Московский авиационный институт
(национальный исследовательский университет)
Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Курсовой проект по курсу «Численные методы»
на тему
«Распараллеливание алгоритмов решения задач линейной алгебры»

Студент: Мукин Ю. Д.
Группа: 80-304Б-18
Преподаватель: Гидаспов В.Ю.

Москва 2021

Оглавление

Постановка задачи....3
1. Синхронный QR алгоритм....3
2. Оптимизация вычислений с помощью технологии OpenCL....4
3. Оптимизация QR алгоритма....5
4) Сравнение производительности....5
5) Техническая реализация....6
Вывод....14
Список литературы....15

Постановка задачи.

Необходимо реализовать параллельный алгоритм QR разложения матрицы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений.

1. Синхронный QR алгоритм.

Пусть дана квадратная матрица A, с размерностью n. Для этой матрицы необходимо вычислить с точностью ε все собственные значения. Алгоритм их нахождения основанный на QR разложении состоит из следующих пунктов:

1. Найдем QR разложении матрицы A:
    1. Рассмотрим матрицу A как совокупность линейно независимых векторов $a_1 \dots a_n$. Пусть $proj_b a$ - оператор проекции вектора a на вектор b, определенный как:
    2. $proj_b a = \frac{\langle a,b \rangle}{\langle b,b \rangle} b$, где $\langle a,b \rangle$ - скалярное произведение векторов a и b.
    3. Рассчитаем вектора $b_1 \dots b_n$ по следующей рекуррентный формуле:
    4. $$\begin{aligned} b_1 &= a_1 \\ b_2 &= a_2 - proj_{b_1} a_2 \\ b_3 &= a_3 - proj_{b_1} a_3 - proj_{b_2} a_3 \\ &\dots \\ b_n &= a_n - proj_{b_1} a_n - proj_{b_2} a_n - proj_{b_{n-1}} a_n \end{aligned}$$
    5. Рассчитаем вектор $e_i = \frac{b_i}{||b_i||}$, $i = 1 \dots n$.
    6. Из векторов e составим матрицу $Q = (e_1 \dots e_n)^T$.
    7. С помощью ранее рассчитанной матрицы определим R по формуле:

       $R = Q^{-1} A$. Поскольку матрица Q является ортогональной, можно заменить Q^{-1} на Q^T $\Rightarrow$ $R = Q^T A$.
2. Найдем собственные значения матрицы A:
    1. Рассчитаем следующее значение матрицы A по формуле: $A^{(k+1)} = R^k Q^k$.
    2. На диагонали полученной матрицы находятся k-ое приближение собственных значений матрицы A.

Существует два критерия остановки:

1. сумма поддиаганальных элементов меньше заданной эпсилон;
2. Разность двух идущих друг за другом найденных собственных значений по модулю меньше заданной эпсилон;

2. Оптимизация вычислений с помощью технологии OpenCL.

OpenCL – фреймвор предназначенный для выполнения параллельных вычислений на CPU и GPU. Синтаксис OpenCL соответствует стандарту C99, что делает его простым в освоении для пользователей знакомых с языком программирования СИ.

Тот факт, что вычисления могут производиться не только на CPU, но и на GPU накладывает ряд ограничений на использование некоторых операций. Так, к примеру, крайне не эффективно использовать оператор if. Это обусловлено архитектурными особенностями GPU, при выполнении кода графический ускоритель выделяет группу ядер, которые управляются одним единственным планировщиком (рис.1), в противовес GPU, в CPU у каждого ядра собственный планировщик задач (рис.2), благодаря чему код находящийся в блоке if else, эффективно выполняемый как в однопоточном, так и многпотомном варианте исполнения на CPU, на GPU будет выполнен каждым потоком, вне зависимости от той информации которой они располагают, что приведет к значительному снижению производительности.

Важным ограничением накладываемым параллельными вычислениями является невозможность эффективной реализации скалярного произведения векторов. Так как задача нахождения скалярного произведения сводится к суммированию элементов массива, возникает необходимость использовать функцию `atomic_add`, скорость работы которой практически невозможно скомпенсировать количеством потоков. Отказ от использования `atomic_add`, влечет за собой *гонку потоков*, что приводит к непредсказуемому поведению программы.

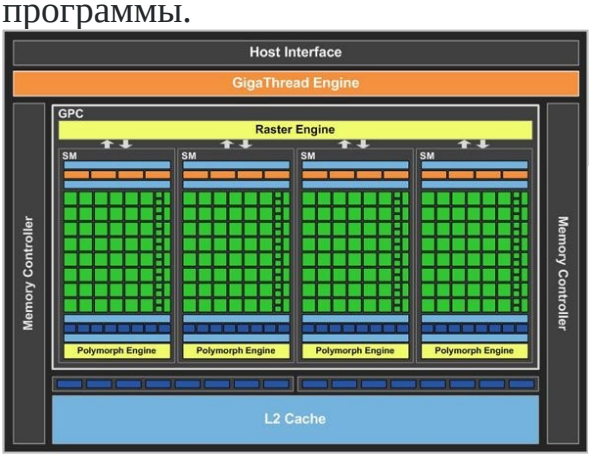


рис.1

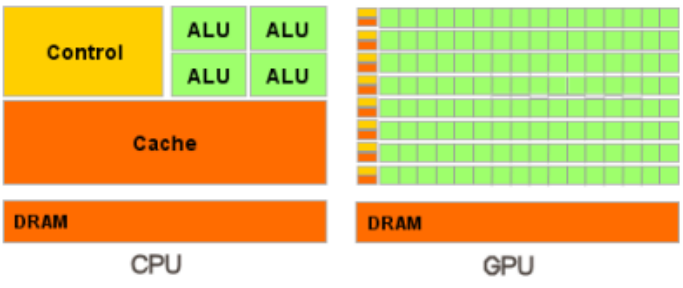


рис.2

Однако, несмотря на все накладываемые ограничения, графический процессор является превосходным инструментом для обработки больших объемов данных, в силу возможности выполнять более тысячи однотипных

потоков команд одновременно, из чего следует высокая эффективность выполнения операций над матрицами.

3. Оптимизация QR алгоритма.

QR алгоритм имеет несколько возможных реализаций, каждая из них предполагает использование рекуррентной функции. Это усложняет использование много поточности, но не исключает полностью.

Самая времязатратная операция поддающаяся распараллеливанию — умножение матриц. Умножение матриц осуществляется с помощью трех вложенных циклов, поэтому оптимизация этой функции внесет не малый вклад в быстродействие программы. С помощью OpenCL это можно осуществить запустив n*n потоков с двумерной индексацией. Таким образом каждый поток будет выполнять лишь один цикл, благодаря чему можно получить существенный прирост скорости выполнения, не смотря на необходимость выполнения довольно «дорогой» операции копирования данных из оперативной памяти в видеопамять.

Операция транспонирования так же предполагает два вложенных цикла, из чего следует, что процессору необходимо выполнить n^2 итераций. В свою очередь до определенной вылечены n (эта величина зависит от модели видеокарты и версии api) GPU может выполнить туже задачу за одну итерацию.

Рекуррентность алгоритма Грама — Шмидта не позволяет разбить нахождение столбцов матрицы Q на отдельные потоки, однако нахождение даже одного столбца предполагает две группы по два вложенных цикла, благодаря чему можно переложить задачу нахождения столбцов b на графический процессор.

Таким образом на видеокарту были переложены следующие пункты алгоритма:

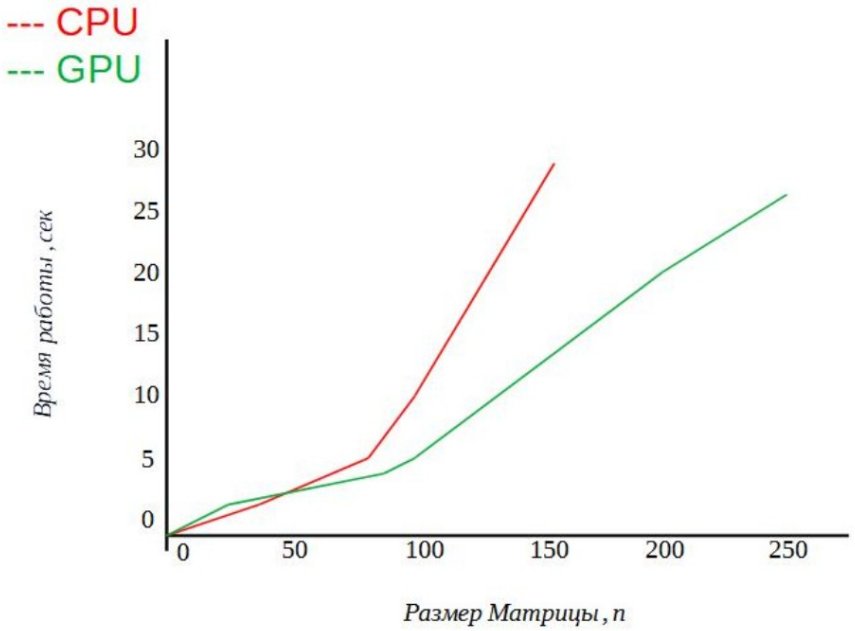
- Расчет вектора $b_1 ... b_n$
- Рассчитаем вектор $e_i = \frac{b_i}{||b_i||}, \quad i = 1...n$.
- вычисление матрицы R

4) Сравнение производительности

Поскольку для произведения вычислений на графическом процессоре необходимо копировать данные из оперативной памяти в видео и обратно при

небольших размерностях матрицы, быстрее провести расчет исключительно на центральном процессоре.

С ростом количества элементов в матрице время необходимое для выполнения расчетов на GPU растет приблизительно линейно, в то время как закон согласно которому изменяет время выполнения вычислений на CPU, ближе к экспоненциальному.



5) Техническая реализация

Программа реализована на языке программирования си, с использованием фреймворка OpenCL.

Код выполняемый на CPU:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <time.h>

#include <math.h>
#ifdef __APPLE__
#include <OpenCL/cl.h>
#else
#include "lib/CL/cl.h"
#endif

#define PROGRAM_FILE "QR.cl"
```

```
#define KERNEL_FUNC "QR_decompos"

cl_kernel matrix_mult, matrix_transpose, matrix_getA, matrix_getB;
cl_command_queue queue;
cl_context context;
cl_int err;


typedef struct matrix
{
   int columns, rows;
   double *body;
}matrix;

matrix create_matrix(int n_row, int n_col)
{
   assert(n_row>0 && n_col>0);
   matrix mat;
   mat.columns = n_col;
   mat.rows = n_row;
   mat.body = calloc(n_col* n_row, sizeof(double));
   return mat;
}

double* get_element(matrix *mat, int row, int col)
{
   assert(col < (*mat).columns && row < (*mat).rows);
   return &(*mat).body[row*((*mat).columns) + col];
}

void print_matrix(matrix *mat)
{
   for (int i=0; i<(*mat).rows; i++)
   {
      printf("\n");
      for (int j=0; j<(*mat).columns; j++)
         printf("%lf ", *get_element(&(*mat), i, j));
   }
}

matrix get_matrix_column(matrix mat, int i)
{
   assert(mat.columns>i);
   matrix res = create_matrix(mat.rows,1);
   for (int j=0; j<mat.rows; j++)
      *get_element(&res, j, 0) = *get_element(&mat, j, i);
   return res;
}

void insert_matrix_column(matrix *matA, matrix matB, int i)
{
```

```
    for (int j=0; j<(*matA).rows; j++)
        *get_element(&(*matA), j, i) = *get_element(&matB, j, 0);
}



matrix multiply_matrix_GPU(matrix matA, matrix matB)
{
    assert(matA.columns == matB.rows);
    matrix matC = create_matrix(matA.rows, matB.columns);
    cl_mem matA_buff, matB_buff, cr_buff, res_buff;
    int *cr = malloc(4*sizeof(int));
    cr[0] = matA.rows; cr[1] = matA.columns; cr[2] = matB.rows; cr[3] = matB.columns;
    matA_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(double)*cr[0]*cr[1], matA.body, &err);
    matB_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(double)*cr[2]*cr[3], matB.body, &err);
    cr_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(int)*4, cr, &err);
    res_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(double)*cr[0]*cr[3], NULL, &err);
    clSetKernelArg(matrix_mult, 0, sizeof(cl_mem), &matA_buff); //линкуем буферы с аргументами фунции ядра
    clSetKernelArg(matrix_mult, 1, sizeof(cl_mem), &matB_buff);
    clSetKernelArg(matrix_mult, 2, sizeof(cl_mem), &cr_buff);
    clSetKernelArg(matrix_mult, 3, sizeof(cl_mem), &res_buff);
    size_t *dimG = malloc(2*sizeof(size_t));
    dimG[0]=matA.rows; dimG[1]=matB.columns;
    clEnqueueNDRangeKernel(queue, matrix_mult, 2, NULL, dimG, NULL, 0, NULL, NULL); //запускаем ядро
    clEnqueueReadBuffer(queue, res_buff, CL_TRUE, 0,sizeof(double)*cr[0]*cr[3], matC.body, 0, NULL, NULL); //значение из буфера
переписываем в понятный си вид
    free(cr); free(dimG); clReleaseMemObject(matA_buff); clReleaseMemObject(matB_buff); clReleaseMemObject(cr_buff);
    return matC;
}


double scalar_product(matrix matA, matrix matB)
{
    assert(matA.columns == matB.columns && matA.rows == matB.rows);
    double res=0;
    if (matA.columns == 1)
    {
        for (int i=0; i< matA.rows; i++)
            res+= (*get_element(&matA, i, 0))*(*get_element(&matB, i, 0));
    }
    else
    {
        for (int i=0; i< matA.columns; i++)
            res+= (*get_element(&matA, 0, i))*(*get_element(&matB, 0, i));
    }
    return res;
}


matrix transpose_matrix_GPU(matrix mat)
{
    matrix result = create_matrix(mat.columns, mat.rows);
    cl_mem matA_buff, matB_buff, cr_buff, res_buff;
    int *cr = malloc(1*sizeof(int));
```

```
    cr[0] = mat.columns;
    matA_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(double)*cr[0]*cr[0], mat.body, &err);
    matB_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(double), NULL, &err);
    cr_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(int), cr, &err);
    res_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(double)*cr[0]*cr[3], NULL, &err);
    clSetKernelArg(matrix_transpose, 0, sizeof(cl_mem), &matA_buff); //линкуем буферы с аргументами фунции ядра
    clSetKernelArg(matrix_transpose, 1, sizeof(cl_mem), &matB_buff);
    clSetKernelArg(matrix_transpose, 2, sizeof(cl_mem), &cr_buff);
    clSetKernelArg(matrix_transpose, 3, sizeof(cl_mem), &res_buff);
    size_t *dimG = malloc(2*sizeof(size_t));
    dimG[0]=mat.rows; dimG[1]=mat.columns;
    clEnqueueNDRangeKernel(queue, matrix_transpose, 2, NULL, dimG, NULL, 0, NULL, NULL); //запускаем ядро
    clEnqueueReadBuffer(queue, res_buff, CL_TRUE, 0,sizeof(double)*cr[0]*cr[0], result.body, 0, NULL, NULL); //значение из буфера
переписываем в понятный си вид
    free(cr); free(dimG); clReleaseMemObject(matA_buff); clReleaseMemObject(matB_buff); clReleaseMemObject(cr_buff);
    return result;
}

void prepare_GPU_code()
{
    cl_platform_id platform;
    cl_device_id device;
    clGetPlatformIDs(1, &platform, NULL); //находим все устройства в системе
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,&device, NULL); //находим первое устройство
    context = clCreateContext(NULL, 1, &device, NULL, NULL, &err); //выбираем устройство для выполнения кода
    FILE *program_handle = fopen("matvec.cl", "r"); //открываем программу выполняемую на GPU
    fseek(program_handle, 0, SEEK_END);
    size_t program_size = ftell(program_handle);
    rewind(program_handle);
    char *program_buffer = (char*)malloc(program_size + 1);
    program_buffer[program_size] = '\0'; //создаем буфер для программы
    fread(program_buffer, sizeof(char), program_size, program_handle); //записываем в него программу
    fclose(program_handle); //закрываем файл
    cl_program program = clCreateProgramWithSource(context, 1,(const char**)&program_buffer, &program_size, &err); //создаем программу
    free(program_buffer); //чистим буфер
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL); //компилируем
    matrix_mult = clCreateKernel(program, "matrix_mult", &err); //создаем указатель на функцию
    printf("%s\n", getErrorString(err));
    matrix_transpose = clCreateKernel(program, "matrix_transpose", &err);
    printf("%s\n", getErrorString(err));
    matrix_getA = clCreateKernel(program, "matrix_getA", &err);
    printf("%s\n", getErrorString(err));
    matrix_getB = clCreateKernel(program, "matrix_getB", &err);
    printf("%s\n", getErrorString(err));
    queue = clCreateCommandQueue(context, device, 0, &err);
}

void get_c(matrix a, matrix b, matrix *c, int l)
{
    cl_mem matA_buff, matB_buff, cr_buff, res_buff;
    int *cr = malloc(sizeof(int));
```

```
    cr[0] = b.columns;
    matA_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(double)*cr[0], a.body, &err);
    matB_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(double)*b.rows*b.columns, b.body,
&err);
    cr_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(int), cr, &err);
    res_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(double)*(*c).rows, NULL, &err);
    clSetKernelArg(matrix_getA, 0, sizeof(cl_mem), &matA_buff); //линкуем буферы с аргументами фунции ядра
    clSetKernelArg(matrix_getA, 1, sizeof(cl_mem), &matB_buff);
    clSetKernelArg(matrix_getA, 2, sizeof(cl_mem), &cr_buff);
    clSetKernelArg(matrix_getA, 3, sizeof(cl_mem), &res_buff);
    size_t dim=l;
    clEnqueueNDRangeKernel(queue, matrix_getA, 1, NULL, &dim, NULL, 0, NULL, NULL); //запускаем ядро
    clEnqueueReadBuffer(queue, res_buff, CL_TRUE, 0,sizeof(double)*l, c->body, 0, NULL, NULL); //значение из буфера переписываем в
понятный си вид
    free(cr); clReleaseMemObject(matA_buff); clReleaseMemObject(matB_buff); clReleaseMemObject(cr_buff);
}


void get_b(matrix *a, matrix b, matrix c, int k)
{
    cl_mem matA_buff, matB_buff, cr_buff, res_buff;
    int *cr = malloc(2*sizeof(int));
    cr[0] = k; cr[1] = b.columns;
    matA_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(double)*b.columns*b.rows, b.body,
&err);
    matB_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(double)*b.columns-1, c.body, &err);
    cr_buff = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(int)*2, cr, &err);
    res_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(double)*b.columns, a->body, &err);
    clSetKernelArg(matrix_getB, 0, sizeof(cl_mem), &matA_buff); //линкуем буферы с аргументами фунции ядра
    clSetKernelArg(matrix_getB, 1, sizeof(cl_mem), &matB_buff);
    clSetKernelArg(matrix_getB, 2, sizeof(cl_mem), &cr_buff);
    clSetKernelArg(matrix_getB, 3, sizeof(cl_mem), &res_buff);
    size_t dim=b.columns;
    clEnqueueNDRangeKernel(queue, matrix_getB, 1, NULL, &dim, NULL, 0, NULL, NULL); //запускаем ядро
    clEnqueueReadBuffer(queue, res_buff, CL_TRUE, 0,sizeof(double)*b.columns, a->body, 0, NULL, NULL); //значение из буфера переписываем
в понятный си вид
    free(cr); clReleaseMemObject(matA_buff); clReleaseMemObject(matB_buff); clReleaseMemObject(cr_buff);
}


void get_QR_GPU(matrix matA, matrix *matQ, matrix *matR)
{
    int n = matA.columns;
    double ab, bb, norm;
    matrix c, a, b, e;
    a = create_matrix(n,1);
    b = create_matrix(n,n);
    e = create_matrix(n,1);
    c = create_matrix(n-1,1);
    for (int i=0; i<n; i++)
    {
        if (i==0)
        {
            insert_matrix_column(&b,get_matrix_column(matA, i),i);
            norm = scalar_product(get_matrix_column(b, i),get_matrix_column(b, i));
```

```
            for(int j =0;j<n; j++)
                *get_element(&e, j, 0) = (*get_element(&b, j, 0))/sqrt(norm);
        }
        else
        {
            a = get_matrix_column(matA, i);
            get_c(a,b,&c,i);
            get_b(&a,b,c,i);
            insert_matrix_column(&b,a,i);
            norm = scalar_product(a,a);
            for(int j =0;j<n; j++)
                *get_element(&e, j, 0) = (*get_element(&b, j, i))/sqrt(norm);
        }
        insert_matrix_column(&(*matQ),e, i);
    }
    (*matR) = multiply_matrix_GPU(transpose_matrix_GPU(*matQ), matA);
}


double square_sum_column(matrix mat, int column_number, int first_index)
{
    int n = mat.columns;
    double sum =0;
    for(int i=first_index; i<n;i++)
        sum+= (*get_element(&mat,i, column_number)) * (*get_element(&mat,i, column_number));
    return sqrt(sum);
}


void get_lambda(double** lambda, matrix mat)
{
    int n = mat.columns;
    for(int j=0; j<n;j++)
        if(j==n-1)
        {
            lambda[j][0] = *get_element(&mat, j, j);
            lambda[j][1] = 0;
        }
        else
        {
            double aii = *get_element(&mat, j, j);
            double ajj = *get_element(&mat, j+1, j+1);
            double aij = *get_element(&mat, j, j+1);
            double aji = *get_element(&mat, j+1, j);
            double x = (aii + ajj) / 2;
            double D = (-(aii+ajj)*(aii+ajj) + 4*(aii*ajj - aij*aji));
            if (D<0)
            {
                lambda[j][0] = *get_element(&mat, j, j);
                lambda[j][1] = 0;
            }
```

```
            else
            {
                double y = sqrt(D) / 2;
                lambda[j][0] = x;
                lambda[j][1] = y;
                j++;
                lambda[j][0] = x;
                lambda[j][1] = -y;
            }
        }
}

int is_end_by_lambda(double** lambda_old, double** lambda_new, double eps, int n)
{
    int res = 0;
    for(int i=0; i<n; i++)
        if(fabs(lambda_old[i][0]-lambda_new[i][0])>eps)
            res++;
    return res;
}

void run_on_GPU(matrix mat, double eps, int n)
{
    matrix matR, matQ, matA = mat;
    int k;
    k=0;
    matR = create_matrix(n,n); matQ = create_matrix(n,n);
    double **lambda_old, **lambda_new;
    lambda_old = malloc(n*sizeof(double*));
    lambda_new = malloc(n*sizeof(double*));
    for(int i=0; i<n; i++)
    {
        lambda_old[i] = malloc(2*sizeof(double));
        lambda_new[i] = malloc(2*sizeof(double));
    }
    get_lambda(lambda_new, matA);
    do
    {
        for(int i=0; i<n; i++)
        {
            lambda_old[i][0] = lambda_new[i][0];
            lambda_old[i][1] = lambda_new[i][1];
        }
        get_QR(matA, &matQ, &matR);
        matA = multiply_matrix(matR, matQ);
        get_lambda(lambda_new, matA);
        k++;
    } while (is_end_by_lambda(lambda_old, lambda_new, eps, mat.columns)!=0 && k<20000);
    printf("\n%d iterations have been done\n", k);
    for(int i=0; i<n; i++)
        printf("\nlambda%d: %lf + %lfi", i, lambda_new[i][0],  lambda_new[i][1]);
```

```
}


int main()
{
   prepare_GPU_code();
   matrix matA;
   clock_t tic;
   int n, k;
   double eps;
   float cpu, gpu;
   k=0;
   FILE *fil = fopen("input.txt", "r");
   printf("enter the dimension of the matrix: ");
   scanf("%d", &n);
   //printf("enter the matrix in one line: ");
   matA = create_matrix(n,n);
   for (int i=0; i<matA.columns; i++)
      for (int j=0; j<matA.columns; j++)
         //scanf("%lf", &*get_element(&matA, i, j));
         fscanf(fil, "%lf", &*get_element(&matA, i, j));
   printf("\nmatrix read successfully\n");
   fclose(fil);
   printf("enter the desired precision: ");
   scanf("%lf", &eps);
   tic = clock();
   run_on_GPU(matA, eps, n);
   gpu = (double)(clock() - tic) / CLOCKS_PER_SEC;
   printf("\ncomputed on GPU by %lf", gpu);
}
```

Код выполняемый на GPU

```
__kernel void matrix_mult(__global double* matA, __global double* matB, __global int* row_column, __global double* result)
{
   int i = get_global_id(0);
   int j = get_global_id(1);
   for(int k = 0; k < row_column[2]; k++)
      result[i*row_column[3]+j] += matA[i*row_column[1]+k] * matB[k*row_column[3]+j];
}

__kernel void matrix_transpose(__global double* matA, __global double* matB, __global int* row_column, __global double* result)
{
   int i = get_global_id(0);
   int j = get_global_id(1);
   result[j*row_column[0]+i] = matA[i*row_column[0]+j];
}

__kernel void matrix_getA(__global double* matA, __global double* matB, __global int* row_column, __global double* result)
{
   int i = get_global_id(0);
   int j = get_global_id(1);
```

```
    double ab=0, bb=0;
    for(int k = 0; k < row_column[0]; k++)
    {
        ab+=matA[k]*matB[k*row_column[0]+i];
        bb+=matB[k*row_column[0]+i]*matB[k*row_column[0]+i];
    }
    result[i] = ab/bb;
}

__kernel void matrix_getB(__global double* matA, __global double* matB, __global int* row_column, __global double* result)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    for(int k = 0; k < row_column[0]; k++)
        result[i] -= matA[i*row_column[1]+k]*matB[k];
}
```

Вывод.

Вычисления на GPU крайне эффективны в случаях, когда необходимо обработать больной объем данных, но совершенно лишены смысла если набор данных состоит менее чем из ≈2000 записей. Это обусловлено тем, что выигрыш от параллельных вычислений нивелируется дорогими с точки зрения процессорного времени операциями копирования данных из оперативной памяти в память графического ускорители.

Список литературы.

1. Matthew Scarpino OpenCL in Action ISBN 9781617290176 November 2011
2. интернет — источники https://habr.com/ru/post/261323/ - хабр
3. интернет — источники https://ru.wikipedia.org/wiki/QR-%D1%80%D0%B0%D0%B7%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5 — свободная библиотека Wikipedia
4. интернет — источники https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/ - документация OpenCL