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: Р4116

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)

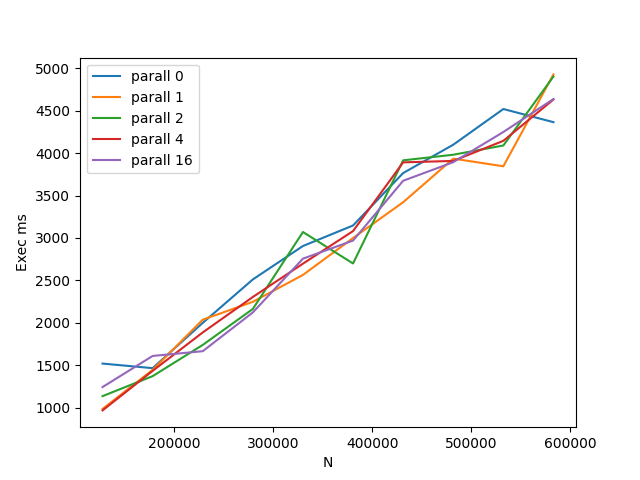
[01]: 192.168.56.1

[02]: fe80::527e:5766:393d:acc6

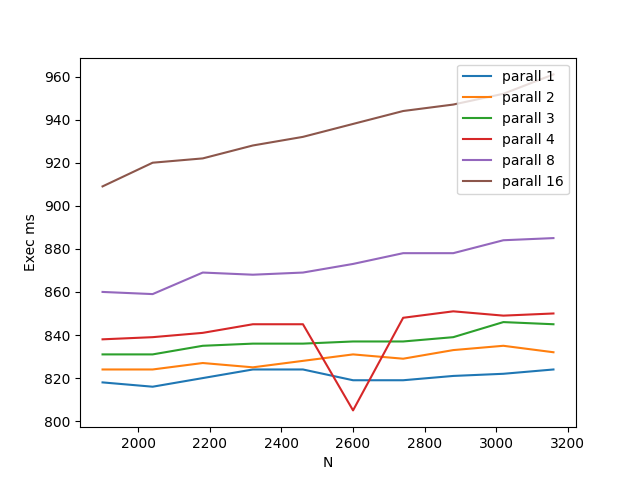
Hyper-V Requirements: A hypervisor has been detected. Features required for Hyper-V will not be displayed.

## Результаты работы

**CLANG (автоматизированное распараллеливание)**

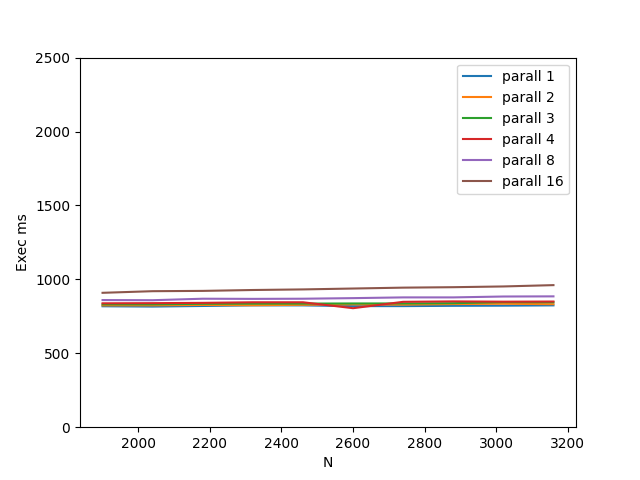
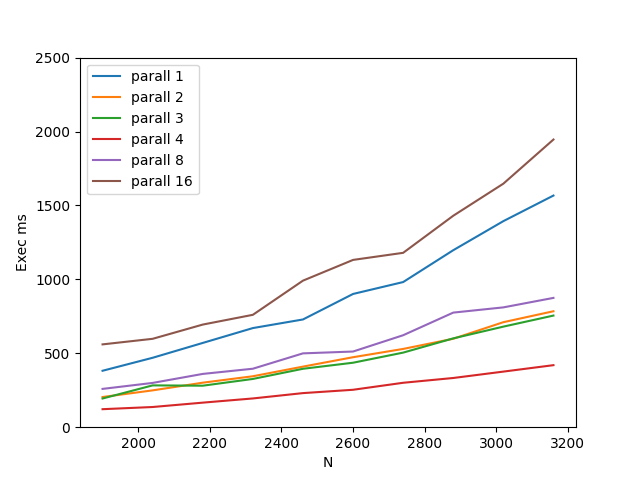


**OpenCL**

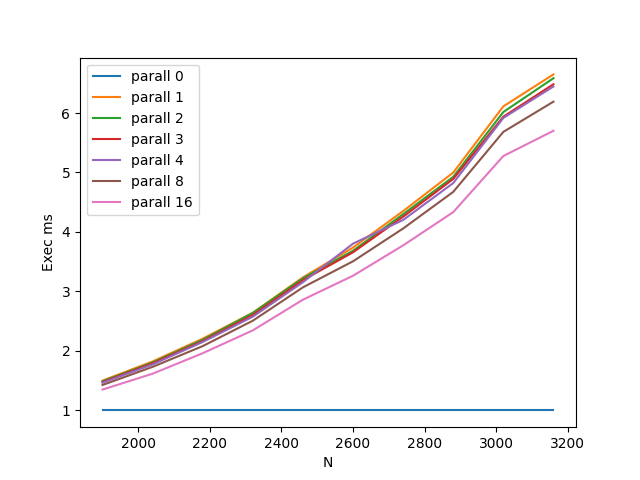
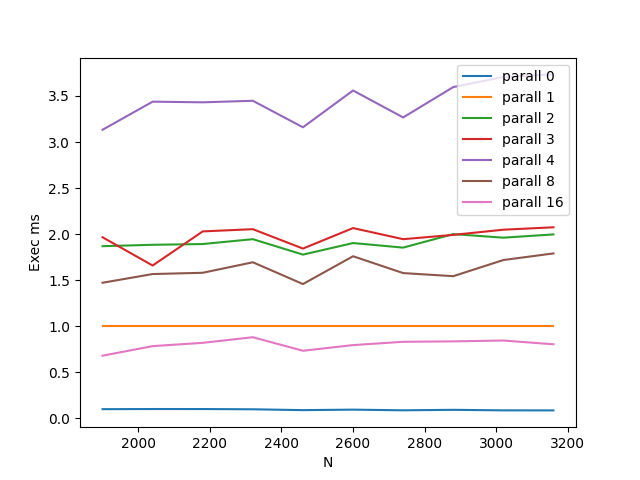


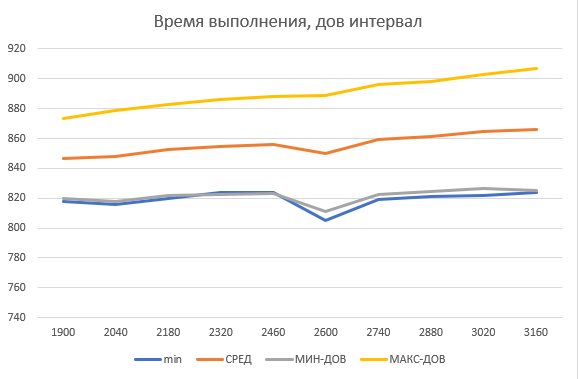
**OpenMP OpenCL**

Время выполнения

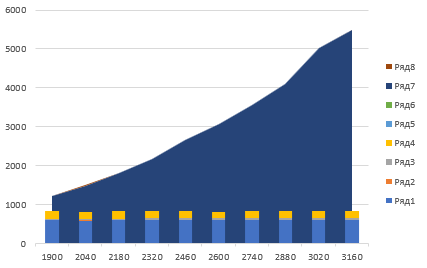
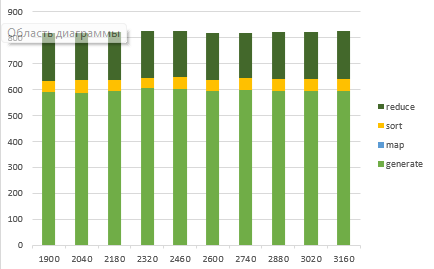
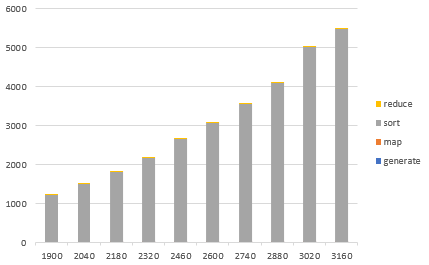
****

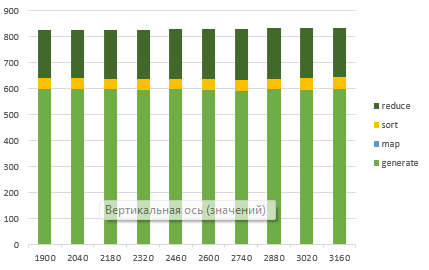
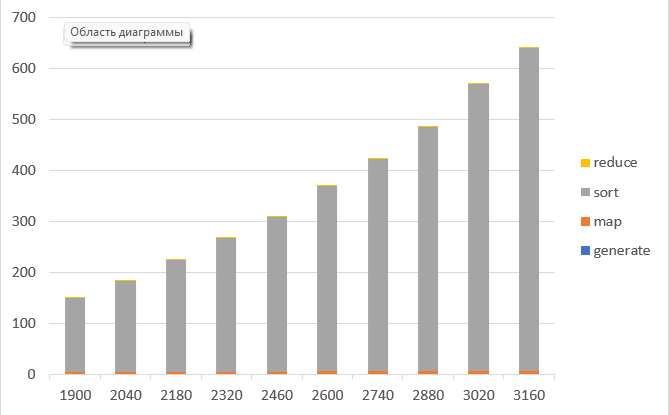
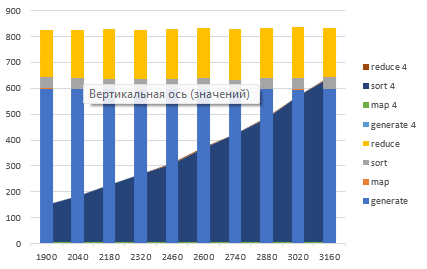
Параллельное ускорение

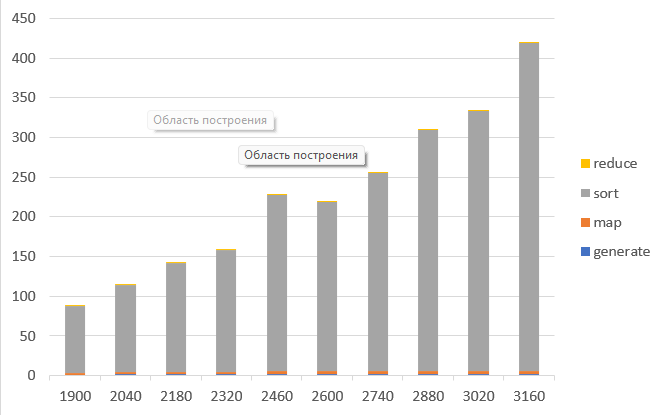
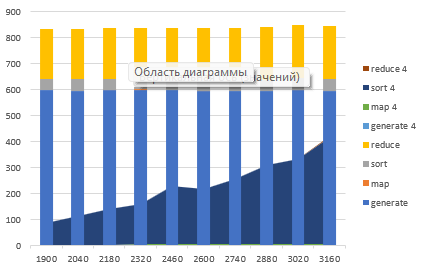
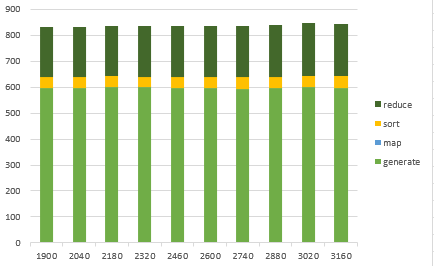


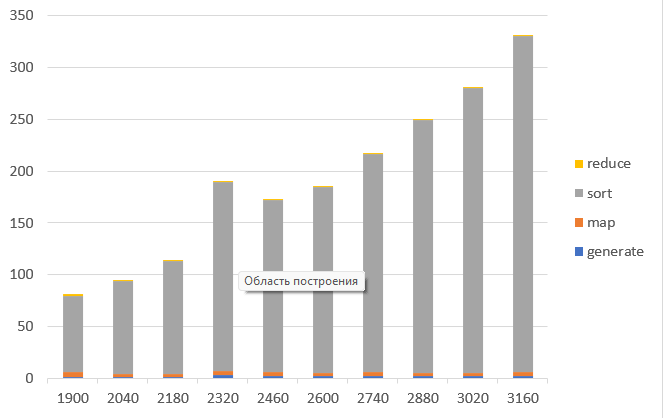
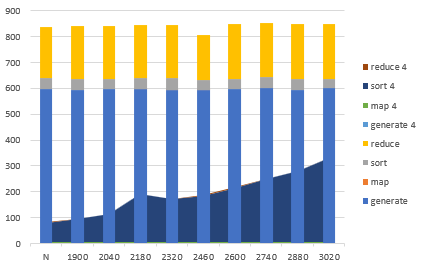
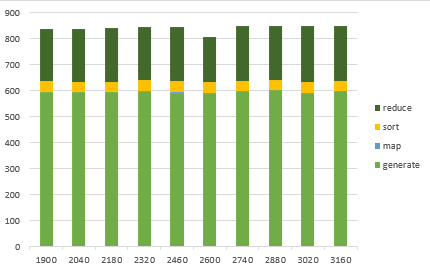
****

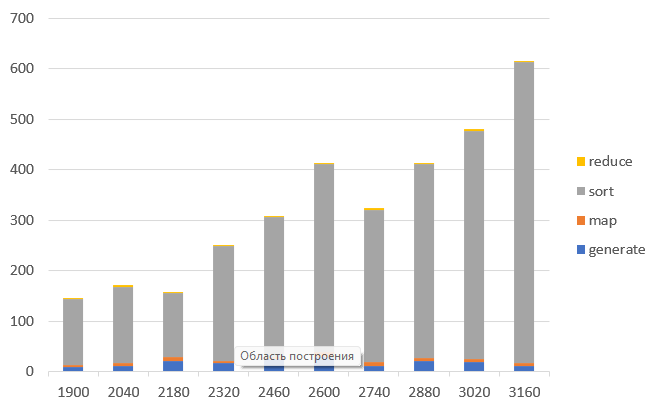
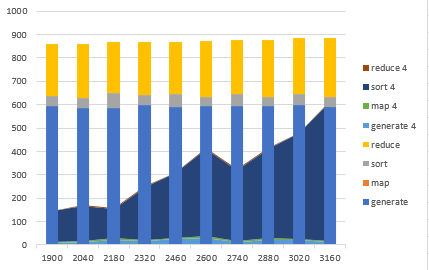
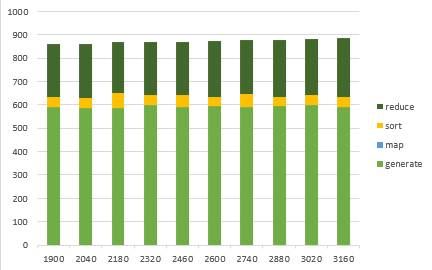
**Сравнение времени выполнения на разных участках программы**

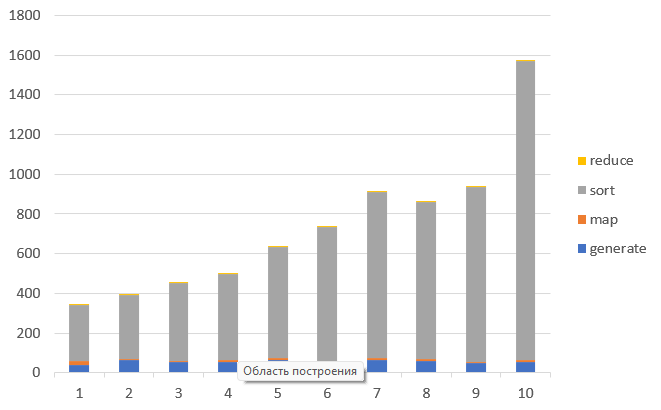
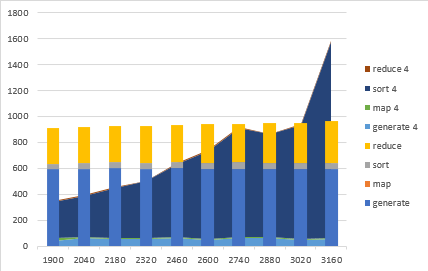
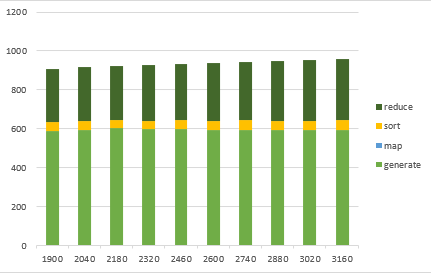
**1 поток**

** **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 – 2 рабочих дня
* 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.