clEnqueueWriteBuffer(queue, m2, CL_TRUE, 0, N_2 * sizeof(cl_double), m2_host,
0, NULL, NULL);
        err |= clEnqueueCopyBuffer(queue, m2, m2_cpy, 0, 0, N_2 * sizeof(cl_double), 0,
NULL, NULL);
        print_err(&ctx, &err, "m1, m2, m2_cpy clEnqueueWriteBuffer, clEnqueueCopyBuffer()",
NULL);
        finish_benchmark(benchmarking_time, benchmarking_results, 0);

        // map
        start_benchmark(benchmarking_time, 1);
        run_kernel(
            "ctanh_sqrt", ctanh_sqrt, &ctx ,&program, &queue, N, 2,
            sizeof(cl_mem *), &m1, sizeof(cl_mem *), &m1
        );
        run_kernel(
            "sum_prev", sum_prev, &ctx ,&program, &queue, N_2, 3,
            sizeof(cl_mem *), &m2, sizeof(cl_mem *), &m2_cpy, sizeof(cl_mem *), &m2
        );
        run_kernel(
            "pow_log10", pow_log10, &ctx ,&program, &queue, N_2, 2,
            sizeof(cl_mem *), &m2, sizeof(cl_mem *), &m2
        );
        run_kernel(
            "max_2_src", max_2_src, &ctx ,&program, &queue, 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(&ctx, &program, &queue, 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(queue, m2_cpy, CL_TRUE, 0, N_2 * sizeof(cl_double), m2_host, 0,
NULL, NULL);
        int k = 0;
        while (m2_host[k] == 0 && k < N_2 - 1) k++;
        cl_double m2_min = m2_host[k];

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

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

```
    clFinish( queue );
    free(m1_host);
    free(m2_host);
    show_benchmark_results(benchmarking_time, benchmarking_results, N_benchmarks);

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

Compute.cl

```
kernel void memset(
    global double *dst
) {
    dst[get_global_id(0)] = get_global_id(0) * 2;
}

kernel void ctanh_sqrt(
    global double *src,
    global double *dst
) {
    int i = get_global_id(0);
    dst[i] = 1 / tanh(sqrt(src[i]));
}

kernel void sum_prev(
    global double *src1,
    global double *src2,
    global double *dst
) {
    int i = get_global_id(0);
    dst[i] = i > 0 ? src1[i] + src2[i - 1] : src1[i];
}

kernel void pow_log10(
    global double *src,
    global double *dst
) {
    int i = get_global_id(0);
    dst[i] = pow(log10(src[i]), M_E);
}

kernel void max_2_src(
    global double *src1,
    global double *src2,
    global double *dst
) {
    int i = get_global_id(0);
    dst[i] = max(src1[i], src2[i]);
}
```

```
kernel void map_sin(
    global double *src,
    global double *dst,
    double min_v
) {
    int i = get_global_id(0);
    if( (int)(src[i] / min_v) % 2 == 0 ) {
        dst[i] = sin(src[i]);
    } else {
        dst[i] = 0;
    }
}

kernel void reduce_sum(
    global int *src_offset,
    global int *src_size,
    global double *src,
    global double *dst
) {
    int k = get_global_id(0);
    dst[k] = 0;
    for (int i = 0; i < src_size[k]; i++) {
        dst[k] += src[src_offset[k] + i];
    }
}

kernel void sort(
    global double *src_offset,
    global double *src_size,
    global double *src
) {
    int j = get_global_id(0);
    int offset = src_offset[j];
    int i = 0;
    while (i < src_size[i] - 1) {
        if (src[offset + i + 1] < src[offset + i]) {
            double t = src[offset + i + 1];
            src[offset + i + 1] = src[offset + i];
            src[offset + i] = t;
            i = 0;
        } else {
            i++;
        }
    }
}

kernel void merge_sorted(
    global double *src,
    global double *dst,
    int offset_1,
    int offset_2,
    int offset_dst,
    int n_src_1,
    int n_src_2
```

```
) {
    int i1 = offset_1;
    int i2 = offset_2;
    int i = offset_dst;
    while (i < n_src_1 + n_src_2) {
        dst[i++] = src[i1] > src[i2] && i2 < n_src_2 + offset_2 ? src[i2++] : src[i1++];
    }
}
```

Вывод

В процессе реализации был разработан код, который запускает вычисления на видеокарте с помощью OpenCL. Процесс вычислений в данном случае крайне быстр и эффективен, однако гораздо больше затрат на накладные расходы. В том числе итоговая программа получается сильно сложнее. Я считаю, что верным решениям вычисления на видеокарте являются в случае необходимости постоянных вычислений на больших массивах данных. На относительно небольших массивах, тестирующихся в лабораторной результаты сравнимы с OpenMP и суммарное время выполнения в основном уступает. Однако на кратно массивах прирост OpenCL очевиден.