САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: OpenCL

Отчет

по лабораторной работе №2

**«OpenCL. Умножение матриц»**

Выполнил: Акимов Николай Сергеевич

Номер ИСУ: 264328

студ. гр. M33381

Санкт-Петербург

2021

**Цель работы:** реализовать эффективную программу, выполняющую умножение матриц с использованием OpenCL (версия 1.2).

**Инструментарий и требования к работе:**стандартные библиотеки C++ и библиотека OpenCL.

**Практическая часть**

Для начала реализовал обычное умножение матриц без использования локальной памяти. Также было протестировано четыре варианта расположение умножаемых матриц: не транспонировать вообще, транспонировать только первую, транспонировать только вторую, транспонировать обе. В итоге по результатам тестирования был выбран второй вариант, т. е. транспонируем только первую умножаемую матрицу. Одна из возможных причин – потоки, работающие одновременные, чаще всего имеют одинаковую y координату, связи с этим на каждой итерации цикла они обращаются в одну и ту же ячейку памяти первой матрицы. Насчёт второй матрицы, то у одновременно работающих потоков x координаты находятся рядом, в связи с чем при обращении ко второй матрицы одновременно обращаются к разным банкам памяти, из-за этого суммарное время обращение к глобальной памяти минимально.

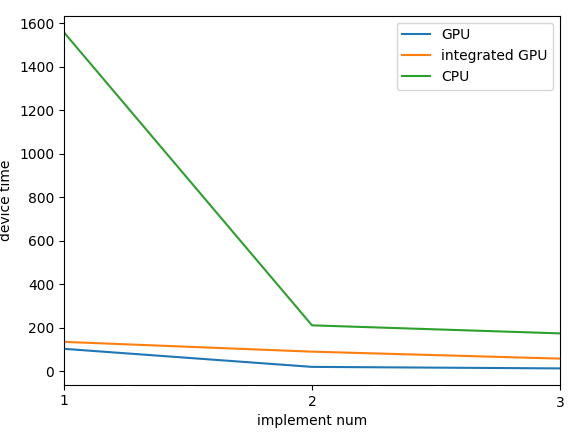
Дальше по задании необходимо было реализовать умножение матриц с локальной памятью. Для возможности экспериментировать с размером локальных матриц была заведена константа, а весь device-код опирается на эту константу. В итоге для тестирования необходимо было только менять одну константу. По результатам тестирования было выяснено, что размер локальной матрицы 16x16 быстрее всего работает на матрицах большого размера, при этом сделать матрицу больше нельзя из-за ограничения некоторых устройств на максимальный размер локальной группы в 256.

И последняя модификация – использовать вектора для увеличения полезной работы каждого потока. После тестирования на сервере было выявлено, что максимальный размер вектор составляет два, при размере четыре алгоритм уже отрабатывает медленнее. Вероятно, это связано, что при большом размере увеличивается размер локальной переменной, в результате чего уменьшается общее число запущенных одновременно потоков, из-за чего и падает время работы.

Технические характеристики:

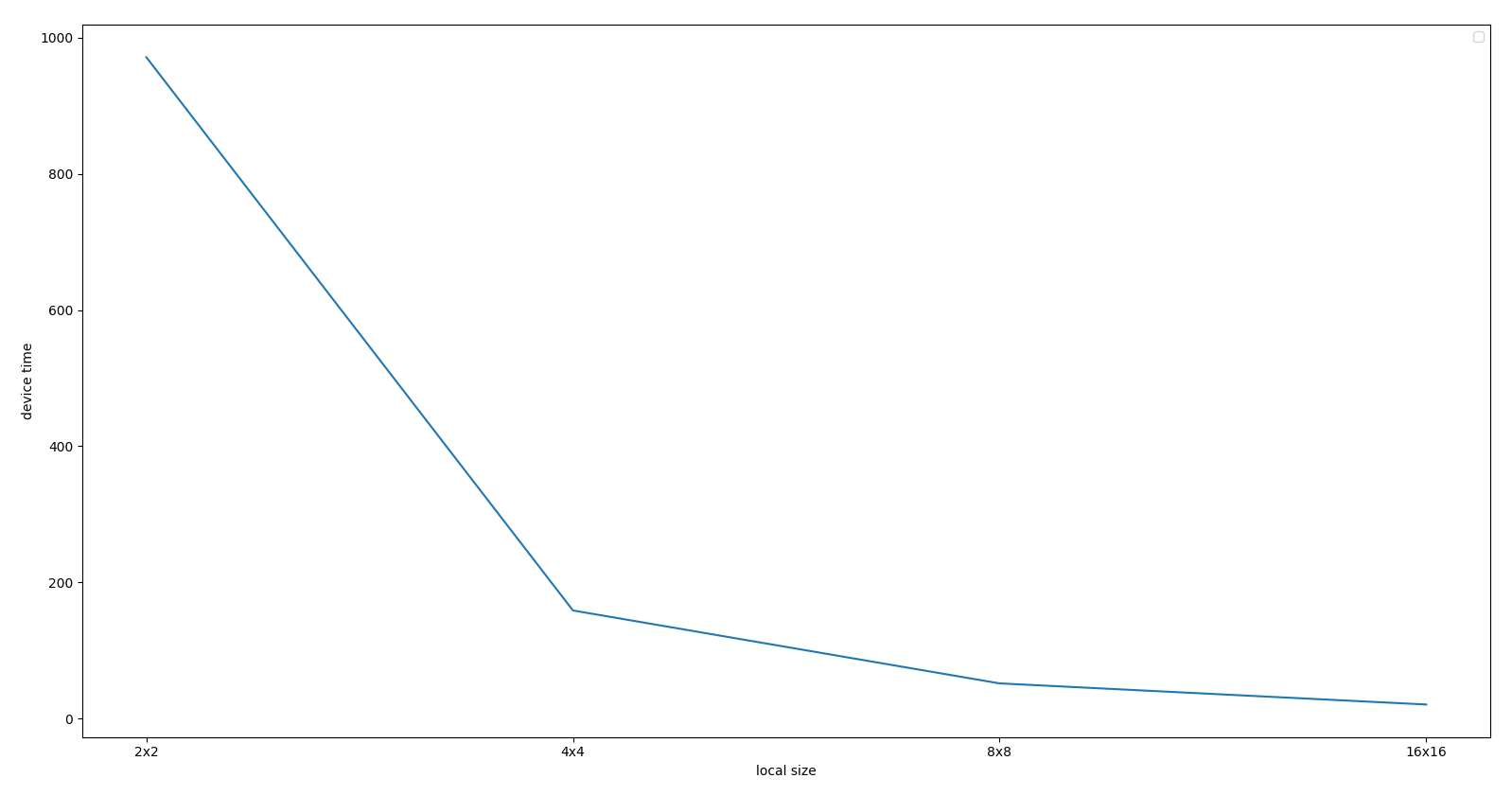
* CPU: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
* Integrated GPU: Intel(R) HD Graphics 520
* GPU: GeForce 940MX

Ниже представлены графики времени работы алгоритма в зависимости от выбранного устройства, выбранных размера матриц и выбранных параметров алгоритма.



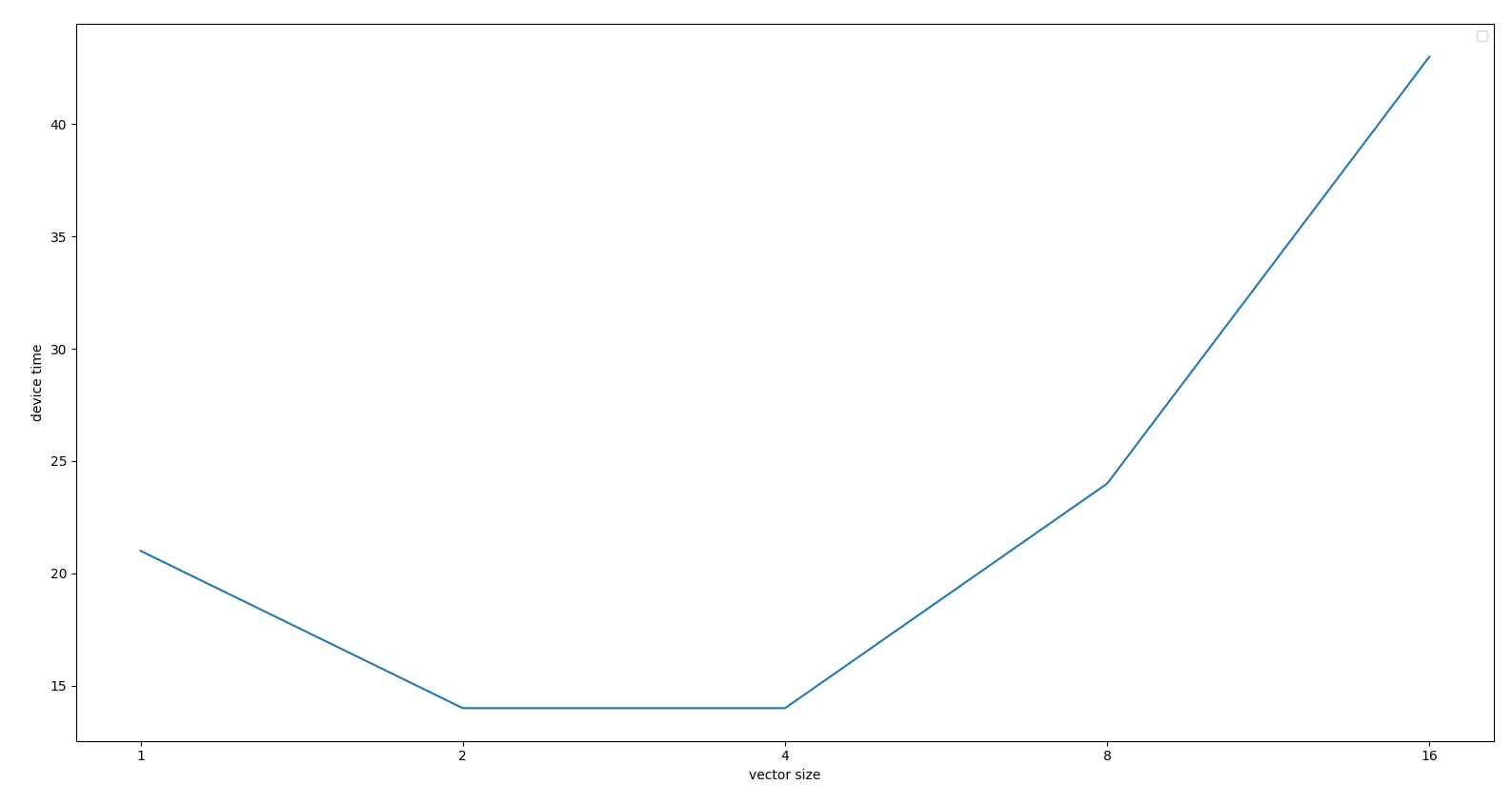
Сравнение времени работы (в мс) на матрицах размером n = 1000, m = 1000, k = 1000

Результат вполне предсказуемый, поскольку на дискретной видеокарте больше запускаемых потоков, поэтому при больших данных (порядка 1 000 000 000) суммарное время работы меньше остальных устройств.



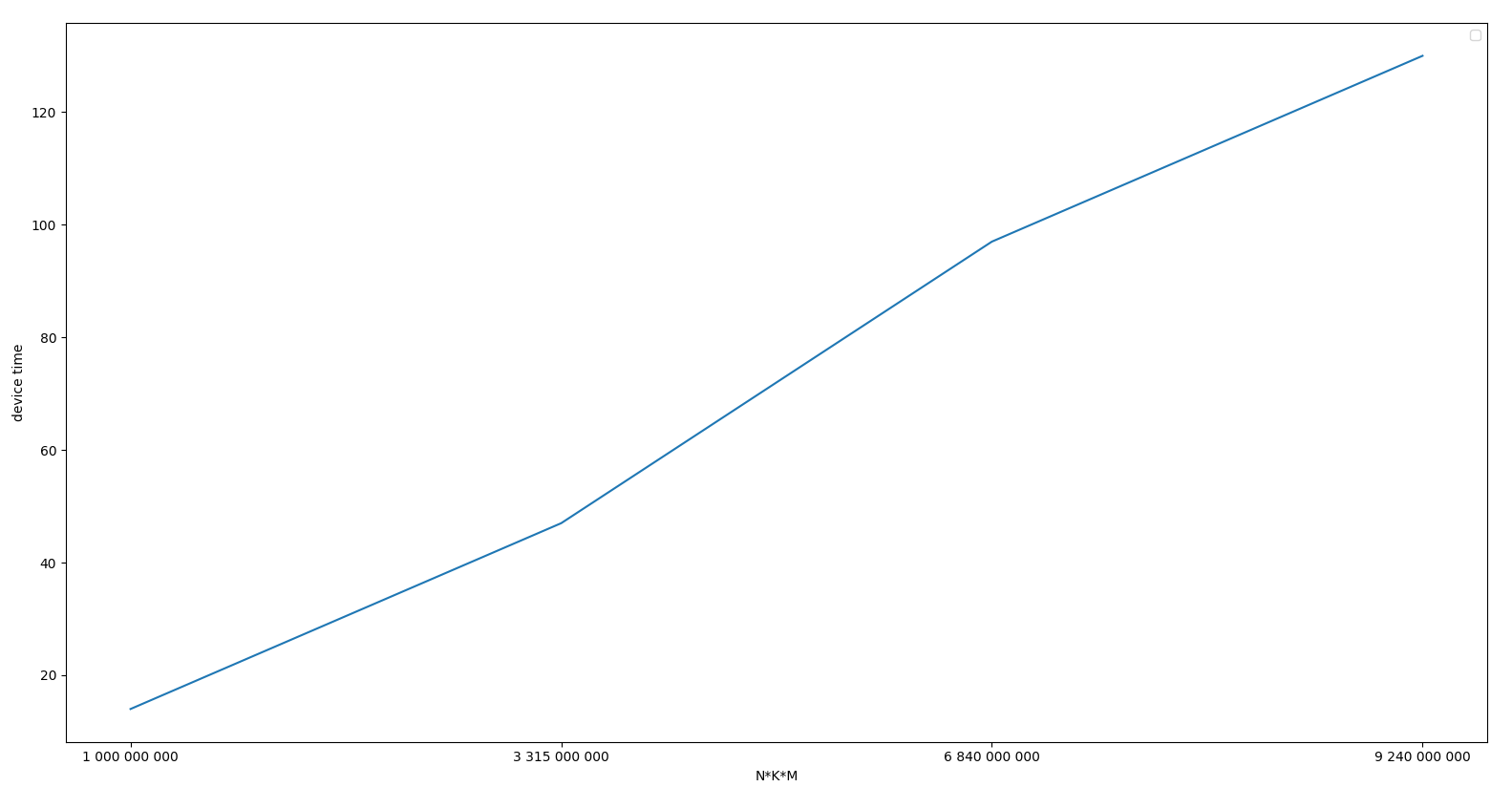
Время работы GPU (в мс) в зависимости от размера локальной группы на матрицах размером n = 1000, m = 1000, k = 1000

Из-за ограничения OpenCL в размере локальной группы в 256 пришлось ограничить максимальный размер локальной группы в 16x16. Результат очевиден, при большем размере локальной группы получаем большую скорость.



Время работы GPU (в мс) в зависимости от размера вектора при размере локальной группы 16x16 на матрицах размером n = 1000, m = 1000, k = 1000

Если один поток будет обрабатывать по несколько элементов результирующей матрицы, то полезная работа одного потока возрастёт, а накладные расходы будут составлять меньшее число процентов от общей работы потока. Но при сильно большом значении количества обрабатываемых элементов получаем, что каждый поток делает большее количество обращений к памяти, при этом общее число одновременно работающих потоков уменьшается, из-за чего получаем проигрыш по суммарному времени работы.



Время работы GPU (в мс) при размере локальной группы 16x16, при размере вектора 2, на матрицах различного размера. По оси X указано произведение размерностей матрицы.

По графику видно, что время работы возрастает линейно относительно произведения размерностей матрицы, что вполне себе предсказуемо.

**main.cpp**

#include <iostream>

#include <fstream>

#include <exception>

#include <string>

#include <vector>

#include <algorithm>

#include <chrono>

#include "opencl\_utils.h"

static uint64\_t get\_valid\_uint\_number(char const\* s) {

if (s[0] == '-') throw std::runtime\_error("");

size\_t index;

uint64\_t ans = std::stoull(s, &index);

if (s[index] != '\0') throw std::runtime\_error("");

return ans;

}

static float get\_valid\_float(char const\* s) {

size\_t index;

float ans = std::stof(s, &index);

if (s[index] != '\0') throw std::runtime\_error("");

return ans;

}

static void read\_file(size\_t& n, size\_t& k, size\_t& m, std::vector<float>& first\_matrix, std::vector<float>& second\_matrix, char const\* input\_file) {

std::ifstream input(input\_file);

if (input.fail()) throw std::runtime\_error("Could not open input file");

try {

{

std::string input\_str;

std::getline(input, input\_str);

std::string num;

size\_t index = 0;

while (index < input\_str.length() && input\_str[index] != ' ') num += input\_str[index++];

if (index == input\_str.length()) throw std::runtime\_error("");

n = get\_valid\_uint\_number(num.c\_str());

num = "";

index++;

while (index < input\_str.length() && input\_str[index] != ' ') num += input\_str[index++];

if (index == input\_str.length()) throw std::runtime\_error("");

k = get\_valid\_uint\_number(num.c\_str());

num = "";

index++;

while (index < input\_str.length()) num += input\_str[index++];

m = get\_valid\_uint\_number(num.c\_str());

}

if (n == 0 || k == 0 || m == 0) throw std::runtime\_error("");

for (size\_t i = 0; i < m; i++) {

std::string input\_str;

std::getline(input, input\_str);

std::string num;

size\_t index = 0;

for (size\_t j = 0; j < k - 1; j++) {

while (index < input\_str.length() && input\_str[index] != ' ') num += input\_str[index++];

if (index == input\_str.length()) throw std::runtime\_error("");

first\_matrix.push\_back(get\_valid\_float(num.c\_str()));

num = "";

index++;

}

while (index < input\_str.length()) num += input\_str[index++];

first\_matrix.push\_back(get\_valid\_float(num.c\_str()));

}

for (size\_t i = 0; i < k; i++) {

std::string input\_str;

std::getline(input, input\_str);

std::string num;

size\_t index = 0;

for (size\_t j = 0; j < n - 1; j++) {

while (index < input\_str.length() && input\_str[index] != ' ') num += input\_str[index++];

if (index == input\_str.length()) throw std::runtime\_error("");

second\_matrix.push\_back(get\_valid\_float(num.c\_str()));

num = "";

index++;

}

while (index < input\_str.length()) num += input\_str[index++];

second\_matrix.push\_back(get\_valid\_float(num.c\_str()));

}

std::string input\_str;

std::getline(input, input\_str);

if (input\_str != "") throw std::runtime\_error("");

} catch (...) {

throw std::runtime\_error("Incorrect input file");

}

}

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

if (argc != 5) {

std::cerr << "Input format: <device's number> <input file> <output file> <implementation's number>" << std::endl;

return 1;

}

uint32\_t device\_num;

try {

device\_num = get\_valid\_uint\_number(argv[1]);

} catch (...) {

std::cerr << "device's number should be the not negative integer" << std::endl;

return 1;

}

if (argv[4][1] != '\0' || argv[4][0] < '1' || argv[4][0] > '3') {

std::cerr << "implementation's number should be from 1 to 3" << std::endl;

return 1;

}

size\_t n, k, m;

std::vector<float> first\_matrix;

std::vector<float> second\_matrix;

try {

read\_file(n, k, m, first\_matrix, second\_matrix, argv[2]);

std::vector<float> traspose\_first\_matrix(m \* k);

for (size\_t i = 0; i < m; i++) {

for (size\_t j = 0; j < k; j++) {

traspose\_first\_matrix[j \* m + i] = first\_matrix[i \* k + j];

}

}

first\_matrix = traspose\_first\_matrix;

multiply(device\_num, argv[4][0], argv[3], n, k, m, first\_matrix, second\_matrix);

} catch (std::exception &e) {

std::cerr << e.what() << std::endl;

return 1;

}

return 0;

}

**opencl\_utils.h**

заголовочный файл для операций на OpenCL

#include <vector>

void multiply(uint32\_t device\_num, char implement\_num, char const\* output\_file, size\_t const& N, size\_t const& K, size\_t const& M, std::vector<float>& first\_matrix, std::vector<float>& second\_matrix);

**opencl\_utils.cpp**

реализация операция на OpenCL

#include "opencl\_utils.h"

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include <chrono>

#include <algorithm>

#include <CL/opencl.h>

size\_t const LOCAL\_SIZE = 16;

size\_t const VECTOR\_SIZE = 2;

void multiply(uint32\_t device\_num, char implement\_num, char const\* output\_file, size\_t const& N, size\_t const& K, size\_t const& M, std::vector<float>& first\_matrix, std::vector<float>& second\_matrix) {

auto t\_start = std::chrono::high\_resolution\_clock::now();

size\_t NN = (N % LOCAL\_SIZE == 0 ? N : (N / LOCAL\_SIZE + 1) \* LOCAL\_SIZE);

size\_t MM = (M % LOCAL\_SIZE == 0 ? M : (M / LOCAL\_SIZE + 1) \* LOCAL\_SIZE);

size\_t KK = (K % LOCAL\_SIZE == 0 ? K : (K / LOCAL\_SIZE + 1) \* LOCAL\_SIZE);

if (implement\_num == '3') {

NN = (NN % (LOCAL\_SIZE \* VECTOR\_SIZE) == 0 ? NN : (NN / (LOCAL\_SIZE \* VECTOR\_SIZE) + 1) \* (LOCAL\_SIZE \* VECTOR\_SIZE));

}

std::vector<float> new\_first\_matrix(KK \* MM);

for (size\_t i = 0; i < KK; i++) {

for (size\_t j = 0; j < MM; j++) {

if (i < K && j < M) {

new\_first\_matrix[i \* MM + j] = first\_matrix[i \* M + j];

} else {

new\_first\_matrix[i \* MM + j] = 0;

}

}

}

std::vector<float> new\_second\_matrix(KK \* NN);

for (size\_t i = 0; i < KK; i++) {

for (size\_t j = 0; j < NN; j++) {

if (i < K && j < N) {

new\_second\_matrix[i \* NN + j] = second\_matrix[i \* N + j];

}

else {

new\_second\_matrix[i \* NN + j] = 0;

}

}

}

cl\_uint n;

cl\_int error;

cl\_device\_id device = nullptr;

error = clGetPlatformIDs(0, nullptr, &n);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get platform ids");

std::unique\_ptr<cl\_platform\_id[]> platforms;

try {

platforms = std::unique\_ptr<cl\_platform\_id[]>(new cl\_platform\_id[n]);

} catch (...) {

throw std::runtime\_error("Could not allocate memory");

}

error = clGetPlatformIDs(n, platforms.get(), nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get platfrom ids");

uint32\_t num\_of\_devices = 0;

for (auto i = 0; i < n; i++) {

cl\_uint m;

error = clGetDeviceIDs(platforms[i], CL\_DEVICE\_TYPE\_ALL, 0, nullptr, &m);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device ids");

num\_of\_devices += m;

}

if (device\_num >= num\_of\_devices) device\_num = 0;

for (auto i = 0; i < n; i++) {

cl\_uint m;

error = clGetDeviceIDs(platforms[i], CL\_DEVICE\_TYPE\_GPU, 0, nullptr, &m);

if (error == CL\_DEVICE\_NOT\_FOUND) continue;

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device ids");

std::unique\_ptr<cl\_device\_id[]> devices;

try {

devices = std::unique\_ptr<cl\_device\_id[]>(new cl\_device\_id[m]);

} catch (...) {

throw std::runtime\_error("Could not allocate memory");

}

error = clGetDeviceIDs(platforms[i], CL\_DEVICE\_TYPE\_GPU, m, devices.get(), nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device ids");

for (auto j = 0; j < m; j++) {

size\_t k;

clGetDeviceInfo(devices[j], CL\_DEVICE\_HOST\_UNIFIED\_MEMORY, 0, nullptr, &k);

std::unique\_ptr<char[]> type(new char[k]);

clGetDeviceInfo(devices[j], CL\_DEVICE\_HOST\_UNIFIED\_MEMORY, k, type.get(), nullptr);

if (\*reinterpret\_cast<cl\_bool\*>(type.get()) == CL\_FALSE) {

if (device\_num == 0) {

device = devices[j];

break;

}

else {

device\_num--;

}

}

}

}

if (device == nullptr) {

for (auto i = 0; i < n; i++) {

cl\_uint m;

error = clGetDeviceIDs(platforms[i], CL\_DEVICE\_TYPE\_GPU, 0, nullptr, &m);

if (error == CL\_DEVICE\_NOT\_FOUND) continue;

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device ids");

std::unique\_ptr<cl\_device\_id[]> devices;

try {

devices = std::unique\_ptr<cl\_device\_id[]>(new cl\_device\_id[m]);

} catch (...) {

throw std::runtime\_error("Could not allocate memory");

}

error = clGetDeviceIDs(platforms[i], CL\_DEVICE\_TYPE\_GPU, m, devices.get(), nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device ids");

for (auto j = 0; j < m; j++) {

size\_t k;

clGetDeviceInfo(devices[j], CL\_DEVICE\_HOST\_UNIFIED\_MEMORY, 0, nullptr, &k);

std::unique\_ptr<char[]> type(new char[k]);

clGetDeviceInfo(devices[j], CL\_DEVICE\_HOST\_UNIFIED\_MEMORY, k, type.get(), nullptr);

if (\*reinterpret\_cast<cl\_bool\*>(type.get()) == CL\_TRUE) {

if (device\_num == 0) {

device = devices[j];

break;

}

else {

device\_num--;

}

}

}

}

}

if (device == nullptr) {

for (auto i = 0; i < n; i++) {

cl\_uint m;

error = clGetDeviceIDs(platforms[i], CL\_DEVICE\_TYPE\_CPU, 0, nullptr, &m);

if (error == CL\_DEVICE\_NOT\_FOUND) continue;

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device ids");

std::unique\_ptr<cl\_device\_id[]> devices;

try {

devices = std::unique\_ptr<cl\_device\_id[]>(new cl\_device\_id[m]);

} catch (...) {

throw std::runtime\_error("Could not allocate memory");

}

error = clGetDeviceIDs(platforms[i], CL\_DEVICE\_TYPE\_CPU, m, devices.get(), nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device ids");

for (auto j = 0; j < m; j++) {

if (device\_num == 0) {

device = devices[j];

break;

} else {

device\_num--;

}

}

}

}

if (device == nullptr) throw std::runtime\_error("Could not find device");

{

size\_t m;

error = clGetDeviceInfo(device, CL\_DEVICE\_VENDOR, 0, nullptr, &m);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device vendor");

std::unique\_ptr<char[]> name;

try {

name = std::unique\_ptr<char[]>(new char[m]);

} catch (...) {

throw std::runtime\_error("Could not allocate memory");

}

error = clGetDeviceInfo(device, CL\_DEVICE\_VENDOR, m, name.get(), nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device vendor");

std::cout << reinterpret\_cast<char\*>(name.get()) << std::endl;

error = clGetDeviceInfo(device, CL\_DEVICE\_NAME, 0, nullptr, &m);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device name");

try {

name = std::unique\_ptr<char[]>(new char[m]);

} catch (...) {

throw std::runtime\_error("Could not allocate memory");

}

error = clGetDeviceInfo(device, CL\_DEVICE\_NAME, m, name.get(), nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get device name");

std::cout << reinterpret\_cast<char\*>(name.get()) << std::endl;

}

cl\_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, &error);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not create context");

cl\_command\_queue queue = clCreateCommandQueue(context, device, CL\_QUEUE\_PROFILING\_ENABLE, &error);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not create command queue");

std::string code\_program; if (implement\_num == '1') {

code\_program += "kernel void sum(global const float \*at, global const float \*b, global float \*c, int const N, int const K, int const M) {\n";

code\_program += " uint x = get\_global\_id(0);\n";

code\_program += " uint y = get\_global\_id(1);\n";

code\_program += " c[y \* N + x] = 0;\n";

code\_program += " for (int i = 0; i < K; i++) {\n";

code\_program += " c[y \* N + x] += at[i \* M + y] \* b[i \* N + x];\n";

code\_program += " }\n";

code\_program += "}\n";

} else if (implement\_num == '2') {

code\_program += "kernel void sum(global const float \*at, global const float \*b, global float \*c, int const N, int const K, int const M) {\n";

code\_program += " local float first\_arr[" + std::to\_string(LOCAL\_SIZE) + "][" + std::to\_string(LOCAL\_SIZE) + "];\n";

code\_program += " local float second\_arr[" + std::to\_string(LOCAL\_SIZE) + "][" + std::to\_string(LOCAL\_SIZE) + "];\n";

code\_program += " uint x = get\_global\_id(0);\n";

code\_program += " uint x\_local = get\_local\_id(0);\n";

code\_program += " uint y = get\_global\_id(1);\n";

code\_program += " uint y\_local = get\_local\_id(1);\n";

code\_program += " c[y \* N + x] = 0;\n";

code\_program += " for (int i = 0; i < K; i += " + std::to\_string(LOCAL\_SIZE) + ") {\n";

code\_program += " first\_arr[y\_local][x\_local] = at[(i + x\_local) \* M + y];\n";

code\_program += " second\_arr[y\_local][x\_local] = b[(i + y\_local) \* N + x];\n";

code\_program += " barrier(CLK\_LOCAL\_MEM\_FENCE);\n";

code\_program += " for (int uu = 0; uu < " + std::to\_string(LOCAL\_SIZE) + "; uu++) {\n";

code\_program += " c[y \* N + x] += first\_arr[y\_local][uu] \* second\_arr[uu][x\_local];\n";

code\_program += " }\n";

code\_program += " }\n";

code\_program += "}\n";

} else {

code\_program += "kernel void sum(global const float \*at, global const float \*b, global float \*c, int const N, int const K, int const M) {\n";

code\_program += " local float first\_arr[" + std::to\_string(LOCAL\_SIZE) + "][" + std::to\_string(LOCAL\_SIZE) + "];\n";

code\_program += " local float" + std::to\_string(VECTOR\_SIZE) + " second\_arr[" + std::to\_string(LOCAL\_SIZE) + "][" + std::to\_string(LOCAL\_SIZE) + "];\n";

code\_program += " uint x = get\_global\_id(0);\n";

code\_program += " uint x\_local = get\_local\_id(0);\n";

code\_program += " uint y = get\_global\_id(1);\n";

code\_program += " uint y\_local = get\_local\_id(1);\n";

code\_program += " float" + std::to\_string(VECTOR\_SIZE) + " ans = 0;\n";

code\_program += " for (int i = 0; i < K; i += " + std::to\_string(LOCAL\_SIZE) + ") {\n";

code\_program += " first\_arr[y\_local][x\_local] = at[(i + x\_local) \* M + y];\n";

code\_program += " second\_arr[y\_local][x\_local] = vload" + std::to\_string(VECTOR\_SIZE) + "(0, &b[(i + y\_local) \* N + " + std::to\_string(VECTOR\_SIZE) + " \* x]);\n";

code\_program += " barrier(CLK\_LOCAL\_MEM\_FENCE);\n";

code\_program += " for (int uu = 0; uu < " + std::to\_string(LOCAL\_SIZE) + "; uu++) {\n";

code\_program += " ans += first\_arr[y\_local][uu] \* second\_arr[uu][x\_local];\n";

code\_program += " }\n";

code\_program += " }\n";

code\_program += " vstore" + std::to\_string(VECTOR\_SIZE) + "(ans, 0, &c[y \* N + " + std::to\_string(VECTOR\_SIZE) + " \* x]);\n";

code\_program += "}\n";

} char const\* code = code\_program.c\_str();

cl\_program program = clCreateProgramWithSource(context, 1, &code, nullptr, &error);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not create program");

error = clBuildProgram(program, 1, &device, nullptr, nullptr, nullptr);

if (error != CL\_SUCCESS) {

size\_t q;

error = clGetProgramBuildInfo(program, device, CL\_PROGRAM\_BUILD\_LOG, 0, nullptr, &q);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get build info");

std::unique\_ptr<char[]> log;

try {

log = std::unique\_ptr<char[]>(new char[q]);

} catch (...) {

throw std::runtime\_error("Could not allocate memory");

}

error = clGetProgramBuildInfo(program, device, CL\_PROGRAM\_BUILD\_LOG, q, log.get(), nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not get build info (2)");

std::cout << log.get() << std::endl;

throw std::runtime\_error("Could not build program");

}

cl\_kernel kernel = clCreateKernel(program, "sum", &error);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not create kernel");

cl\_mem buffer\_a = clCreateBuffer(context, CL\_MEM\_READ\_WRITE, MM \* KK \* 4, nullptr, &error);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not create buffer a");

error = clEnqueueWriteBuffer(queue, buffer\_a, CL\_FALSE, 0, MM \* KK \* 4, new\_first\_matrix.data(), 0, nullptr, nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not write buffer a");

cl\_mem buffer\_b = clCreateBuffer(context, CL\_MEM\_READ\_WRITE, KK \* NN \* 4, nullptr, &error);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not create buffer b");

error = clEnqueueWriteBuffer(queue, buffer\_b, CL\_FALSE, 0, KK \* NN \* 4, new\_second\_matrix.data(), 0, nullptr, nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not write buffer b");

cl\_mem buffer\_c = clCreateBuffer(context, CL\_MEM\_READ\_WRITE, MM \* NN \* 4, nullptr, &error);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not create buffer c");

error = clSetKernelArg(kernel, 0, 8, &buffer\_a);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not set first argument");

error = clSetKernelArg(kernel, 1, 8, &buffer\_b);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not set second argument");

error = clSetKernelArg(kernel, 2, 8, &buffer\_c);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not set third argument");

error = clSetKernelArg(kernel, 3, 4, &NN);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not set fourth argument");

error = clSetKernelArg(kernel, 4, 4, &KK);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not set fifth argument");

error = clSetKernelArg(kernel, 5, 4, &MM);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not set sixth argument");

size\_t const dim[2] = { (implement\_num == '3' ? NN / VECTOR\_SIZE : NN), MM };

size\_t const local\_size[2] = { LOCAL\_SIZE, LOCAL\_SIZE };

size\_t const\* local\_ptr = (implement\_num == '1' ? nullptr : local\_size);

cl\_event event;

error = clEnqueueNDRangeKernel(queue, kernel, 2, nullptr, dim, local\_ptr, 0, nullptr, &event);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not enqueue ND range kernel");

std::vector<float> cl\_ans(MM \* NN);

error = clEnqueueReadBuffer(queue, buffer\_c, CL\_TRUE, 0, MM \* NN \* 4, cl\_ans.data(), 0, nullptr, nullptr);

if (error != CL\_SUCCESS) throw std::runtime\_error("Could not enque read buffer");

auto t\_end = std::chrono::high\_resolution\_clock::now();

std::ofstream output(output\_file);

if (output.fail()) throw std::runtime\_error("Could not open file for writing");

output << N << " " << M << std::endl; for (size\_t i = 0; i < M; i++) {

for (size\_t j = 0; j < N; j++) {

output << cl\_ans[i \* NN + j];

if (j + 1 != N) {

output << " ";

} else {

output << std::endl;

}

}

}

cl\_ulong start\_time, finish\_time;

clGetEventProfilingInfo(event, CL\_PROFILING\_COMMAND\_START, sizeof(cl\_ulong), &start\_time, nullptr);

clGetEventProfilingInfo(event, CL\_PROFILING\_COMMAND\_END, sizeof(cl\_ulong), &finish\_time, nullptr);

std::cout << "\nTime: " << (finish\_time - start\_time) / 1'000'000 << "\t" << std::chrono::duration<double, std::milli>(t\_end - t\_start).count() << "\n";

}