Московский Авиационный Институт
(национальный исследовательский
университет)

Факультет прикладной математики и физики

Курсовой проект по курсу «Численные методы»

на тему:«Распараллеливание вычислительных алгоритмов решения задач линейной алгебры.»

Студент: А. И. Балес
Преподаватель: Д. Л. Ревизников

Москва, 2016

Постановка задачи:

Необходимо реализовать алгоритм параллельного нахождения собственных значений и собственных векторов `методом вращений (Якоби)`. Для решения данной проблемы можно пойти по одному из следующих путей:

- Путь наименьшего сопротивления – распараллелить не сам алгоритм как таковой, а постараться распараллелить как можно больше его подчастей, т.е. перемножение матриц, транпспонирование матрицы, поиска максимального элемента в матрице. Данный способ менее эффективен чем тот, который будет описан в следующем пункте.

- Постараться распараллелить не части алгоритма, а целиком направить вычисления на каждый процессор, а затем просто склеить значения воедино и все. Сложность представляется в том, что принцип разделения задачи на подзадачи не очевиден.

К сожалению, я пошел не по тому пути, по которому следовало бы... Давайте рассмотрим метод Якоби и попытаемся порассуждать о том, как можно было бы разбить на подзадачи данный агоритм.

1.2.2. Метод вращений Якоби численного решения задач на собственные значения и собственные векторы матриц

Метод вращений Якоби применим только для симметрических матриц A_{nxn} ($A = A^T$) и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия $\Lambda = U^{-1}AU$, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной ($U^{-1} = U^T$), то $\Lambda = U^T AU$, где Λ - диагональная матрица с собственными значениями на главной диагонали

$$\Lambda = \begin{pmatrix} \lambda_1 & \dots & 0 \\ \dots & \ddots & \dots \\ 0 & \dots & \lambda_n \end{pmatrix}.$$

Пусть дана симметрическая матрица *A*. Требуется для нее вычислить с точностью ε все собственные значения и соответствующие им собственные векторы. Алгоритм метода вращения следующий:

Пусть известна матрица $A^{(k)}$ на *k*–й итерации, при этом для *k*=0 $A^{(0)} = A$.

1. Выбирается максимальный по модулю недиагональный элемент $a_{ij}^{(k)}$ матрицы $A^{(k)}$ ($\left|a_{ij}^{(k)}\right| = \max_{l<m}\left|a_{lm}^{(k)}\right|$).

2. Ставится задача найти такую ортогональную матрицу $U^{(k)}$, чтобы в результате преобразования подобия $A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)}$ произошло обнуление элемента $a_{ij}^{(k+1)}$ матрицы $A^{(k+1)}$. В качестве ортогональной матрицы выбирается матрица вращения, имеющая следующий вид:

$$U^k = \begin{array}{c} \begin{matrix} & & & i & & & j & & \end{matrix} \\ \begin{pmatrix} 1 & & & \vdots & & & \vdots & & \\ & \ddots & & \vdots & & & \vdots & 0 & \\ & & 1 & \vdots & & & \vdots & & \\ \cdots & \cdots & \cdots & \cos\varphi^{(k)} & \cdots & -\sin\varphi^{(k)} & \cdots & \cdots & \end{pmatrix} i \\ \begin{pmatrix} & & & \vdots & 1 & & \vdots & & \\ & & & \vdots & & \ddots & \vdots & & \\ & & & \vdots & & & 1 & & \\ \cdots & \cdots & \cdots & \sin\varphi^{(k)} & \cdots & \cos\varphi^{(k)} & \cdots & \cdots & \end{pmatrix} j \\ \begin{matrix} & & & & & & & & \end{matrix} \end{array},$$

В матрице вращения на пересечении $i-$й строки и $j-$го столбца находится элемент $u_{ij}^{(k)} = -\sin\varphi^{(k)}$, где $\varphi^{(k)}$ - угол вращения, подлежащий определению. Симметрично относительно главной диагонали (j-я строка, i-й столбец) расположен элемент $u_{ji}^{(k)} = \sin\varphi^{(k)}$; Диагональные элементы $u_{ii}^{(k)}$ и $u_{jj}^{(k)}$ равны соответственно $u_{ii}^{(k)} = \cos\varphi^{(k)}$, $u_{jj}^{k} = \cos\varphi^{(k)}$; другие диагональные элементы $u_{mm}^{(k)} = 1, m = \overline{1,n},\ m \neq i,\ m \neq j$; остальные элементы в матрице вращения $U^{(k)}$ равны нулю.

Угол вращения $\varphi^{(k)}$ определяется из условия $a_{ij}^{(k+1)} = 0$:

$$\varphi^{(k)} = \frac{1}{2} arctg \frac{2a_{ij}^{(k)}}{a_{ii}^{(k)} - a_{jj}^{(k)}},$$

причем если $a_{ii}^{(k)} = a_{jj}^{(k)}$, то $\varphi^{(k)} = \frac{\pi}{4}$.

3. Строится матрица $A^{(k+1)}$

$$A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)},$$

в которой элемент $a_{ij}^{(k+1)} \approx 0.$

В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов внедиагональных элементов:

$$t\left(A^{(k+1)}\right) = \left(\sum_{l,m;l<m} \left(a_{lm}^{(k+1)}\right)^2 \right)^{1/2}.$$

Если $t\left(A^{(k+1)}\right) > \varepsilon,$ то итерационный процесс

$$A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)} = U^{(k)T} U^{(k-1)T} ... U^{(0)T} A^{(0)} U^{(0)} U^{(1)} ... U^{(k)}$$

продолжается. Если $t\left(A^{(k+1)}\right) < \varepsilon$, то итерационный процесс останавливается, и в качестве искомых собственных значений принимаются $\lambda_1 \approx a_{11}^{(k+1)}$, $\lambda_2 \approx a_{22}^{(k+1)}, ..., \lambda_n \approx a_{nn}^{(k+1)}$.

Координатными столбцами собственных векторов матрицы A в единичном базисе будут столбцы матрицы $U = U^{(0)} U^{(1)} ... U^{(k)}$, т.е.

$$\left(x^1\right)^T = \left(u_{11}\, u_{21}\, ... u_{n1}\right), \quad \left(x^2\right)^T = \left(u_{12}\, u_{22}\, ... u_{n2}\right), \ \left(x^n\right)^T = \left(u_{1n}\, u_{2n}\, ... u_{nn}\right),$$

причем эти собственные векторы будут ортогональны между собой, т.е. $\left(x^l, x^m\right) \approx 0,\ l \neq m.$

Пример 1.7. С точностью $\varepsilon = 0{,}3$ вычислить собственные значения и собственные векторы матрицы

$$A = \begin{bmatrix} 4 & 2 & 1 \\ 2 & 5 & 3 \\ 1 & 3 & 6 \end{bmatrix} \equiv A^{(0)}.$$

Р е ш е н и е.

1). Выбираем максимальный по модулю внедиагональный элемент матрицы $A^{(0)}$, т.е. находим $a_{ij}^{(0)}$, такой что $\left|a_{ij}^{(0)}\right| = \max_{l<m} \left|a_{lm}^{(0)}\right|$. Им является элемент $a_{23}^{(0)} = 3$.

2). Находим соответствующую этому элементу матрицу вращения:

$$U^{(0)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi^{(0)} & -\sin\varphi^{(0)} \\ 0 & \sin\varphi^{(0)} & \cos\varphi^{(0)} \end{bmatrix}; \varphi^{(0)} = \frac{1}{2} arctg \frac{2 \cdot 3}{5-6} =$$

$= -0{,}7033;\ \sin\varphi^{(0)} = -0{,}65;\ \cos\varphi^{(0)} = 0{,}76;$

$$U^{(0)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0{,}76 & 0{,}65 \\ 0 & -0{,}65 & 0{,}76 \end{bmatrix}; .$$

3). Вычисляем матрицу $A^{(1)}$:

$$A^{(1)} = U^{(0)\,T} A^{(0)} U = \begin{bmatrix} 4 & 0{,}87 & 2{,}06 \\ 0{,}87 & 2{,}46 & -0{,}03 \\ 2{,}06 & -0{,}03 & 8{,}54 \end{bmatrix}.$$

В полученной матрице с точностью до ошибок округления элемент $a_{23}^{(1)} = 0$.

$t\left(A^{(1)}\right) = \left(\sum_{l,m;l<m} \left(a_{lm}^{(1)}\right)^2 \right)^{1/2} = (0{,}87^2 + 2{,}06^2 + (-0{,}03)^2)^{1/2} > \varepsilon$, следовательно итерационный процесс необходимо продолжить.

Переходим к следующей итерации $(k = 1)$:

$a_{13}^{(1)} = 2{,}06; \quad \left(\left|a_{13}^{(1)}\right| = \max_{l,m;\ l<m} \left|a_{lm}^{(1)}\right| \right).$

$$U^{(1)} = \begin{bmatrix} \cos\varphi^{(1)} & 0 & -\sin\varphi^{(1)} \\ 0 & 1 & 0 \\ \sin\varphi^{(1)} & 0 & \cos\varphi^{(1)} \end{bmatrix};$$

$$\varphi^{(1)} = \frac{1}{2} arctg \frac{2 \cdot 2{,}06}{4 - 8{,}54} = -0{,}3693; \quad \sin\varphi^{(1)} = -0{,}361;\ \cos\varphi^{(1)} = 0{,}933;$$

$$U^{(1)} = \begin{bmatrix} 0{,}933 & 0 & 0{,}361 \\ 0 & 1 & 0 \\ -0{,}361 & 0 & 0{,}933 \end{bmatrix}; \quad A^{(2)} = U^{(1)\,T} A^{(1)} U^{(1)} = \begin{bmatrix} 3{,}19 & 0{,}819 & 0{,}005 \\ 0{,}819 & 2{,}46 & 0{,}28 \\ 0{,}005 & 0{,}28 & 9{,}38 \end{bmatrix}.$$

$$t\left(A^{(2)}\right) = \left(\sum_{l,m;l<m} \left(a_{lm}^{(2)}\right)^2 \right)^{1/2} = (0{,}819^2 + 0{,}28^2 + 0{,}005^2)^{1/2} > \varepsilon.$$

Переходим к следующей итерации $(k = 2)$

$a_{12}^{(2)} = 0{,}819; \quad \left(\left|a_{12}^{(2)}\right| = \max_{l,m;\ l<m} \left|a_{lm}^{(2)}\right| \right)$

$$\varphi^{(2)} = \frac{1}{2} actg \frac{2 \cdot 0{,}819}{3{,}19 - 2{,}46} = 0{,}5758;\ \sin\varphi^{(2)} = 0{,}5445;\ \cos\varphi^{(2)} = 0{,}8388.$$

$$U^{(2)} = \begin{bmatrix} 0{,}8388 & -0{,}5445 & 0 \\ 0{,}5445 & 0{,}8388 & 0 \\ 0 & 0 & 1 \end{bmatrix};\ A^{(3)} = U^{(2)T} A^{(2)} U^{(2)} = \begin{bmatrix} 3{,}706 & 0{,}0003 & 0{,}1565 \\ 0{,}0003 & 1{,}929 & 0{,}232 \\ 0{,}1565 & 0{,}232 & 9{,}38 \end{bmatrix};$$

$t\left(A^{(3)}\right) = (0{,}0003^2 + 0{,}1565^2 + 0{,}232^2)^{1/2} = 0{,}07839^{1/2} < \varepsilon$.

Таким образом в качестве искомых собственных значений могут быть приняты диагональные элементы матрицы $A^{(3)}$:

$\lambda_1 \approx 3{,}706;\ \lambda_2 \approx 1{,}929;\ \lambda_3 \approx 9{,}38.$

Собственные векторы определяются из произведения

$$U^{(0)}U^{(1)}U^{(2)} = \begin{bmatrix} 0{,}78 & -0{,}5064 & 0{,}361 \\ 0{,}2209 & 0{,}7625 & 0{,}6 \\ -0{,}58 & -0{,}398 & 0{,}7 \end{bmatrix};\ x^1 = \begin{bmatrix} 0{,}78 \\ 0{,}2209 \\ -0{,}58 \end{bmatrix};\ x^2 = \begin{bmatrix} -0{,}5064 \\ 0{,}7625 \\ -0{,}398 \end{bmatrix};\ x^3 = \begin{bmatrix} 0{,}361 \\ 0{,}6 \\ 0{,}7 \end{bmatrix}.$$

Полученные собственные векторы ортогональны в пределах заданной точности, т.е. $\left(x^1, x^2\right) = -0{,}00384;\ \left(x^1, x^3\right) = 0{,}0081;\ \left(x^2, x^3\right) = -0{,}0039.$

Вот мои догадки по поводу того, как бы я решал эту задачу, разбивая алгоритм на подзадачи, которые бы решались параллельно, а затем решения склеивались бы в единую матрицу:

1. Первым делом в обычном методе Якоби ищется максимальный внедиагональный элемент, пусть его координаты будут (m, l), где m–строка, а l–столбец, $\Rightarrow$ после умножения на матрицу вращения элемент $\mathbb{A}[m, l]$ и $\mathbb{A}[l, m]$ обнулятся (в силу симметричности).

2. После того, как выбрали элемент $\mathbb{A}[m, l]$ мы помечаем их как использованные, и в дальнейшем не будем выбирать m, l как строки, так и столбцы. Т.е. дальнейший поиск максимумов будет осуществляться в строках и столбцах не равных m, l.

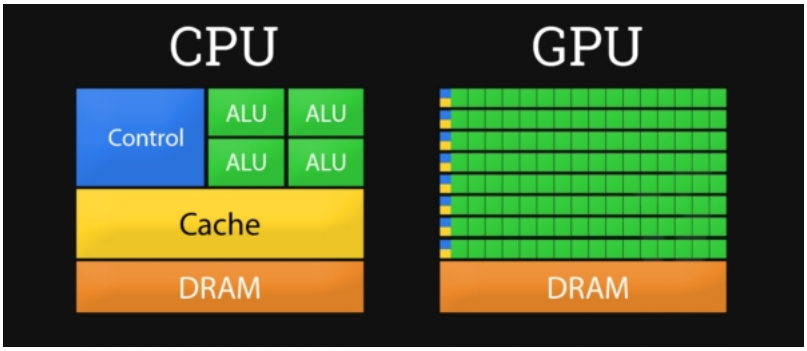
3. Для каждого из таких максимумов мы выделим по две строки (m и l) и два столбца (m и l) и будем осуществлять все операции только с ними. Следует быть осторожным с обработкой элементов, находящийхся на пересечении m строки и l столбца, а также l строки и m столбца, т.к. их нужно изменять в обоих массивах сразу.

4. Теперь сливаем все эти строки и столбцы снова в матрицу, и повторяешь п.1 пока не окажется, что все элементы в матрице $\{\mathbb{A}[i, j] | i \neq j\} < \varepsilon$.

Технологии:

В настоящее время очень широкое применение имеют параллельные вычисления на GPU (графических процессорах), т.к. GPU более производительное, чем CPU. Обусловленно это тем, что GPU имеет огромное количество исполнительных блоков: в современных GPU их 2048 и более, в то время как у CPU их количество может достигать 48, но чаще всего их количество лежит в диапазоне 2-8. Есть множество различий и в поддержке многопоточности: CPU исполняет 1-2 потока вычислений на одно процессорное ядро, а GPU может поддерживать несколько тысяч потоков на каждый мультипроцессор, которых в чипе несколько штук! И если переключение с одного потока на другой для CPU стоит сотни тактов, то GPU переключает несколько потоков за один такт.

В CPU большая часть площади чипа занята под буферы команд, аппаратное предсказание ветвления и огромные объемы кэш-памяти, а в GPU большая часть площади занята исполнительными блоками. Вышеописанное устройство схематично изображено ниже:



Если CPU – это своего рода «начальник», принимающий решения в соответствии с указаниями программы, то GPU – это «рабочий», который производит огромное количество однотипных вычислений. Выходит, что если подавать на GPU независимые простейшие математические задачи, то он справится значительно быстрее, чем центральный процессор.

Предлагаю теперь обсудить технологию, которой я пользовался при распараллеливании отдельных частей метода Якоби. Она называется `OpenCL` (**Open C**omputing **L**anguage – открытый язык вычислений) – фреймворк для написания компьютерных программ, связанных с параллельными вычислениями на различных графических (GPU) и центральных процессорах (CPU), а также FPGA. Во фреймворк OpenCL входят язык программирования, который базируется на стандарте C99, и интерфейс программирования приложений (API). OpenCL обеспечивает параллелизм на уровне инструкций и на уровне данных и является реализацией техники GPGPU. OpenCL является полностью открытым стандартом, его использование не облагается лицензионными отчислениями.

Цель OpenCL состоит в том, чтобы дополнить OpenGL и OpenAL, которые являются открытыми отраслевыми стандартами для трёхмерной компьютерной графики и звука, пользуясь возможностями GPU. OpenCL разрабатывается и поддерживается некоммерческим консорциумом *Khronos Group*, в который входят много крупных компаний, включая AMD, Apple, ARM, Intel, Nvidia, Sony Computer Entertainment, Sun Microsystems и другие.

Если в кратце, то OpenCL кроссплатформенная технология распараллеливания вычислений на GPU $\Rightarrow$ без разницы, графический адаптер какой фирмы установлен на машине, будь то Nvidia или AMD, код на OpenCL будет выполняться аналогично на любых устройствах. В этом, несомненно, заключается его преимущество. Существует также и обратная сторона медали, для того, чтобы вызвать код на OpenCL необходимо сначала перенести все данные в буфер, с которым будет оперировать

непосредственно процессор GPU. Время затрачиваемое на запись и считывание может существенно ухудшить показатели производительности, если нерационально применять распараллеливание на GPU.

Также стоит сказать и об альтернативной технологии – CUDA, разработка компании Nvidia, которая в технологическом плане идет на шаг впереди OpenCL, но по сравнению с ней она является платформозависимой.

Алгоритм и реализация:

В коде, который будет приведен ниже, происходит инициализация аппаратуры, а именно, происходит поиск платформ, далее на нужной платформе осуществляется поиск устройства, которое будет производить вычисления. После этого создается контекст и очередь задач. Контекст представляет собой пространство, в котором будут храниться все наши данные (буфферы). В очередь задач будут добавляться задачи на выполнение. Также нам потребуется создать кернелы, которые будет отвечать за отдельную функцию в коде на OpenCL, т.е. кернелы являются прослойкой между исходным кодом на C/C++ и кодом на OpenCL. Код на языке OpenCL компилируется в *RUNTIME* режиме, его можно хранить как отдельным файлом, так и строкой внутри кода на C/C++.

```
1  void RotateMethodGPU() {
2      string plotFileGPU = "plotDataGPU";
3      // инициализация устройств
4      // ************************************************************
       **********************
5      cl_int              err;
6      cl_platform_id*     platforms;
7      cl_uint             num_platforms;
8      cl_device_id*       devices;
9      cl_uint             num_devices;
10     cl_context          context;
11     cl_command_queue    command_queue;
12     cl_program          program                 = NULL;
13     cl_kernel           kernelFindMaxAbs        = NULL;
14     cl_kernel           kernelMultMatrix        = NULL;
15     cl_kernel           kernelComputeNorm       = NULL;
16     cl_kernel           kernelTranspose         = NULL;
17     cl_context_properties platforms_properties[3];
18
19     /* получить доступные платформы */
20     err = clGetPlatformIDs(0, NULL, &num_platforms);
```

```
21      checkRet(err, __LINE__);
22      platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id) *
        num_platforms);
23      err = clGetPlatformIDs(num_platforms, platforms, NULL);
24      checkRet(err, __LINE__);
25      /* получить доступные устройства */
26      // цикл нужен если платформ с GPU > 1
27      //for (size_t i = 0; i < num_platforms; ++i) {
28          platforms_properties[0] = (cl_context_properties)
        CL_CONTEXT_PLATFORM;  // indicates that next element is platform
29          platforms_properties[1] = (cl_context_properties)platforms[0
        ];  // platform is of type cl_platform_id
30          platforms_properties[2] = (cl_context_properties)0;   //
        last element must b
31          err = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, 0,
        NULL, &num_devices);
32          checkRet(err, __LINE__);
33          devices = (cl_device_id*)malloc(sizeof(cl_device_id) *
        num_devices);
34          err = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU,
        num_devices, devices, NULL);
35          checkRet(err, __LINE__);
36      //}
37      size_t param_sz;
38      cl_uint param_int, valFPS = 0;
39      char* param_string;
40      for (size_t i = 0; i < num_devices; ++i) {
41          err = clGetDeviceInfo(devices[i], CL_DEVICE_ADDRESS_BITS,
        sizeof(cl_uint), (void*)&param_int, NULL);
42          checkRet(err, __LINE__);
43          cout << "ADDRESS␣BITS␣=␣" << param_int << endl;
44          err = clGetDeviceInfo(devices[i],
        CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), (void*)&param_int,
        NULL);
45          checkRet(err, __LINE__);
46          valFPS = param_int;
47          err = clGetDeviceInfo(devices[i],
        CL_DEVICE_MAX_CLOCK_FREQUENCY, sizeof(cl_uint), (void*)&param_int
        , NULL);
48          checkRet(err, __LINE__);
49          valFPS *= param_int;
50          err = clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 0, NULL,
         &param_sz);
```

```
51          checkRet(err, __LINE__);
52          param_string = (char*)malloc(sizeof(char) * param_sz);
53          err = clGetDeviceInfo(devices[i], CL_DEVICE_NAME, param_sz,
    (void*)param_string, NULL);
54          checkRet(err, __LINE__);
55          cout << "FPS = " << valFPS << " | " << "device name: " <<
    param_string << endl;
56          free(param_string);
57          err = clGetDeviceInfo(devices[i], CL_DEVICE_VENDOR, 0,
    NULL, &param_sz);
58          checkRet(err, __LINE__);
59          param_string = (char*)malloc(sizeof(char) * param_sz);
60          err = clGetDeviceInfo(devices[i], CL_DEVICE_VENDOR, param_sz
    , (void*)param_string, NULL);
61          checkRet(err, __LINE__);
62          cout << "vendor: " << param_string << endl;
63          cout << "*********************************" << endl;
64          free(param_string);
65      }
66      // 3-ий аргумен = 0, тк.. всего одно устройство
67      context = clCreateContextFromType(platforms_properties,
    CL_DEVICE_TYPE_GPU, NULL, NULL, &err);
68      checkRet(err, __LINE__);
69      command_queue = clCreateCommandQueue(context, devices[0], 0, &
    err);
70      checkRet(err, __LINE__);
71
72      FILE* fp;
73      const char* fileName = "./test.cl";
74      char *source_str;
75      size_t source_size;
76
77      try {
78          fp = fopen(fileName, "r");
79          if (!fp) {
80              std::cerr << "Failed to load kernelMultMatrix." << std::
    endl;
81              exit(1);
82          }
83
84          source_str = (char *)malloc(MAX_SOURCE_SIZE);
85          source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
86          fclose(fp);
```

```
87      }
88      catch (int a) {
89          printf ("%d", a);
90      }
91      /* создать бинарник из кода программы */
92      program = clCreateProgramWithSource(context, 1, (const char **)&
        source_str, (const size_t *)&source_size, &err);
93      checkRet(err, __LINE__);
94      /* скомпилировать программу */
95      err = clBuildProgram(program, 1, devices, NULL, NULL, NULL
        );
96      // build failed
97      if (err != CL_SUCCESS) {
98          cl_build_status status;
99          size_t logSize;
100         // check build error and build status first
101         clGetProgramBuildInfo(program, devices[0],
        CL_PROGRAM_BUILD_STATUS,
102                 sizeof(cl_build_status), &status, NULL);
103
104         // check build log
105         clGetProgramBuildInfo(program, devices[0],
106                 CL_PROGRAM_BUILD_LOG, 0, NULL, &logSize);
107         char* programLog = (char*)malloc((logSize + 1) * sizeof(char
        ));
108         clGetProgramBuildInfo(program, devices[0],
109                 CL_PROGRAM_BUILD_LOG, logSize + 1, programLog,
        NULL);
110         printf("Build␣failed;␣error=%d,␣status=%d,␣programLog:nn%s",
111                 err, status, programLog);
112         free(programLog);
113     }
114     // ************************************************************
        **********************
115     cl_float EPS = 0.01;
116     cl_float EPSk;
117
118     /*  создать кернел */
119     kernelMultMatrix = clCreateKernel(program, "MultMatrix", &err);
120     checkRet(err, __LINE__);
121     kernelFindMaxAbs = clCreateKernel(program, "FindMaxABS", &err);
122     checkRet(err, __LINE__);
123     kernelComputeNorm = clCreateKernel(program, "ComputeNorm", &err)
```

```
    ;
124     checkRet(err, __LINE__);
125     kernelTranspose = clCreateKernel(program, "TransposeMatrix", &
        err);
126     checkRet(err, __LINE__);
```

Из заключительных строчек листинга можно заметить, что всего мною было распараллелено четыре основных типа вычислений:

1. Перемножение матриц
2. Поиск максимума
3. Вычисление нормы матрицы
4. Транспонирование матрицы

Давайте теперь расмотрим код каждого из них по отдельности, а затем я приведу исходный код на C/C++ целиком.

MultMatrix:

Принцип по которому я делал перемножение матриц таков: бралась i-ая строка матрицы $\mathbb{A}$ и j-ый столбец матрицы mB и перемножались, а результат записывался в ячейку $\mathbb{C}[i, j]$. Мною было эксперементально проверено, что такой способ гораздо эффективнее, нежели осуществлять перемножением следующим образом: берем i-ую строку матрицы $\mathbb{A}$ и в цикле пробегаем по все столбцам матрицы $\mathbb{B}$ ($j \in \overline{[1, N]}$) и в соответствующий элемент $\mathbb{C}[i, j]$ все то записываем. Т.е. моё решение использует двумерную размерность, а данное одномерную, в связи с чем ресурсы GPU задействуются не в полной мере.

Забыл отметить один момент, N-мерные массивы OpenCL не поддерживает, либо мои познания в нем недостаточны, поэтому мне пришлось проецировать двумерный массив на одномерный.

```
1  kernel void MultMatrix(global float* mA,
2                         global float* mB,
3                         global float* mC,
4                         private unsigned int sz)
5  {
6      int row = get_global_id(0);
7      int col = get_global_id(1);
8      float temp = 0;
9      for (int k = 0; k < sz; ++k)
```

```
    {
        temp += mA[row * sz + k] * mB[k * sz + col];
    }
    mC[row * sz + col] = temp;
}
```

Ничего сверхестественного в данной реализации я не вижу, стоит только сказать, что в интернете есть способ, который позволяет ещё быстрее перемножать матрицы (поблочно), некий аналог метода Штрассена, но уже для параллельных вычислений.

FindMaxABS:

Максимальный по модулю элемент ищется следующим образом:

- Мы запускаем N потоков, где N – кол-во строк матрицы $\mathbb{A}$
- В каждой из строк мы находим локальный максимум, т.е. максимум в данной строке
- Устанавливаем барьер, дойдя до которого каждый из потоков остановиться и не будет продолжать дальнейших вычислений, пока все потоки не дойдут до этого барьера
- Выбираем любой из N потоков (я выбрал самый первый, нулевой) и в нем уже ищем глобальный максимум, т.е. максимум из локальных максимумов

Стоит отметить, что во время всех этих поисков нам нужно хранить не только максимум, но и позицию этого максимума (строку и столбец).

```
kernel void FindMaxABS(global float* a,
                       global float* maxElem,
                       global int* posMax,
                       private unsigned int sz)
{
    int pos = get_global_id(0);
    int row = pos * sz;
    float _max = -1;
    int posI, posJ;
    for (int i = 0; i < sz; ++i) {
        if (i == pos) continue;
        if (_max < fabs(a[row + i])) {
            _max = fabs(a[row + i]);
            posJ = i;
```

```
        }
    }
    maxElem[pos] = _max;
    posMax[pos] = posJ;
    barrier(CLK_GLOBAL_MEM_FENCE);
    if (!pos) {
        _max = -1;
        posJ = -1;
        for (int i = 0; i < sz; ++i) {
            if (_max < maxElem[i]) {
                _max = maxElem[i];
                posI = i;
                posJ = posMax[i];
            }
        }
        posMax[0] = posI;
        posMax[1] = posJ;
    }
}
```

ComputeNorm:

По своей структуре вычисление нормы схоже с поиском максимума, мы пробегаемся по каждой строке, не включая диагональные элементы, суммируем квадраты элементов $\mathbb{A}[i, j]$, а затем, просто суммируем все эти локальные значения в одно глобальное, опять-таки при помощи барьера.

```
kernel void ComputeNorm(global float* a,
                        global float* res,
                        private unsigned int sz)
{
    int pos = get_global_id(0);
    int row = pos * sz;
    float sum = 0;
    for (int i = 0; i < sz; ++i) {
        if (pos == i) continue;
        sum += a[row + i] * a[row + i];
    }
    res[pos] = sum;
    barrier(CLK_GLOBAL_MEM_FENCE);
    if (!pos) {
        sum = 0;
        for (int i = 0; i < sz; ++i) {
```

```
            sum += res[i];
        }
        res[0] = sum;
    }
}
```

TransposeMatrix:

Принцип прост, мы выбираем i-ую строку в матрице, пробегаемся по ней, и все значения записываем в i-ый столбец результирующей матрицы.

```
kernel void TransposeMatrix(global float* a,
                            global float* res,
                            private unsigned int sz)
{
    int pos = get_global_id(0);
    int row = pos * sz;
    for (int i = 0; i < sz; ++i) {
        res[pos + i * sz] = a[row + i];
    }
}
```

Исходный код:

Я посторался максимально информативно прокомментировать исходник, поэтому остается пожелать Вам приятного чтения.

Основные этапы запуска функции на OpenCL:

1. Создание буфферов
2. Запись значений в буффер
3. Установка аргументов кернелу
4. Запуск
5. Ожидание выполнения и считывание из буфферов

```
        cl_uint szMatrix = cnt;
        size_t global_work_size[2] = { szMatrix, szMatrix };

        // квадратная матрица, симметричная
```

```
5       matrixA         = (cl_float*)malloc(sizeof(cl_float) *
    szMatrix * szMatrix);
6       matrixB         = (cl_float*)malloc(sizeof(cl_float) *
    szMatrix * szMatrix);
7       matrixC         = (cl_float*)malloc(sizeof(cl_float) *
    szMatrix * szMatrix);
8       matrixRotate    = (cl_float*)malloc(sizeof(cl_float) *
    szMatrix * szMatrix);
9       matrixRotateT   = (cl_float*)malloc(sizeof(cl_float) *
    szMatrix * szMatrix);
10
11      cl_mem      buffMatrixA      = NULL;
12      cl_mem      buffMatrixB      = NULL;
13      cl_mem      buffMatrixC      = NULL;
14      // инициализация симметричной матрицы
15      for (cl_uint i = 0; i < szMatrix; ++i) {
16          for (cl_uint j = i; j < szMatrix; ++j) {
17              matrixA[i * szMatrix + j] = matrixA[j * szMatrix + i
    ] = rand() % 301 - 150;
18          }
19      }
20      EPSk = Off(matrixA, szMatrix, kernelComputeNorm,
    command_queue, context, global_work_size);
21      while (EPSk >= EPS) {
22          for (cl_uint i = 0; i < szMatrix; ++i) {
23              for (cl_uint j = 0; j < szMatrix; ++j) {
24                  matrixRotate[i * szMatrix + j] = (i == j  ? 1 :
     0);
25              }
26          }
27
28          // **********************************
29          // поиск максимального элемента
30          cl_float* maxElem   = (cl_float*)malloc(sizeof(cl_float)
     * szMatrix);
31          cl_int* maxPos      = (cl_int*)malloc(sizeof(cl_int) *
    max(1u * 2, szMatrix));
32          cl_mem buffMaxPos       = NULL;
33          cl_mem buffMaxElem      = NULL;
34          buffMatrixA  = clCreateBuffer(context, CL_MEM_READ_ONLY,
     szMatrix * szMatrix * sizeof(cl_float), NULL, &err);
35          checkRet(err, __LINE__);
36          buffMaxElem  = clCreateBuffer(context, CL_MEM_READ_WRITE
```

```
, szMatrix * sizeof(cl_float), NULL, &err);
37              checkRet(err, __LINE__);
38              buffMaxPos  = clCreateBuffer(context, CL_MEM_READ_WRITE,
 max(1u * 2, szMatrix) * sizeof(cl_int), NULL, &err);
39              checkRet(err, __LINE__);
40              err = clEnqueueWriteBuffer(command_queue, buffMatrixA,
CL_FALSE, 0,
41                  szMatrix * szMatrix * sizeof(cl_float), matrixA, 0,
NULL, NULL);
42              checkRet(err, __LINE__);
43              err = clSetKernelArg(kernelFindMaxAbs, 0, sizeof(cl_mem)
, (void *)&buffMatrixA);
44              checkRet(err, __LINE__);
45              err = clSetKernelArg(kernelFindMaxAbs, 1, sizeof(cl_mem)
, (void *)&buffMaxElem);
46              checkRet(err, __LINE__);
47              err = clSetKernelArg(kernelFindMaxAbs, 2, sizeof(cl_mem)
, (void *)&buffMaxPos);
48              checkRet(err, __LINE__);
49              err = clSetKernelArg(kernelFindMaxAbs, 3, sizeof(cl_uint
), (void *)&szMatrix);
50              checkRet(err, __LINE__);
51              err = clEnqueueNDRangeKernel(command_queue,
kernelFindMaxAbs, 1, NULL, global_work_size, NULL, 0, NULL,
 NULL);
52              checkRet(err, __LINE__);
53              err = clEnqueueReadBuffer(command_queue, buffMaxPos,
CL_TRUE, 0, max(1u * 2, szMatrix) * sizeof(cl_int), maxPos, 0,
NULL, NULL);
54              checkRet(err, __LINE__);
55              clReleaseMemObject(buffMatrixA);
56              clReleaseMemObject(buffMaxElem);
57              clReleaseMemObject(buffMaxPos);
58              // ***********************************
59
60              // ***********************************
61              // задаем угол вращения и матрицу вращения
62              cl_int i = maxPos[0], j = maxPos[1];
63              free(maxElem);
64              free(maxPos);
65              cl_float angel = fabs(matrixA[i * szMatrix + i] -
matrixA[j * szMatrix + j])
66                  < 0.01 * EPS ? M_PI / 4 :
```

```
67                  (0.5 * atan(2.0 * matrixA[i * szMatrix + j]
68                      / (matrixA[i * szMatrix + i] - matrixA[j *
    szMatrix + j])));
69          matrixRotate[i * szMatrix + i] = matrixRotate[j *
    szMatrix + j] = cos(angel);
70          matrixRotate[j * szMatrix + i] = -(matrixRotate[i *
    szMatrix + j] = -sin(angel));
71          // *********************************
72
73          // *********************************
74          // транспанируем матрицу поворота
75          buffMatrixA  = clCreateBuffer(context, CL_MEM_READ_ONLY,
     szMatrix * szMatrix * sizeof(cl_float), NULL, &err);
76          checkRet(err, __LINE__);
77          buffMatrixB  = clCreateBuffer(context, CL_MEM_WRITE_ONLY
    , szMatrix * szMatrix * sizeof(cl_float), NULL, &err);
78          checkRet(err, __LINE__);
79          err = clEnqueueWriteBuffer(command_queue, buffMatrixA,
    CL_FALSE, 0,
80              szMatrix * szMatrix * sizeof(cl_float), matrixRotate
    , 0, NULL, NULL);
81          checkRet(err, __LINE__);
82          err = clSetKernelArg(kernelTranspose, 0, sizeof(cl_mem),
     (void *)&buffMatrixA);
83          checkRet(err, __LINE__);
84          err = clSetKernelArg(kernelTranspose, 1, sizeof(cl_mem),
     (void *)&buffMatrixB);
85          checkRet(err, __LINE__);
86          err = clSetKernelArg(kernelTranspose, 2, sizeof(cl_uint)
    , (void *)&szMatrix);
87          checkRet(err, __LINE__);
88          err = clEnqueueNDRangeKernel(command_queue,
    kernelTranspose, 1, NULL, global_work_size, NULL, 0, NULL,
    NULL);
89          checkRet(err, __LINE__);
90          err = clEnqueueReadBuffer(command_queue, buffMatrixB,
    CL_TRUE, 0,
91              szMatrix * szMatrix * sizeof(cl_float),
    matrixRotateT, 0, NULL, NULL);
92          checkRet(err, __LINE__);
93          clReleaseMemObject(buffMatrixA);
94          clReleaseMemObject(buffMatrixB);
95          // *********************************
```

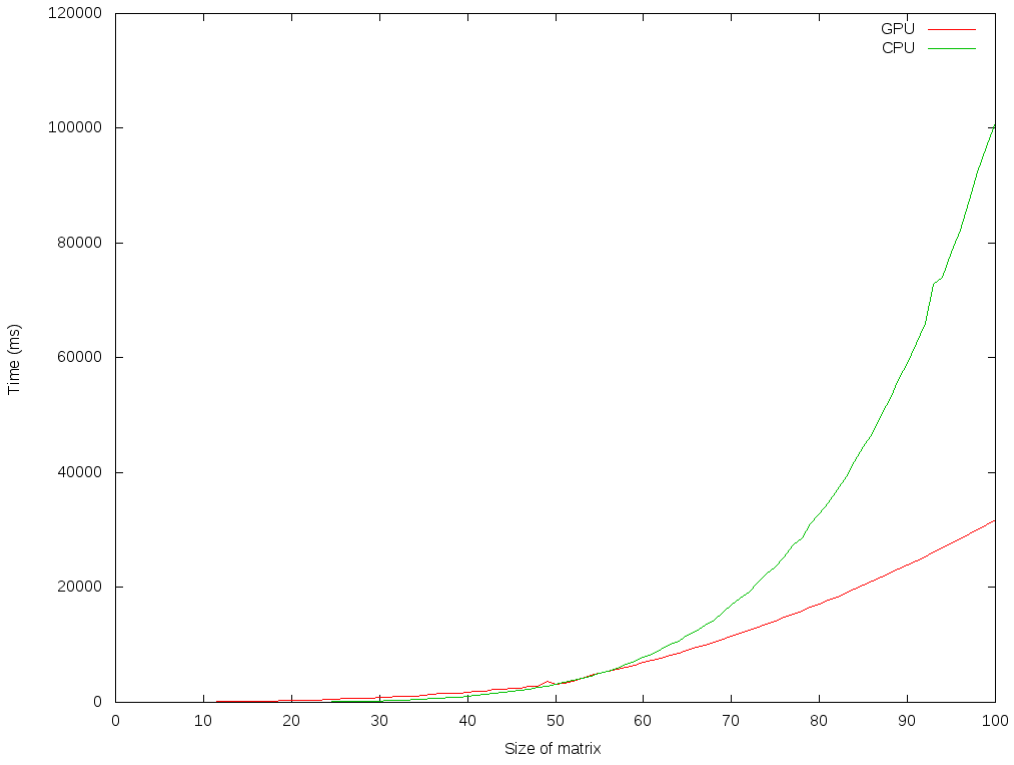
```
96
97              // **********************************
98              // Выполняем в два этапа перемножение матрицы R'␣*␣A␣*␣R
    ,
99  ␣␣␣␣␣␣␣␣␣␣␣␣//␣R'-транспонированная матрица поворота
100             buffMatrixA  = clCreateBuffer(context, CL_MEM_READ_ONLY,
     szMatrix * szMatrix * sizeof(cl_float), NULL, &err);
101             checkRet(err, __LINE__);
102             buffMatrixB  = clCreateBuffer(context, CL_MEM_READ_ONLY,
     szMatrix * szMatrix * sizeof(cl_float), NULL, &err);
103             checkRet(err, __LINE__);
104             buffMatrixC  = clCreateBuffer(context, CL_MEM_WRITE_ONLY
    , szMatrix * szMatrix * sizeof(cl_float), NULL, &err);
105             checkRet(err, __LINE__);
106             err = clEnqueueWriteBuffer(command_queue, buffMatrixA,
    CL_FALSE, 0,
107                 szMatrix * szMatrix * sizeof(cl_float),
    matrixRotateT, 0, NULL, NULL);
108             checkRet(err, __LINE__);
109             err = clEnqueueWriteBuffer(command_queue, buffMatrixB,
    CL_FALSE, 0,
110                 szMatrix * szMatrix * sizeof(cl_float), matrixA, 0,
    NULL, NULL);
111             checkRet(err, __LINE__);
112             err = clSetKernelArg(kernelMultMatrix, 0, sizeof(cl_mem)
    , (void *)&buffMatrixA);
113             checkRet(err, __LINE__);
114             err = clSetKernelArg(kernelMultMatrix, 1, sizeof(cl_mem)
    , (void *)&buffMatrixB);
115             checkRet(err, __LINE__);
116             err = clSetKernelArg(kernelMultMatrix, 2, sizeof(cl_mem)
    , (void *)&buffMatrixC);
117             checkRet(err, __LINE__);
118             err = clSetKernelArg(kernelMultMatrix, 3, sizeof(cl_uint
    ), (void *)&szMatrix);
119             checkRet(err, __LINE__);
120             err = clEnqueueNDRangeKernel(command_queue,
    kernelMultMatrix, 2, NULL, global_work_size, NULL, 0, NULL,
     NULL);
121             checkRet(err, __LINE__);
122             err = clEnqueueReadBuffer(command_queue, buffMatrixC,
    CL_TRUE, 0,
123                 szMatrix * szMatrix * sizeof(cl_float), matrixC, 0,
```

```
NULL, NULL);
124             checkRet(err, __LINE__);
125             err = clEnqueueWriteBuffer(command_queue, buffMatrixA,
    CL_FALSE, 0,
126                 szMatrix * szMatrix * sizeof(cl_float), matrixC, 0,
    NULL, NULL);
127             checkRet(err, __LINE__);
128             err = clEnqueueWriteBuffer(command_queue, buffMatrixB,
    CL_FALSE, 0,
129                 szMatrix * szMatrix * sizeof(cl_float), matrixRotate
    , 0, NULL, NULL);
130             err = clSetKernelArg(kernelMultMatrix, 0, sizeof(cl_mem)
    , (void *)&buffMatrixA);
131             checkRet(err, __LINE__);
132             err = clSetKernelArg(kernelMultMatrix, 1, sizeof(cl_mem)
    , (void *)&buffMatrixB);
133             checkRet(err, __LINE__);
134             err = clSetKernelArg(kernelMultMatrix, 2, sizeof(cl_mem)
    , (void *)&buffMatrixC);
135             checkRet(err, __LINE__);
136             err = clSetKernelArg(kernelMultMatrix, 3, sizeof(cl_uint
    ), (void *)&szMatrix);
137             checkRet(err, __LINE__);
138             err = clEnqueueNDRangeKernel(command_queue,
    kernelMultMatrix, 2, NULL, global_work_size, NULL, 0, NULL,
     NULL);
139             checkRet(err, __LINE__);
140             err = clEnqueueReadBuffer(command_queue, buffMatrixC,
    CL_TRUE, 0,
141                 szMatrix * szMatrix * sizeof(cl_float), matrixA, 0,
    NULL, NULL);
142             checkRet(err, __LINE__);
143             clReleaseMemObject(buffMatrixA);
144             clReleaseMemObject(buffMatrixB);
145             clReleaseMemObject(buffMatrixC);
146             // ***********************************
147
148             // ***********************************
149             // Вычисляем норму матрицы
150             EPSk = Off(matrixA, szMatrix, kernelComputeNorm,
    command_queue, context, global_work_size);
151             // ***********************************
152         }
```

```
153         free(matrixA);
154         free(matrixB);
155         free(matrixC);
156         free(matrixRotate);
157         free(matrixRotateT);
```

Вывод:

Выполняя данный курсовой проект, я смог не только повторить и закрепить знания и опыт работы с методом Якоби, но так же познакомился с передовыми технологиями параллельного вычисления на GPU, которые очень активно применяются как в научной сфере, так и в коммерческой.



На графике отчетливо видно, что производительность вычислений на GPU растет с ростом размерности матрицы. На данном примере прослеживается трёхкратное преобладание в скорости работы алгоритма на GPU при размерности задачи N = 100, т.е. матрицы $\mathbb{A}_{100\times100}$