Final exam

Parallel Computing

Andrea Boido

March 25, 2019

Abstract

I introduce briefly the main aspects and advantages of the parallelization in computational science, in particular referring to the OpenCL framework. Then I explain and analyse an application of this technique to the eigenproblem and eventually to the numerical solution of the 2D Schrödinger equation.

1 Theory

1.1 Parallel computing

In the last years the increasing difficulties that emerged in designing and building faster and faster (in terms of frequency) processors forced the us to go towards another way to improve the computational power of computers: the parallelization. It mainly consists in the splitting of the CPUs in subpieces called *compute units* (or *cores*) that have the capability of working semi-independently (it will be clear what I mean with "semi" afterwards) and simultaneously. In this way different parts of a given algorithm can be executed by different compute units at the same time (if the algorithm allows such a splitting), which results typically in a shorter time to complete the task. This is the same kind of approach that have been used for many years and on a much bigger scale in supercomputers, even though there are big differences in the concrete implementation of the idea. Nevertheless the basic features that we will consider are mostly common.

Among this scenario a key role has been played recently also by the development of the GPGPU, standing for *General-purpose computing on graphics processing units*, which is basically the practice of exploiting the characteristic features of a GPU to perform computing tasks that are traditionally carried on by a CPU. The main advantage is that GPUs are built specifically to be as fast as possible in graphics processing and thus if we are able to translate our data of interest into graphical form and make the GPU handle them, it is likely that we achieve a great speed-up. Moreover GPUs generally have a higher number of compute units compared to CPUs, hence we can furtherly take advantage of the parallelization paradigm. Compared to CPUs, they also provide an impressive power efficiency.

It is clear that these techniques can be very useful in the field of scientific computing and, for what concerns us, in particular in computational physics where the models are becoming increasingly complex and the simulations resources-demanding. Especially the use of GPUs is becoming more and more widespread even inside the supercomputers, so that GPUs producers like AMD and NVIDIA have built graphics cards designed specifically for the purpose of doing computations (for instance the series *Tesla* by NVIDIA).

Obviously the parallelization carries also some downsides; the greater is probably the higher complexity both of the algorithms, which have to be optimized properly for taking advantage of this approach, and also of the coding phase, which requires the developer to manually handle many more tasks compared to traditional programs (called *single-thread* in the following), for instance the synchronization between the compute units, memory management and data transfers. Writing a good parallelized program is much more difficult than writing a good single-thread one. However I will speak more concretely about some of these aspects in the following.

The most famous and used mainstream software for parallel computing are CUDA, a platform for GPGPU on NVIDIA graphics cards designed to work with C, C++, and Fortran, OpenMP an API[1] for multiprocessing programming again in C, C++, and Fortran (only on CPUs), and OpenCL, which is the main subject of this report. In the next section I will briefly introduce some basic aspects of the latter one.

1.2 Brief introduction to OpenCL

OpenCL is an open source standard for parallel programming developed initially by Apple and maintained by the consortium Khronos Group. Its main characteristic is that the same OpenCL code can run across *heterogeneus* platforms: CPUs, GPUs but also other kinds of data processors (for instance FPGAs[2]); the work of interfacing with the different hardware is done mostly by the OpenCL library, leaving the programmer the only job of writing a good code without worrying too much about the device it will run on. However in the following I will focus on standard computers.

The OpenCL standard defines a set of specifications for a C/C++ library that have to be implemented concretely by hardware vendors. The various implementations of OpenCL are called *platforms* and OpenCL allows one to use even more than one platform inside the same program, so that one can take full advantage of all the computational power he has in its computer. For instance if a computer has an Intel CPU and an AMD GPU, one can write a program using both of them to do computations simultaneously! The only limit to such a broad parallelization is the algorithm itself.

Each OpenCL program is composed by two main parts: the *host program* and the *kernels*. The former is a code written in C or C++ that runs on the CPU and takes care of initializing the OpenCL environment and managing the overall execution of the algorithm. Kernels are kinds of functions written in the OpenCL language, which is actually quite similar to C, that run on a chosen target device which can be a GPU or even the CPU itself; they are dispatched to the device (or devices) by the host together with the data of interest to be elaborated.

On the host side there are a couple of fundamental structures defined in the OpenCL standard that need to be understood in order to learn how to code. They are listed below; refer to figure 1 for a pictorial representation.

- *Context* - A logical container inside which the host puts a set of devices selected to work together. It is a structure that becomes useful when the code must run on complex machines with distributed processors, while when using a standard computer one typically initializes a single context. Note that devices running on different platforms cannot be in the same context, so it may be useful to define more than one context also in the latter case if one wants to run OpenCL on a CPU and a GPU from different vendors.

[1] Application Programming Interface.

[2] Field-programmable gate array; they are integrated circuits designed for specific applications.

- *Command queue* - The name is rather self-explanatory: it is the structure that allows the host to tell devices what to do. Each device has his own command queue and the instructions that are enqueued by the host are executed sequentially, unless otherwise specified by the host. Through the command queue the host mainly deploys kernels to devices and transfers data from/to them.

- *Program* - A container storing a set of kernels. This structure is necessary since kernels have to be compiled run-time during the execution of the code and thus many of them are grouped inside a single program that is built just once. Clearly the compiling process depends on the particular platform and device one is using, so the compiling function requires to have gained informations about them in advance, which is done with some simple functions included in the OpenCL library. Also we can pass optional arguments to the compiling function through flags.

In addition to this, the OpenCL standard defines additional data types and data structures that must be used to interface properly with its functions.

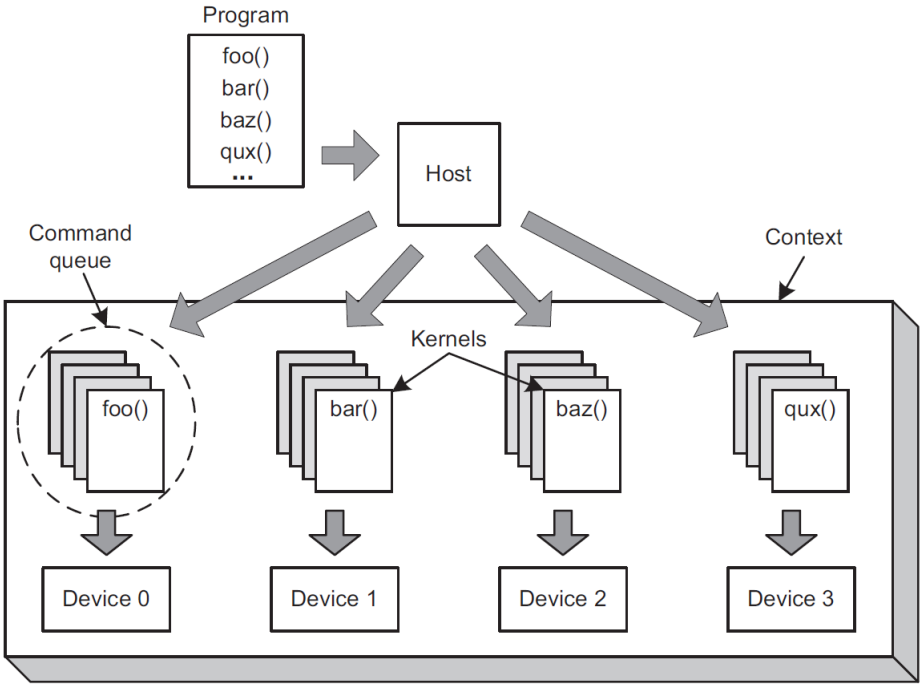


Figure 1: *Host managing the execution (picture taken from the book "OpenCL in action").*

Switching to the kernel side, one has to understand how kernels are executed on the devices and this requires to understand how a device is structured within the OpenCL framework. Refer to figure 2 for a pictorial representation of the following discussion. As I already said, physically each device is divided into compute units but also each of them has multiple processing elements[3]. The data being processed is stored in various different memory location, mainly RAM, cache, VRAM, registers, depending also on what kind of device is being used, and each memory has a different capacity and speed (typically the higher the capacity, the lower the speed and viceversa). From the more abstract point of

[3]This is thanks to the SIMD architecture, meaning "Single instruction, multiple data" that allows a processor to execute simultaneously the same task with different input data. Here one should distinguish between CPUs and GPUs since they are quite different, but I will not go into further details of hardware-related topics.

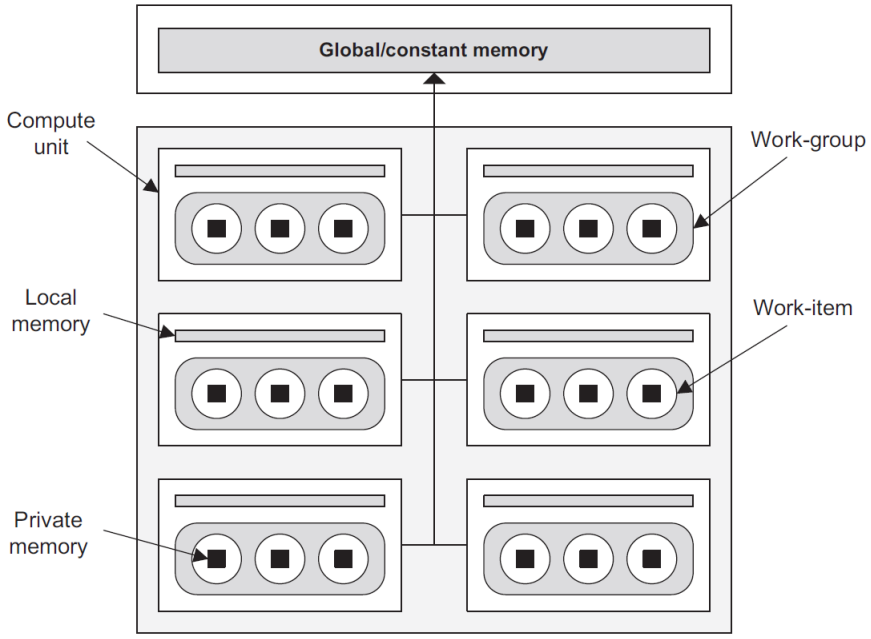


Figure 2: *Device model (picture taken from the book "OpenCL in action").*

view of OpenCL what happens is that, at the time of deploying a kernel to a device, the host fixes also the number of its instances that have to be executed; this number is called *global size* and each instance is called *work item*. So each work item executes the kernel once for a given set of input parameters, in general different from those of the other work items. Physically a work item is executed by a single processing element lying inside a compute unit of the device. Moreover a bunch of work items can be grouped together to form a *work group*; a given work group will be executed by a single compute unit and the number of work items inside it is called *local size*. The maximum possible local size is established by the device producer, while the number of work groups and the global size can be arbitrarily big, though one has to keep in mind that each compute unit can execute a single work group at a time. A work item is thus identified by both a *global id* and a *local id* whereas a work group is identified by the *work group id*. For what concerns the memory, from the OpenCL point of view we have three kinds of memory:

- *Global memory* - A memory shared among all the work items, which is usually the biggest but also the slowest. Most of the data the host transfers to the device goes into the global memory (exceptions can be only single variables that go directly into the private memory).
- *Local memory* - A memory common to all work items inside a work group. It should be sensibly faster than global memory but with less capacity and the host cannot access it directly. It is useful to store intermediate results.
- *Private memory* - The own memory of each work item. It should be the fastest of all but it is even smaller than local memory and can be used only by the single work item.

The physical partition of such memories is not easy to understand in general and it highly depends on the device producer; anyway to fix ideas for example we can think of GPUs as storing global memory inside VRAM and local/private memory inside the cache and the registers. The last important feature that has to be understand is that an uncontrolled

access to shared memory (global and private) could result in bugs inside the code. Indeed the programmer has almost no control on the timing of the events that happens on a device, thus for instance it may be possible that a work item accesses a value in the local memory that is not the correct one since another work item is still processing it. In order to avoid such problems one can implement *barriers* inside a kernel which are instructions that tell the device to wait until all the work items inside a work group have done their job; this way one can synchronize the execution within work groups. Unfortunately up to now it is impossible to synchronize work items belonging to different work groups.

Once one has understood the basic principles of working of the OpenCL framework, it is pretty clear what is the direction to follow in writing a code. Whenever an iterative cycle of definite size occurs inside an algorithm, it can be replaced by a kernel executing (hopefully simultaneously) a work item for each iteration, so that the execution time will be dramatically reduced, at least in principle. Obviously this approach cannot be always followed since a wide class of iterative algorithms are structured to use at a given iteration data computed during the previous one (we will see examples of this situation in the following).

Finally I should mention that the OpenCL kernel language provides the programmer with a set of data structures and built-in functions (mainly arithmetic functions) that are highly optimized for parallelization at a lower level and should help in improving furtherly the computations speed, for examples vectors, atomic operations etc. Some of them like vectors have been slightly used in my code, nevertheless these are quite advanced tools that should not be used until having achieved a minimum of experience with OpenCL, thus I will not speak about them.

1.3 Environment specifications and settings

All the programs analysed in the following have runned on my desktop PC. It has an Intel Core i5 4670 as CPU with 8 GB of DDR3 RAM, an AMD Radeon HD 7870 with 2 GB of GDDR5 VRAM as GPU and Windows 10 version 1809 as operative system. I installed on the system both the AMD platform[4] and the Intel platform[5] so as to compare the performance of OpenCL on different devices. The Intel CPU has also an integrated GPU but unfortunately it does not support the OpenCL extension for the double type, so I could not include it in the comparison. See table 1 for more detailed technical specifications of the two OpenCL devices. Note that though the latest version of OpenCL is the 2.2, my hardware is capable of running only the 1.2.

Device	Intel Core i5 4670	AMD Radeon HD 7870
Clock	3400 MHz	1000 MHz
OpenCL version	1.2	1.2
# (compute units)	4	20
Global memory	2047 MB	2048 MB
Local memory	32 kB	32 kB
Max work group size	8192	256

Table 1: *Technical specification of the two devices used (taken from GPU Caps Viewer).*

As a compiler for C/C++ I used the Windows porting of GCC 8.1.0 provided by MinGW-W64[6].

[4] AMD Accelerated Parallel Processing
[5] Intel(R) OpenCL
[6] `https://mingw-w64.org`

I also installed *LAPACK* and its C interface *LAPACKE*[7] with the main aim of checking the results of my algorithms, and the C++ library for sparse linear algebra *ViennaCL*[8] that is capable of taking advantage of OpenCL and that I used for solving numerically the Schrödinger equation. Unfortunately I did not find any similar library for C, therefore I was forced to use also C++.

2 Matrix diagonalization

The main purpose of the following application is to write an original code using OpenCL that finds the lowest eigenpair of a given matrix (assumed to be hermitian) and to test its performance, in particular comparing it with the corresponding single-thread algorithm.

2.1 Algorithm

Despite the appearance of a rather basic task, the matrix diagonalization is not easy at all in computational science and includes lots of subtleties that prevent to identify a single method as the universally best one. Provided that the main purpose of this application was to evaluate OpenCL performances and not to write an excellent diagonalization algorithm, I ignored most of these issues and I tried to proceed straightforwardly with a relatively easy implementable solution. I individuated in the QR algorithm the best option.

The QR algorithm relies on the linear algebra statement that every real square matrix A can be written as $A = Q_0 R_0$ where Q_0 is an orthogonal matrix and R_0 is an upper triangular matrix; this is called QR decomposition. Provided that one is able to perform such a decomposition on a computer (it is not trivial how to do it, I will discuss it in the following), then he can build a succession of matrices $A_k = R_{k-1} Q_{k-1}$ with $k \in \mathbb{N}$. Then I can write:

$$A_{k+1} = R_k Q_k = Q_k^{-1} A_k Q_k = Q_k^T A_k Q_k = Q_k^T \dots Q_0^T \, A \, Q_0 \dots Q_k = (Q_0 \dots Q_k)^T \, A \, (Q_0 \dots Q_k)$$

It can be proved that this succession converges to a diagonal matrix, thus defining

$$D \equiv \lim_{k \to +\infty} A_k \qquad\qquad P \equiv \lim_{k \to +\infty} (Q_0 \dots Q_k)$$

it is clear that $D = P^T A P$ with D containing the eigenvalues and P the eigenvectors of A. Clearly in a real implementation the limit has to be intended as a convergence in the computational sense i.e. when the quantities of interest do not vary between an iteration and the other up to a given precision.

However this algorithm carries some fundamental problems:

- the convergence is quite slow, especially if eigenvalues are close to each other;
- it is computationally expensive: each iteration step requires to compute two products of $N \times N$ matrices, thus even without considering the QR decomposition each single iteration step has a complexity of $\mathcal{O}(N^3)$; assuming that the number of steps is proportional to N, which is not even assured because of the above issue, one would get an $\mathcal{O}(N^4)$ complexity;
- it requires the initial matrix to be real, which is not the case in general if one wants to use the algorithm for solving quantum mechanics problems;

[7] `https://icl.cs.utk.edu/lapack-for-windows/lapack/`
[8] `http://viennacl.sourceforge.net/`

- without including further adjustments (that necessarily increase the coding complexity) the algorithm is quite unstable and could lead to completely wrong results depending on the initial matrix.

The latter issue was ignored because I was mostly interested in computing the lowest eigenpair which proved to be quite accurate most of the times. For what concerns the first issue I tried to mitigate it by using the shifted QR algorithm which consists in applying the QR factorization to the matrices $\tilde{A}_k = A_k - \lambda\mathbb{I}$ where λ is the element (N, N) of A_k; this modification was suggested by some references found on internet but I have not observed any tangible speed improvement in the convergence. The second and the third problem can be partially addressed by doing a preliminary step. Indeed any complex matrix can be brought to a real tridiagonal matrix by means of a unitary transformation i.e. given M complex we can write $M = U^\dagger T U$ with T real tridiagonal and U unitary. Also if M is hermitian, then T will be symmetric. So if one performs this transformation on the starting hermitian matrix he solves completely the third problem. Moreover the QR decomposition of a tridiagonal matrix can be computed quite easily and with a good speed by means of Givens' rotations. These exploit the fact that any 2-component vector $v = (v_1, v_2)$ can be rotated so that its second entry vanishs:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} \|v\| \\ 0 \end{pmatrix} \qquad \text{where} \quad c = \frac{v_1}{\|v\|},\ s = \frac{v_2}{\|v\|}$$

By subsequently multiplying T with suitable block matrices each of them containing such a rotation and the identity elsewhere, one can cancel its element below the diagonal ending up with an upper triangular matrix which is exactly the R of the QR decomposition. Then Q will be simply the transpose of the product of the block diagonal matrices. This method has a complexity $\mathcal{O}(N^2)$ (N rotations and for each of them a product between block diagonal matrices that accounts for another N in this particular case) which is negligible when compared to those of matrix products. Note that since the tridiagonal form is preserved at each iteration of the QR algorithm, I could have simplified a bit also the matrix products but I decided not to do it in order to keep the code not too much complex.

The only part that is missing is how to find the proper unitary transformation that makes M tridiagonal. This job is done through the Lanczos algorithm. I followed precisely the procedure illustrated on Wikipedia[9], so I will not report here the entire algorithm. The key points are that each iteration step requires the computation of a matrix-vector multiplication, of two scalar products, and some algebraic componentwise operations on vectors. The complexity of this algorithm for a generic hermitian matrix is $\mathcal{O}(N^3)$ so, considering that it has to be used only once unlike the matrix products in the QR algorithm, it does not affect the overall complexity that much.

Now that we have the full picture we can consider how the parallelization paradigm can be implemented. Unfortunately this particular algorithm is not very suitable for a parallel approach since both the Lanczos algorithm and the QR algorithm include that kind of loops where at each iteration all the previous data is needed. The best I could do was to implement in OpenCL the single linear algebra operation taking place inside these loops, such as matrix-matrix and matrix-vector multiplications. However in section 2.3 I will show that this is enough to achieve already a consistent speed-up, while a more in-depth discussion about the OpenCL implementation of the above described algorithm is carried on in the following section.

[9] `https://en.wikipedia.org/wiki/Lanczos_algorithm`

2.2 Code development

Firstly I implemented the algorithm described in section 2.1 in a simple single-thread C code. Initially I designed it for computing the complete eigendecomposition but unfortunately the instability of the QR algorithm did not allow me to get correct results. In particular for matrices larger than some hundreds of elements some eigenvalues were counted as double eigenvalues, therefore the algorithm missed much of them. Anyhow the lowest eigenvalue (and eigenvector) was the correct one most of the times so I slightly modified the code to keep only this one. In particular I set the convergence condition only on the lowest element of the diagonal of A_k at each iteration as:

$$\left|\lambda_{min}^{(k)} - \lambda_{min}^{(k-1)}\right| < 10^{-12}$$

Once the base algorithm was ready I converted it to run parallelized. Firstly I had to initialize OpenCL and compile the program. See the code below.

```
1   //opencl initialization
2
3   FILE *f;
4   int i,j;
5   char str[40];
6   char name[40];
7   char *source_str;
8    size_t source_size;
9   sprintf(str,"-cl-std=CL1.2 -D N=%d",N);
10
11  int dev=1;
12  cl_platform_id *platforms;
13  cl_uint num_platforms;
14   cl_device_id device;
15   cl_uint num_devices;
16  cl_context context;
17  cl_command_queue queue;
18  cl_program program;
19  size_t glob_size;
20  size_t glob_size2;
21  size_t loc_size=N;
22
23  clGetPlatformIDs(5,NULL,&num_platforms);
24  platforms=(cl_platform_id*) malloc(sizeof(cl_platform_id)*num_platforms
        );
25  clGetPlatformIDs(num_platforms,platforms,NULL);
26  //printf("Number of Platforms detected: %d\n",num_platforms);
27  for(i=0;i<num_platforms;i++){
28     clGetPlatformInfo(platforms[i],CL_PLATFORM_NAME,sizeof(name),&name,
           NULL);
29     //printf("%s\n",name);
30  }
31
32
33  if(dev==1){
34     clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,1,&device,&
           num_devices);
35     for(i=1;loc_size>256;i++){
36        if(loc_size%i==0)
37           loc_size=N/i;
38     }
39     //printf("\nAMD GPU selected\n");
40  }
```

```
41    else if(dev==2){
42       clGetDeviceIDs(platforms[1],CL_DEVICE_TYPE_CPU,1,&device,&
             num_devices);
43       for(i=1;loc_size>8192;i++){
44          if(loc_size%i==0)
45             loc_size=N/i;
46       }
47       //printf("\nIntel CPU selected\n");
48    }
49
50    context=clCreateContext( NULL, 1,&device, NULL, NULL, NULL);
51    queue=clCreateCommandQueueWithProperties(context,device,NULL,NULL);
52
53     f=fopen("kernels.cl", "r");
54     source_str = (char*)malloc(MAX_SOURCE_SIZE);
55     source_size = fread( source_str, 1, MAX_SOURCE_SIZE, f);
56     fclose(f);
57    program=clCreateProgramWithSource(context,1,(const char **)&source_str
          ,(const size_t *)&source_size,NULL);
58    clBuildProgram(program,1,&device,str,NULL,NULL);
59    free(source_str);
60
61    cl_kernel lanczos1 = clCreateKernel(program, "lanczos1", NULL);
62    cl_kernel lanczos2 = clCreateKernel(program, "lanczos2", NULL);
63    cl_kernel transpose = clCreateKernel(program, "transpose", NULL);
64    cl_kernel mat_mul = clCreateKernel(program, "mat_mul", NULL);
65    cl_kernel transfer = clCreateKernel(program, "transfer", NULL);
```

The main aspects to notice are the following.

- At line 9 I specify the compilation flags; in particular I tell the compiler to use OpenCL 1.2, which is the latest supported by my hardware, and I pass the input dimension N as a parameter to the kernels.
- At lines 23-30 I get informations about the installed platforms.
- At lines 33-48 I get informations about the device running in a given platform and I set the dimension of the work group depending on its capabilities and on the input parameter N; in particular I will use both the CPU and the GPU (separately) in the tests.
- At line 50 I create the OpenCL context for the selected device.
- At line 51 I create the command queue for the selected device.
- At lines 53-59 I read the source of the kernels from the file "kernels.cl" into a string and I build the program.
- At lines 61-65 I create the five kernels that I will use in the rest of the program.

The five kernels I have written are functions that replace some parts of the original single-thread C code and are executed in parallel on the selected device. In particular their jobs are the following:

- *lanczos1*: computes the matrix-vector multiplication and some componentwise operation on the vectors; N work items, each of them computing one component of the resulting vector;

- *lanczos2*: similar to the previous but it does another part of the algorithm and it does not compute any matrix-vector product;
- *transpose*: computes the transpose of a matrix; N^2 work items, one for each element of the resulting matrix;
- *mat_mul*: computes the product of two matrices; N^2 work items, one for each element of the resulting matrix;
- *transfer*: copies the diagonal and the subdiagonal of a matrix inside two arrays; N work items, one for each element of the diagonal.

Notice that I had to split the Lanczos algorithm in two kernels since they are separated by the computation of a scalar product which is difficult to parallelize. I tried to implement it but I ended up with numerical instability and I was not able to understand where it came from, thus I preferred to let the host compute these scalar products. Also in these two kernels I had to deal with complex numbers that are not built-in types in the OpenCL language, so I exploited a simple set of functions that I found on internet that defines the *cdouble* custom type simply as a vector *double2* and defines properly the operations on complex numbers. The *double2* type is a vector data type included in the kernel language that is basically a couple of double values; the advantage of such type is that the operations involving it should be furtherly optimized for the parallelization, even though I used it with a different aim.

Now as an example I will briefly explain the kernel *mat_mul* which is quite instructive. Below it is reported the code.

```
1  __kernel void mat_mul(__global double *A,__global double *B,__global
       double *C)
2  {
3     int id=get_global_id(0);
4
5     int i,j,k;
6     double temp=0;
7     i=id/N;
8     j=id-i*N;
9     for(k=0;k<N;k++){
10       temp+=A[i*N+k]*B[k*N+j];
11    }
12    C[id]=temp;
13 }
```

As you can see from line 1 the kernel is defined pretty much as any standard C function. The important differences are that kernels are forced to return *void* and must include the specifier *__kernel* before the definition. This particular function has three pointers to double as arguments and they are preceded by the specifier *__global*, which indicates that they will be stored in the global memory of the device. The first two will contain the matrices to be multiplied and the third the result of the product. Then at line 3 each work item executing the kernel collects its own global id (which is a number between 0 and N^2) and stores it inside the variable *id*. At lines 7-8 it computes the matricial indices corresponding to its global id and then in the *for* cycle at lines 9-11 it computes the scalar product entering in the correct position of the resulting matrix. Finally this value is stored in the right position inside the memory addressed by the pointer *C*.

On the host side, in order to call a kernel one has to create *buffer* objects and copy inside them the data he wants to pass as kernel arguments. The copying operation has to be enqueued in the device command queue just like the call of the kernel. Then, after the kernel execution, data can be copied back from the buffer to some data structure of the host

by enqueuing another operation. See the code below for an example referred to the *mat_mul* kernel.

```
1   A_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,N*N*sizeof(double),NULL
        ,NULL);
2   B_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,N*N*sizeof(double),NULL
        ,NULL);
3   C_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,N*N*sizeof(double),NULL
        ,NULL);
```

```
1       //temp=A*Q
2       clEnqueueWriteBuffer(queue, A_buff, CL_TRUE, 0,N*N*sizeof(double), A
            , 0, NULL, NULL);
3       clEnqueueWriteBuffer(queue, B_buff, CL_TRUE, 0,N*N*sizeof(double), Q
            , 0, NULL, NULL);
4       clSetKernelArg(mat_mul, 0, sizeof(cl_mem), &A_buff);
5       clSetKernelArg(mat_mul, 1, sizeof(cl_mem), &B_buff);
6       clSetKernelArg(mat_mul, 2, sizeof(cl_mem), &C_buff);
7       clEnqueueNDRangeKernel(queue,mat_mul,1,NULL,&glob_size,NULL,0,NULL,
            NULL);
```

Once they proved to work as expected, I put together in a single program the single-thread and the OpenCL codes in order to test their performances. The full host code can be found in appendix A.1 and the kernels in appendix A.2[10]. The former is designed to create some basic statistics out of a sample of execution times that is collected by running the diagonalization algorithms a certain number of times for each given N. Both N and the dimension of the sample are taken as input from the file "inp.txt". To collect each element of the sample the code initializes a random hermitian matrix whose entries have a real part and an imaginary part uniformly distributed between -100 and 100 (function *init* - lines 146-178) and then it finds its lowest eigenvalue using four different methods: the single-thread code (function *stCPU* - lines 292-419), the OpenCL code running on the GPU (function *openclGPU* - lines 452-752), the OpenCL code running on the CPU (function *openclCPU* - lines 757-1056), and also the LAPACK routine *zheev* (function *lapack* - lines 425-446) for checking purposes. The execution times of each of these functions are collected using the function *clock* of the C time library but I keep in the sample only those corresponding to a correct eigenvalue determination by comparing them with the value found by LAPACK up to a precision of 10^{-7}. Then the program computes the mean and the standard deviation of the execution times sample and also the rate of correct results, and writes them in the file "out.txt" in a readable format (main of the code - lines 58-137).

I have to say that I did not deal much with error handling and pre/post conditions since the program was not intended to be rock solid for future applications but rather to test OpenCL performances, and its current behavior is enough for this purpose.

The code execution has been managed by a simple Python script that compiles the host code and launches it for each couple of input parameters (matrix dimension and cardinality of the statistical sample) that I have chosen.

2.3 Results

The program runned on my computer for a few days mapping the execution times and the rate of correct results for different values of N between 10 and 1000. As the input dimension raised I had to reduce the cardinality of the statistical sample otherwise the program would have taken too much time to finish the execution. For the same reason I also stopped the

[10] In the version reported here the eigenvector is not computed but in the end it is just a matter of multiplying two matrices.

		LAPACK		**Single-thread**			**OpenCL - GPU**			**OpenCL - CPU**		
N	#	$t\,[s]$	$\sigma_t\,[s]$	$t\,[s]$	$\sigma_t\,[s]$	η	$t\,[s]$	$\sigma_t\,[s]$	η	$t\,[s]$	$\sigma_t\,[s]$	η
10	100	0	0	0.002	0.005	99%	0.2	0.1	99%	0.08	0.01	99%
20	100	0.000	0.002	0.02	0.04	97%	0.3	0.3	97%	0.13	0.05	97%
30	100	0.000	0.003	0.2	0.7	93%	0.6	1.5	93%	0.2	0.4	93%
40	100	0.001	0.003	0.2	0.2	89%	0.4	0.3	89%	0.2	0.1	84%
50	100	0.002	0.005	0.4	0.3	67%	0.4	0.2	67%	0.2	0.1	76%
60	100	0.003	0.006	1.0	1.1	75%	0.6	0.6	80%	0.3	0.2	78%
70	100	0.004	0.007	1.8	1.2	82%	0.6	0.3	82%	0.4	0.3	87%
80	100	0.005	0.007	3.1	2.7	93%	0.7	0.5	93%	0.6	0.8	89%
90	100	0.006	0.008	7.2	21.4	93%	1.1	2.9	94%	0.9	0.8	91%
100	100	0.008	0.008	8.2	8.6	90%	1.0	0.8	90%	1.6	1.9	93%
120	50	0.013	0.007	13	9	90%	1.1	0.6	90%	2.3	1.9	90%
140	50	0.021	0.007	23	15	98%	1.5	0.7	94%	6.7	14.9	94%
160	50	0.030	0.005	48	31	98%	2.2	1.3	98%	7.2	5.1	96%
180	50	0.040	0.008	69	45	98%	2.5	1.4	98%	13	12	98%
200	50	0.053	0.008	91	67	98%	2.8	1.8	98%	17	14	98%
220	50	0.073	0.007	174	119	100%	3.5	2.2	100%	23	22	98%
240	50	0.092	0.008	252	483	100%	5.2	3.1	100%	30	21	100%
260	50	0.117	0.008	282	208	100%	6.9	4.6	100%	50	32	100%
280	50	0.143	0.007	347	243	100%	8.4	5.7	100%	57	48	100%
300	50	0.170	0.006	567	414	100%	11	7	100%	75	49	100%
320	50	0.206	0.008	-	-	-	14	14	100%	106	68	100%
340	50	0.245	0.009	-	-	-	14	10	100%	113	76	96%
360	50	0.290	0.009	-	-	-	18	11	98%	176	177	100%
380	50	0.332	0.008	-	-	-	25	23	100%	160	114	98%
400	50	0.39	0.01	-	-	-	32	22	100%	239	205	100%
450	20	0.55	0.01	-	-	-	41	30	100%	430	290	100%
500	20	0.75	0.02	-	-	-	60	42	100%	615	408	100%
550	20	0.99	0.02	-	-	-	66	59	100%	703	751	100%
600	20	1.27	0.02	-	-	-	171	114	95%	1034	856	100%
700	10	2.01	0.04	-	-	-	249	176	100%	1411	1245	100%
800	10	2.98	0.03	-	-	-	303	307	100%	2840	2333	100%
900	10	4.1	0.3	-	-	-	792	645	100%	10166	6108	100%
1000	10	5.6	0.3	-	-	-	774	656	100%	5431	5378	100%

Table 2: *Results of the diagonalization algorithm performance test.*

single-thread code at $N = 300$. The results of this analysis are reported in table 2 (# is the dimension of the statistical sample and η is the rate i.e. the number of correct eigenvalues obtained divided by #). The plot of the execution times vs input dimension is report is reported in figure 3 (the y-axis is in logarithmic scale).

There are some quite interesting observation I can point out by looking at this data. Firstly we can see how LAPACK is always much faster than my algorithm, but this is a behavior I was expecting as I have said in the previous section. Yet we can clearly see a remarkable boost in the performance both when passing from a single-thread to a parallelized algorithm and when passing from the CPU to the GPU implementation of OpenCL. This is undoubtedly a success! Apart from the lowest dimensions (approximately up to $N = 100$) the GPU is about 25-50x faster than the single thread code and about 5-10x faster than the CPU running OpenCL. For what concerns the performances at dimensions lower than $N = 100$ we can see that the situation is less clear, with the single-thread algorithm that is the fastest up to $N = 40$ while OpenCL running on the CPU is the fastest from $N = 50$ to $N = 90$. I think that this is mainly due to the fact that at low dimensions the data transfer from the host memory to the devices memory becomes a bottleneck for the program. Indeed this is not a particularly fast operation and thus when the actual computational time becomes very

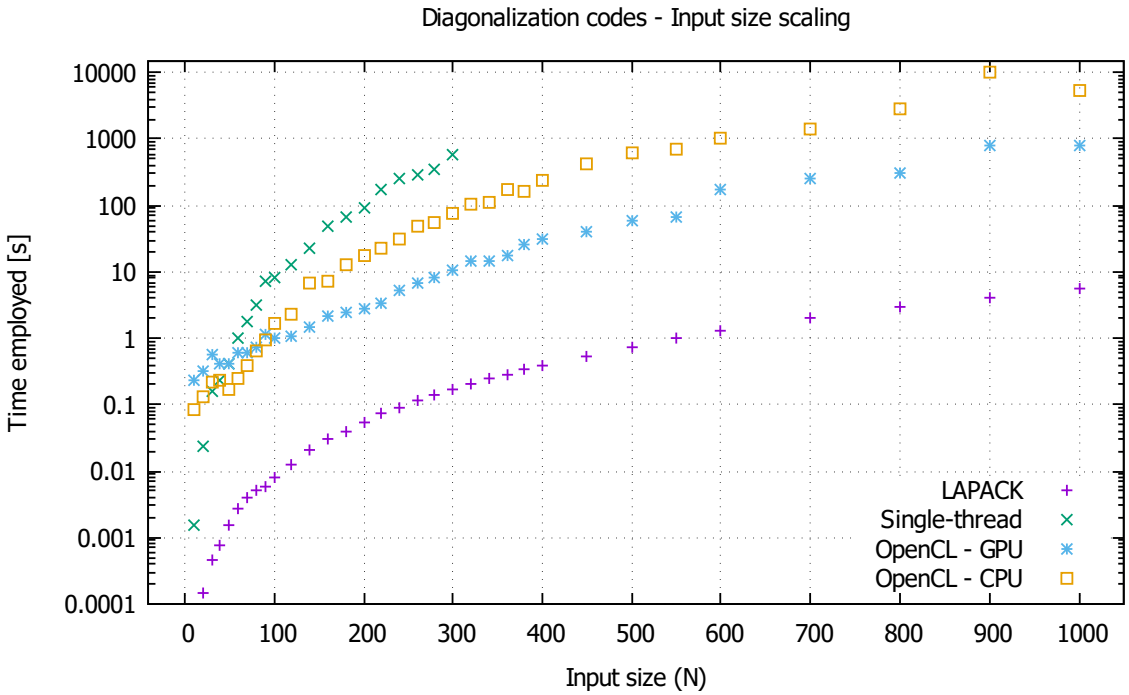


Figure 3: *Scaling of the four codes used for solving the eigenproblem with respect to the size of the input matrix.*

short this contribution to the total time becomes significant.

Next if one looks at the standard deviations he certainly will notice that in most cases it is very big, almost always of the same order of magnitude of the mean itself. This means that the collected times are spread in a wide range. A possible explanation of this great variability could be the algorithm itself, in particular the fact that the convergence speed depends highly on the initial matrix and thus the number of iterations required by the QR algorithm to get to the result is rather casual.

The last interesting comment I can do is about the rate of success. The instability of the QR algorithm lead to a certain number of wrong eigenvalues, but we can see how this issue becomes less and less important as the input dimension N increases. For $N > 200$ we get almost 100% of the times the correct eigenvalue, which is not bad in this context. Instead we cannot trust the program much for lower N and this is a further reason to prefer the parallelized approach.

Finally I performed a more quantitative analysis of the scaling of these algorithms. I linearised the data in the former plot and I fitted them with a line. The results are reported in figure 4. From the plot one can see even more clearly the anomalous behavior of the execution times for low values of N when using OpenCL. For this reason I excluded these data from the linear fit (the fitted points are those darker in the plot). The obtained scaling are:

$$\text{LAPACK} \sim \mathcal{O}(N^{2.69})$$

$$\text{Single-thread} \sim \mathcal{O}(N^{3.70})$$

$$\text{OpenCL on GPU} \sim \mathcal{O}(N^{3.62})$$

$$\text{OpenCL on CPU} \sim \mathcal{O}(N^{3.69})$$

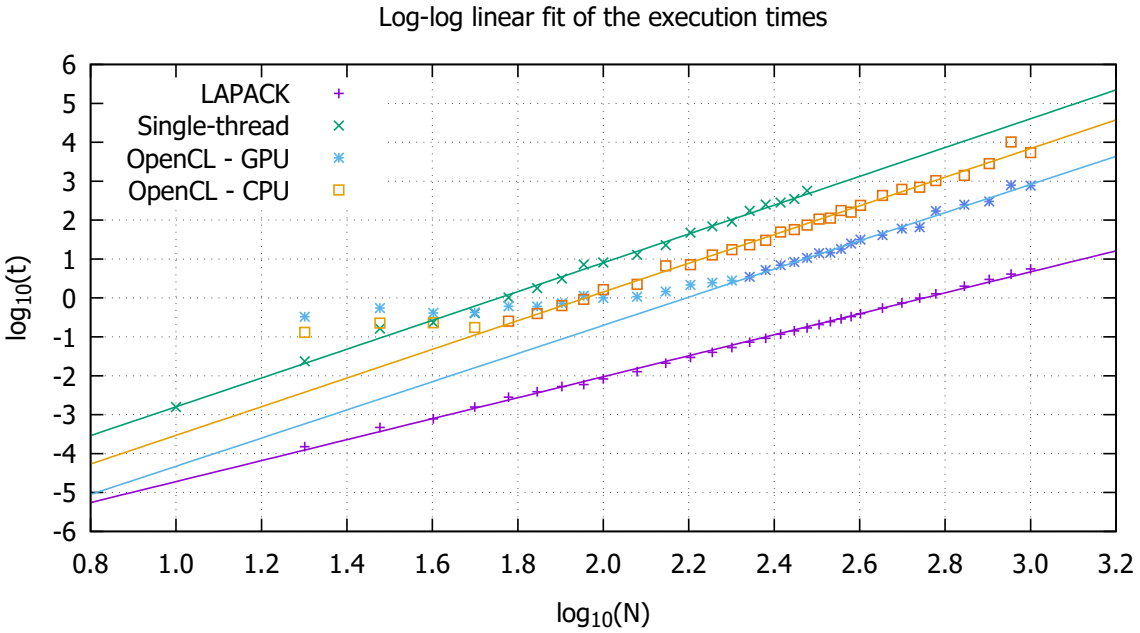


Figure 4: *Log-log plot of the executions times vs input sizes and linear fit.*

Despite the expectations there is not a great difference in the scaling of the three codes I realised (while LAPACK is much faster). This can be seen also on the plot: the three fitting lines corresponding to my algorithms are pretty much parallel unlike that corresponding to LAPACK. What is really different is the y-intercept which is the parameter related to the coefficient in front of the power of N in the scaling. So apparently the parallelization is not able to improve significantly the scaling of an algorithm but only to speedup its execution affecting only the multiplicative coefficient. Nevertheless more analysis and data would be required to state anything conclusive, also because the parallelization algorithm has still optimization possibilities.

3 2D Schrödinger equation

After having learned the basic working principles of OpenCL, I exploited its potentialities to solve the 2D time-independent Schrödinger equation for different potentials (in particular I found the ground state and first excited state).

3.1 Recall of FDM

I exploited the FDM[11] as already done in exercise 6, but this time I used its 2D version. They are quite similar, the only difference is that now the matrix to be diagonalized is no more tridiagonal as in 1D. For simplicity I consider a normalized Schrödinger equation (i.e. $m = 1/2$, $\hbar = 1$):

$$\left[-\nabla^2 + V(x,y)\right]\psi(x,y) = E\,\psi(x,y)$$

After an homogeneous discretization of the domain into intervals of width Δ (the same for both the x and the y coordinates), exploiting the usual approximation for the second

[11] Finite Difference Method.

derivative one gets the expression:

$$-\frac{1}{\Delta^2}\left(\psi_{n+1,m}+\psi_{n-1,m}+\psi_{n,m+1}+\psi_{n,m-1}\right)+\left(V_{n,m}+\frac{4}{\Delta^2}\right)\psi_{n,m}=E\,\psi_{n,m}$$

where $\psi_{n,m}\equiv\psi(x_n,y_m)$, with $(x_n,\,y_m)$ being a point of the lattice, and analogously for $V_{n,m}$. This equation is not in the form we desire yet (i.e. that of an eigenvalue problem) since we have too many indices. Being N the number of lattice points in the x coordinate we apply the following redefinitions:

$$\psi_{n,m}\longrightarrow\psi_{n+mN}\qquad\qquad V_{n,m}\longrightarrow V_{n+mN}$$

Then the above equation can be rewritten as:

$$-\frac{1}{\Delta^2}\left(\psi_{n+1+mN}+\psi_{n-1+mN}+\psi_{n+(m+1)N}+\psi_{n+(m-1)N}\right)+ \\ +\left(V_{n+mN}+\frac{4}{\Delta^2}\right)\psi_{n+mN}=E\,\psi_{n+mN}$$

So at the price of having increased the dimensionality of the problem, we reached the desired expression. The eigenvalues of this operator are the energies of the eigenstates and the eigenvectors contain the values of the corresponding wavefunction evaluated in the points of the lattice. In the following we will assume that $n=1,\ldots,N$ and $m=1,\ldots,N$ that is the domain is symmetric in the two coordinates.

The interesting thing is that it is much more difficult to solve analytically a 2D problem than a 1D problem, whereas the numerical solution is essentially the same, only with bigger matrices. Therefore with enough time/computational resources we will always be able to find the solution. Actually this is true for a generic number of dimensions: the only limits are the physical means.

For what follows it is also important to recall the following fact valid in any dimension: though the wavefunction is complex in general one can always choose the eigenfunctions of a given system to be real[12]. Indeed if $\psi(x)$ is a solution of the time-independent Schrödinger equation, then also $\psi^*(x)$ is a solution of the same equation; now since the Schrödinger equation is linear, the function $\bar{\psi}(x)\equiv\psi(x)+\psi^*(x)$ is still a solution and it is real by construction.

In particular the potentials I will consider in the following are:

(A) Decoupled harmonic oscillator: $V(x,y)=\frac{1}{2}\left(x^2+y^2\right)$

(B) Regularized electrostatic potential[13]: $V(x,y)=\begin{cases}-\frac{3}{\sqrt{x^2+y^2}} & \text{for } x^2-y^2>\frac{9}{100}\\ -10 & \text{for } x^2-y^2\leq\frac{9}{100}\end{cases}$

(C) Central mexican hat: $V(x,y)=-\left(x^2+y^2\right)+\frac{1}{8}\left(x^2+y^2\right)^2$

(D) Double mexican hat: $V(x,y)=-x^2+\frac{1}{8}x^4-y^2+\frac{1}{8}y^4$

[12]This is true if the potential $V(x)$ is real which is the case for any physical interesting system.

[13]I had to regularize the Coulomb potential since otherwise its divergence at $x=y=0$ would have broken the numerical solution.

Clearly the potential (A) has an analytical solution, and indeed I used it for checking the correctness of the numerical results. The energy eigenvalues are given by:

$$E_{\vec{n}} = \hbar\,\omega\,(n_1 + n_2 + 1)$$

where $n_1, n_2 \in \mathbb{N}$. With the above defined normalization $\hbar\,\omega = \sqrt{2}$ so that the ground state and the first excited level have energies:

$$E_0 = \sqrt{2} \approx 1.141214 \qquad E_1 = 2\,\sqrt{2} \approx 2.828427 \tag{1}$$

Also note that potentials (B) and (C) could be simplified to a 1D problem exploiting the central symmetry, yet the aim of this program was to solve general 2D problems therefore I kept them 2D.

3.2 Code development

By looking empirically at the potentials, I set the domain of the algorithm to be $[-4, 4] \times [-4, 4]$ and I chose the lattice to have 401 points for each axis, which results in a total of 160801 points separated by a distance of $\Delta = 0.02$ in each coordinate.

Considering that for this application I had to deal with matrices much bigger than 1000×1000 (they have about $160801 \times 160801 \approx 2.6 \cdot 10^{10}$ elements) and that are sparse instead of dense, I could not use the program described in section 2.2. I searched on internet for some OpenCL library including functions for computing the lowest eigenvalues of a sparse matrix. The only one that I was able to find is the C++ library *ViennaCL*. In particular it contains also an example code that uses Lanczos algorithm and then the bisection method to find the greater k eigenpairs (an approximation of them actually), where k is specified as an input parameter. So once computed the matrix to diagonalize I simply passed its opposite to the ViennaCL function, since I was interested in the lowest eigepairs. The input format required for the matrix to be diagonalized is the Matrix Market exchange format (extension *.mtx*) which is essentially a text file that contains at each row the coordinates and the value of an element of the matrix. An element is assumed to be 0 if not specified in the file. It is clear that this format is very convenient when it is necessary to store a sparse matrix since one can avoid to write a bunch of useless zeros. Another important parameter of this function is the size of the Krylov subspace used in the computations. The behavior of the output when varying this parameter was rather strange, thus I had to use a bit of fine-tuning to find the optimal value, comparing every time the result for the potential (A) with its analytical solution. Eventually I set it to 400 for the computation of the ground state and to 900 for that of the first excited state.

Given that, the number of points used to partition the domain was the highest that allowed my computer to find the first two eigenstates of the quantum systems. In fact, despite the code was still very fast (less than a minute for computing the second lowest eigenpair), a higher number of points resulted in crashes of the code due to VRAM saturation. This limitation is also what prevented me from solving also some 3D problem.

The resulting eigenfunctions are written in a suitable form into the file "psi.txt". The full code I used is reported in appendix B.

3.3 Results

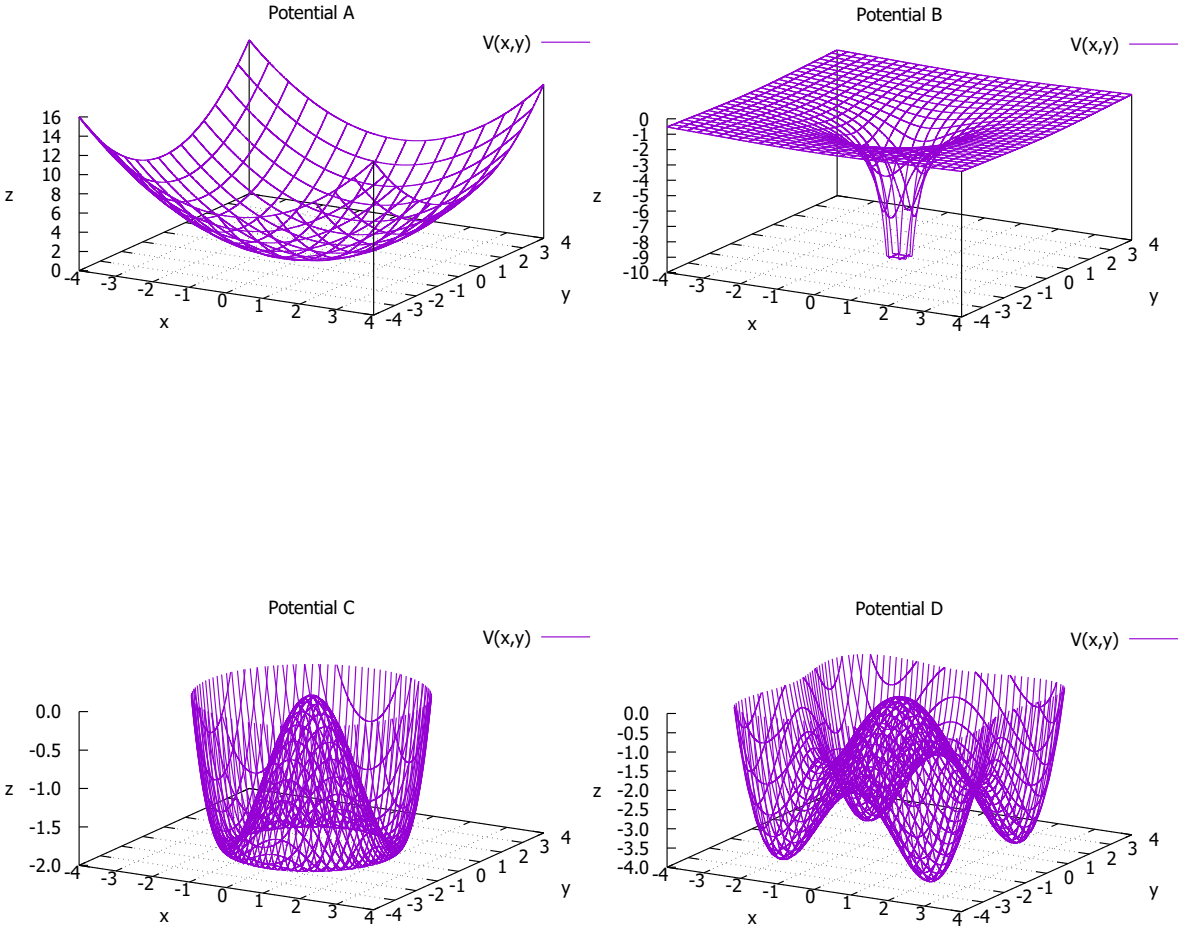
The resulting eigenvalues of the four above reported potentials can be found in table 3. Also the figures below contain the plots of the four potentials and those of the modulus squared of the eigenfunctions found[14]. A positive aspect is that they are quite well defined, thus it

[14] I have plotted only a subset of the total 160801 points since reducing the size of the images was a typographic need.

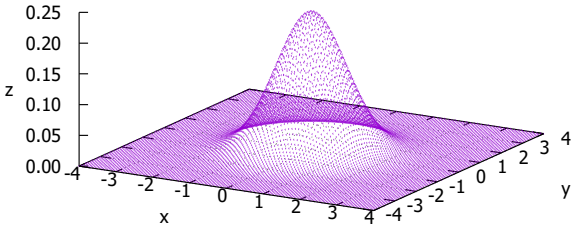
Potential	E_0	E_1
A	1.413258	2.829293
B	-5.178028	-0.944380
C	-0.837434	-0.280717
D	-1.710583	-1.477471

Table 3: *First two energy eigenvalues for the potentials above reported.*

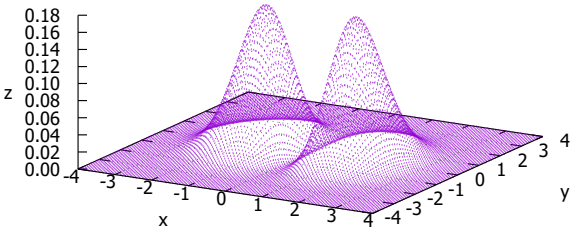
seems that the code has at least a good precision. By considering the numerical estimates of the energy eigenvalues for the potential (A) and comparing them with (1) we can state that the accuracy is also quite good, even though not excellent: both the energies differs from the exact ones by about 10^{-4}. However another consistency check comes from the fact that all the eigenvalues have the correct sign (plus for the harmonic oscillator since there $V(x, y) \geq 0$ always, minus for the other three since it is not true anymore). Moreover all the ground state found have a shape that matches with that one could have expected by looking at the potentials. So I conclude that the code works quite well.



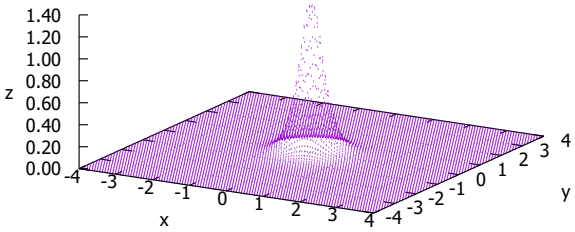
Ground level A



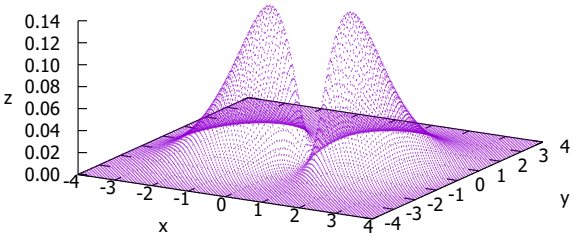
First level A



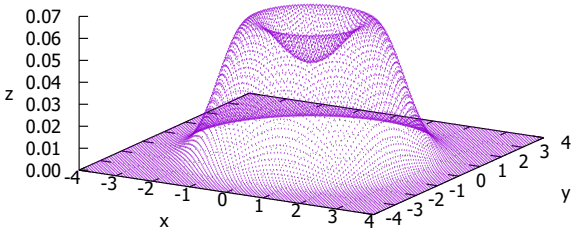
Ground level B



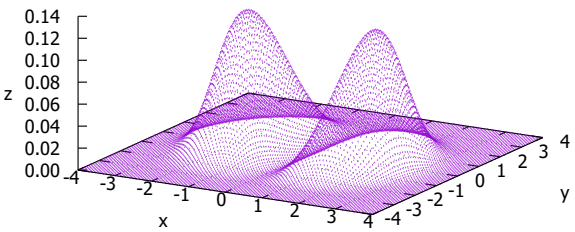
First level B



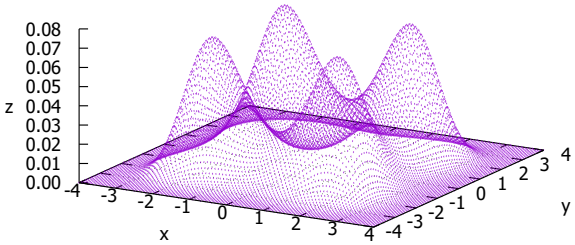
Ground level C



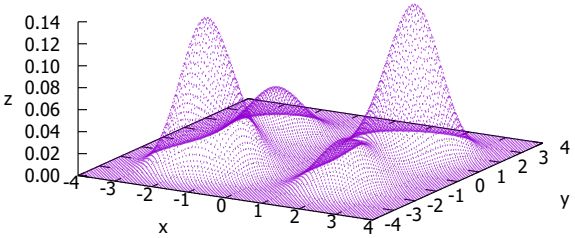
First level C



Ground level D



First level D



4 Conclusion and self-evalutation

The main result of the present work is that I have learned the ideas behind the parallelization techniques in computational science and the basic working principles of OpenCL, which is a very powerful tool that provides a way to implement such techniques in almost any mainstream computer. Now I am able to code basic applications implementing the OpenCL framework and I have gained the capabilities of studying more advanced topics.

OpenCL itself has proven to yield a consistent gain in the performances of an algorithm even if implemented without spending a great effort in the optimization. Therefore it can certainly constitute a powerful ally to save time for heavy computations.

The two application that I realised were quite successful in their intents even though they did not reach perfect results. In particular the diagonalization algorithm hid much more difficulties than I thought and I had to devote the majority of the time employed in fixing issues that occurred while writing it. Especially the numerical instability bothered me for a long time and eventually I had to renounce to some further passages that I would have liked to implement in a parallelized way. On the other hand the application to the solution of the 2D time-independent Schrödinger equation was pretty straightforward thanks to the functions provided by the library *ViennaCL*. Here I learned how to generalize the FDM to higher dimensions and also how to write huge sparse matrices in a suitable form, that is with the Matrix Market exchange format. Also in this application OpenCL proved to have remarkable performances, finding the greatest eigenpairs of a matrix with some billions elements in less than a minute.

Overall I am quite satisfied with the work done and I am glad to have opened some doors for future studying possibilities.

A Matrix diagonalization code

A.1 Host code

```
1  ////////////////////////////////////////////////////////////////////////
2  //
3  // Evaluates the capabilities of the hardware by computing some statistics
       about
4  // the computation of the lowest eigenvalues of a matrix in four different
       ways:
5  // 1) LAPACK library
6  // 2) Lanczos + QR algorithm - pure single-thread C
7  // 3) Lanczos + QR algorithm - OpenCL over GPU
8  // 4) Lanczos + QR algorithm - OpenCL over CPU
9  //
10 ////////////////////////////////////////////////////////////////////////
11
12
13 // libraries
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <math.h>
18 #include <complex.h>
19 #include <time.h>
20 #include <string.h>
21 #include <lapacke.h>
22
23 #ifdef __APPLE__
24 #include <OpenCL/opencl.h>
25 #else
26 #include <CL/cl.h>
27 #endif
28
29 #define MAX_SOURCE_SIZE (0x100000)
30
31
32 int N;   //dimension of the hermitian matrix
33 int it;  //number of iterations
34
35
36
37
38 //PROTOTYPES
39
40 void init(double complex *cmat,double *mat);
41 void prtcmpx(double complex z);
42 double complex dot(double complex *u, double complex *v);
43 double norm(double complex *u);
44 double complex* matvec(double complex *M, double complex *v);
45 double* QR(double *diag,double *subdiag);
46 double* matmul(double *A, double *B);
47 double* transp(double *A);
48
49 void stCPU(double *time ,double *eig0 ,double complex *M);
50 void openclGPU(double *time ,double *eig0 ,double *M);
51 void openclCPU(double *time ,double *eig0 ,double *M);
52 void lapack(double *time ,double *eig0 ,double complex *M);
53
54
55
```

```
56
57
58  //MAIN
59
60  int main(){
61
62     FILE *f;
63     f=fopen("inp.txt","r");
64     fscanf(f,"%d",&N);
65     fscanf(f,"%d",&it);
66     fclose(f);
67
68     int i,j,nhit[3];
69     double time[4],eig[4],mean[4],temp[4][it],dev[4];
70     double *M;
71     double complex *cM;
72
73     printf("DIMENSION: %d\n",N);
74     printf("NUMBER OF ITERATIONS:  %d\n\n",it);
75
76     mean[0]=0; dev[0]=0; nhit[0]=0;
77     mean[1]=0; dev[1]=0; nhit[1]=0;
78     mean[2]=0; dev[2]=0; nhit[2]=0;
79     mean[3]=0; dev[3]=0;
80
81     for(i=0;i<it;i++){
82
83        printf("\nIteration %d\n\n",i+1);
84
85        M=(double*)malloc(sizeof(double)*2*N*N);
86        cM=malloc(sizeof(double complex)*N*N);
87        init(cM,M);
88
89        if(N<=300)
90           stCPU(&time[0],&eig[0],cM);
91        lapack(&time[3],&eig[3],cM);
92        openclGPU(&time[1],&eig[1],M);
93        openclCPU(&time[2],&eig[2],M);
94
95        free(M);
96        free(cM);
97
98        mean[3]+=time[3];
99        temp[3][i]=time[3];
100       for(j=0;j<3;j++){
101          if(fabs(eig[j]-eig[3])<1e-7){
102             mean[j]+=time[j];
103             temp[j][nhit[j]]=time[j];
104             nhit[j]++;
105          }
106       }
107    }
108
109    mean[0]=(double)mean[0]/nhit[0];
110    mean[1]=(double)mean[1]/nhit[1];
111    mean[2]=(double)mean[2]/nhit[2];
112    mean[3]=(double)mean[3]/it;
113
114    for(i=0;i<nhit[0];i++){
115       dev[0]+=(temp[0][i]-mean[0])*(temp[0][i]-mean[0]);
116    }
```

```
117     for(i=0;i<nhit[1];i++){
118        dev[1]+=(temp[1][i]-mean[1])*(temp[1][i]-mean[1]);
119     }
120     for(i=0;i<nhit[2];i++){
121        dev[2]+=(temp[2][i]-mean[2])*(temp[2][i]-mean[2]);
122     }
123     for(i=0;i<it;i++){
124        dev[3]+=(temp[3][i]-mean[3])*(temp[3][i]-mean[3]);
125     }
126     dev[0]=sqrt((double)dev[0]/(nhit[0]-1));
127     dev[1]=sqrt((double)dev[1]/(nhit[1]-1));
128     dev[2]=sqrt((double)dev[2]/(nhit[2]-1));
129     dev[3]=sqrt((double)dev[3]/(it-1));
130
131     f=fopen("out.txt","a");
132     fprintf(f,"%d  %d %f %f %f %f %f %f %f %f %f %f %f\n",N,it,mean[3],dev
            [3],mean[0],dev[0],
133        (double)nhit[0]/it,mean[1],dev[1],(double)nhit[1]/it,mean[2],dev
               [2],(double)nhit[2]/it);
134     fclose(f);
135
136     return 0;
137 }
138
139
140
141
142
143 //FUNCTIONS
144
145
146 //random hermitian matrix generation
147
148 void init(double complex *cmat,double *mat){
149     double x,y;
150     int i,j;
151     srand((unsigned int)time(NULL));
152     for(i=0;i<N;i++){
153        for(j=i;j<N;j++){
154           if(i==j){
155              x = ((double)rand()/(double)(RAND_MAX)) * 200-100;
156              y=0;
157              cmat[i*N+j]=x+y*I;
158              mat[i*2*N+2*j]=x;
159              mat[i*2*N+2*j+1]=y;
160           }
161           else{
162              x = ((double)rand()/(double)(RAND_MAX)) * 200-100;
163              y = ((double)rand()/(double)(RAND_MAX)) * 200-100;
164              cmat[i*N+j]=x+y*I;
165              cmat[j*N+i]=x-y*I;
166              mat[i*2*N+2*j]=x;
167              mat[i*2*N+2*j+1]=y;
168              mat[j*2*N+2*i]=x;
169              mat[j*2*N+2*i+1]=-y;
170           }
171        }
172     }
173     return;
174 }
175
```

```
176
177 //print complex number
178
179 void prtcmpx(double complex z){
180    printf("(%f,%f)\n",creal(z),cimag(z));
181    return;
182 }
183
184
185 //dot product
186
187 double complex dot(double complex *u, double complex *v){
188    int i;
189    double complex z=0+0*I;
190    for(i=0;i<N;i++)
191       z+=conj(u[i])*v[i];
192    return z;
193 }
194
195
196 //euclidean norm
197
198 double norm(double complex *u){
199    return sqrt(cabs(dot(u,u)));
200 }
201
202
203 //matrix-vector multiplication
204
205 double complex* matvec(double complex *M, double complex *v){
206    int i,j;
207    double complex *z;
208    z=malloc(sizeof(double complex)*N);
209    for(i=0;i<N;i++){
210       z[i]=0+0*I;
211       for(j=0;j<N;j++)
212          z[i]+=M[i*N+j]*v[j];
213    }
214    return z;
215 }
216
217
218 //QR decomposition
219
220 double* QR(double *diag,double *subdiag){
221
222    int i,j;
223    double norm,gam,sig,temp1,temp2,temp3;
224    double *Q;
225
226    Q=(double*)calloc(N*N,sizeof(double));
227    norm=sqrt(diag[0]*diag[0]+subdiag[0]*subdiag[0]);
228    gam=diag[0]/norm;
229    sig=subdiag[0]/norm;
230    Q[0]=gam;
231    Q[1]=-sig;
232    Q[N]=sig;
233    Q[N+1]=gam;
234    temp1=-sig*subdiag[0]+gam*diag[1];
235    temp2=gam*subdiag[1];
236
```

```
237     for(i=1;i<N-1;i++){
238        norm=sqrt(temp1*temp1+subdiag[i]*subdiag[i]);
239        gam=temp1/norm;
240        sig=subdiag[i]/norm;
241        for(j=0;j<i+1;j++){
242           temp3=Q[j*N+i];
243           Q[j*N+i]=temp3*gam;
244           Q[j*N+i+1]=-temp3*sig;
245        }
246        Q[(i+1)*N+i]=sig;
247        Q[(i+1)*N+i+1]=gam;
248        temp1=-sig*temp2+gam*diag[i+1];
249        temp2=gam*subdiag[i+1];
250     }
251     return Q;
252  }
253
254
255  //matrix multiplication
256
257  double* matmul(double *A, double *B){
258     int i,j,k;
259     double temp;
260     double *M;
261     M=malloc(sizeof(double)*N*N);
262     for(i=0;i<N;i++){
263        for(j=0;j<N;j++){
264           temp=0;
265           for(k=0;k<N;k++){
266              temp+=A[i*N+k]*B[k*N+j];
267           }
268           M[i*N+j]=temp;
269        }
270     }
271     return M;
272  }
273
274
275  //matrix transpose
276
277  double* transp(double *A){
278     int i,j;
279     double *M;
280     M=malloc(sizeof(double)*N*N);
281     for(i=0;i<N;i++)
282        for(j=0;j<N;j++)
283           M[j*N+i]=A[i*N+j];
284     return M;
285  }
286
287
288
289
290
291
292  //SINGLE-THREAD CPU
293
294
295  void stCPU(double *time ,double *eig0 ,double complex *M){
296
297     printf("Single-thread CPU algorithm started!\n");
```

```
298     clock_t start, end;
299     start = clock();
300
301
302     //lanczos algorithm
303
304     int i,j;
305     double a,b;
306     double complex *P,*w,*v;
307     double *diag,*subdiag;
308
309     v=malloc(sizeof(double complex)*N);
310     P=malloc(sizeof(double complex)*N*N);
311     diag=malloc(sizeof(double)*N);
312     subdiag=malloc(sizeof(double)*(N-1));
313
314     P[0]=1+0*I;
315     v[0]=P[0];
316     for(i=1;i<N;i++){
317        P[i*N]=0+0*I;
318        v[i]=P[i*N];
319     }
320     w=matvec(M,v);
321     a=creal(dot(w,v));
322     diag[0]=a;
323     for(i=0;i<N;i++)
324        w[i]-=(a+0*I)*v[i];
325
326     for(i=1;i<N;i++){
327        b=norm(w);
328        subdiag[i-1]=b;
329        for(j=0;j<N;j++){
330           v[j]=w[j]/(b+0*I);
331           P[j*N+i]=v[j];
332        }
333        free(w);
334        w=matvec(M,v);
335        a=dot(w,v);
336        diag[i]=a;
337        for(j=0;j<N;j++)
338           w[j]=w[j]-(a+0*I)*v[j]-(b+0*I)*P[j*N+i-1];
339     }
340     free(w);
341     free(v);
342     free(P);
343
344
345     //QR algorithm
346
347     int chk=0;
348     double mu,conf,conf2=0;
349     double *Q,*A;
350     mu=diag[N-1];
351     A=malloc(sizeof(double)*N*N);
352     for(i=0;i<N;i++){
353        for(j=0;j<N;j++){
354           if(i==j){
355              A[i*N+j]=diag[i];
356           }
357           else if(i==j+1){
358              A[i*N+j]=subdiag[i-1];
```

```
359             }
360             else if(i==j-1){
361                 A[i*N+j]=subdiag[i];
362             }
363             else{
364                 A[i*N+j]=0;
365             }
366         }
367         diag[i]-=mu;
368     }
369 
370     Q=QR(diag,subdiag);
371     while(chk==0){
372         double *temp,*Qt,*tdiag,*tsubdiag,*Qnew,*Ak;
373         temp=matmul(A,Q);
374         Qt=transp(Q);
375         Ak=matmul(Qt,temp);
376         free(temp);
377         free(Qt);
378         mu=Ak[N*N-1];
379         tdiag=malloc(sizeof(double)*N);
380         tsubdiag=malloc(sizeof(double)*(N-1));
381         for(i=0;i<N-1;i++){
382             tdiag[i]=Ak[i*N+i]-mu;
383             tsubdiag[i]=Ak[i*N+i+1];
384         }
385         free(Ak);
386         tdiag[N-1]=0;
387         Qnew=QR(tdiag,tsubdiag);
388         temp=matmul(Q,Qnew);
389         free(Q);
390         free(Qnew);
391         Q=temp;
392 
393         conf=tdiag[0];
394         for(i=1;i<N;i++){
395             if(tdiag[i]<conf)
396                 conf=tdiag[i];
397         }
398         conf+=mu;
399         if(fabs(conf-conf2)<1e-12)
400             chk=1;
401         conf2=conf;
402         //printf("%f\n",conf2);
403         free(tdiag);
404         free(tsubdiag);
405     }
406 
407     free(diag);
408     free(subdiag);
409     free(A);
410     free(Q);
411 
412     *eig0=conf2;
413 
414     end = clock();
415     *time=((double) (end - start)) / CLOCKS_PER_SEC;
416     printf("Single-thread CPU algorithm finished!\n");
417     printf("Execution time: %f\n",*time);
418     printf("Lowest eigenvalue: %f\n\n",*eig0);
419 }
```

```
420
421
422
423
424
425 //LAPACK
426
427 void lapack(double *time ,double *eig0 ,double complex *M){
428
429    printf("Lapack started!\n");
430    clock_t start, end;
431    start = clock();
432
433    lapack_int n;
434    n=N;
435    double *eig;
436    eig=malloc(sizeof(double)*N);
437    LAPACKE_zheev(LAPACK_ROW_MAJOR,'V','U',n,M,n,eig);
438    *eig0=eig[0];
439    free(eig);
440
441    end = clock();
442    *time=((double) (end - start)) / CLOCKS_PER_SEC;
443    printf("Lapack finished!\n");
444    printf("Execution time: %f\n",*time);
445    printf("Lowest eigenvalue: %f\n\n",*eig0);
446 }
447
448
449
450
451
452 //OpenCL GPU
453
454 void openclGPU(double *time ,double *eig0 ,double *M){
455
456    printf("OpenCL (GPU) started!\n");
457    clock_t start, end;
458    start = clock();
459
460    //opencl initialization
461
462    FILE *f;
463    int i,j;
464    char str[40];
465    char name[40];
466    char *source_str;
467     size_t source_size;
468    sprintf(str,"-cl-std=CL1.2 -D N=%d",N);
469
470    int dev=1;
471    cl_platform_id *platforms;
472    cl_uint num_platforms;
473     cl_device_id device;
474     cl_uint num_devices;
475    cl_context context;
476    cl_command_queue queue;
477    cl_program program;
478    size_t glob_size;
479    size_t glob_size2;
480    size_t loc_size=N;
```

```
481
482     clGetPlatformIDs(5,NULL,&num_platforms);
483     platforms=(cl_platform_id*) malloc(sizeof(cl_platform_id)*num_platforms
            );
484     clGetPlatformIDs(num_platforms,platforms,NULL);
485     //printf("Number of Platforms detected: %d\n",num_platforms);
486     for(i=0;i<num_platforms;i++){
487         clGetPlatformInfo(platforms[i],CL_PLATFORM_NAME,sizeof(name),&name,
                NULL);
488         //printf("%s\n",name);
489     }
490
491
492     if(dev==1){
493         clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,1,&device,&
                num_devices);
494         for(i=1;loc_size>256;i++){
495             if(loc_size%i==0)
496                 loc_size=N/i;
497         }
498         //printf("\nAMD GPU selected\n");
499     }
500     else if(dev==2){
501         clGetDeviceIDs(platforms[1],CL_DEVICE_TYPE_CPU,1,&device,&
                num_devices);
502         for(i=1;loc_size>8192;i++){
503             if(loc_size%i==0)
504                 loc_size=N/i;
505         }
506         //printf("\nIntel CPU selected\n");
507     }
508
509     context=clCreateContext( NULL, 1,&device, NULL, NULL, NULL);
510     queue=clCreateCommandQueueWithProperties(context,device,NULL,NULL);
511
512      f=fopen("kernels.cl", "r");
513      source_str = (char*)malloc(MAX_SOURCE_SIZE);
514      source_size = fread( source_str, 1, MAX_SOURCE_SIZE, f);
515      fclose(f);
516     program=clCreateProgramWithSource(context,1,(const char **)&source_str
            ,(const size_t *)&source_size,NULL);
517     clBuildProgram(program,1,&device,str,NULL,NULL);
518     free(source_str);
519
520     cl_kernel lanczos1 = clCreateKernel(program, "lanczos1", NULL);
521     cl_kernel lanczos2 = clCreateKernel(program, "lanczos2", NULL);
522     cl_kernel transpose = clCreateKernel(program, "transpose", NULL);
523     cl_kernel mat_mul = clCreateKernel(program, "mat_mul", NULL);
524     cl_kernel transfer = clCreateKernel(program, "transfer", NULL);
525
526
527
528
529     //lanczos algorithm
530
531     double a,b=1;
532     double *P, *v, *vold,*w;
533     double *diag,*subdiag,*ab;
534
535     cl_double alpha;
536     cl_double beta;
```

```
537     cl_mem M_buff;
538     cl_mem v_buff;
539     cl_mem vold_buff;
540     cl_mem w_buff;
541     cl_mem ab_buff;
542 
543     diag=malloc(sizeof(double)*N);
544     subdiag=malloc(sizeof(double)*N);
545     w=calloc(2*N,sizeof(double));
546     v=malloc(sizeof(double)*2*N);
547     vold=calloc(2*N,sizeof(double));
548     P=calloc(2*N*N,sizeof(double));
549     ab=malloc(N*sizeof(double));
550     P[0]=1;
551     P[1]=0;
552     w[0]=P[0];
553     w[1]=P[1];
554 
555     glob_size=N;
556     M_buff=clCreateBuffer(context,CL_MEM_READ_ONLY,2*N*N*sizeof(double),
          NULL,NULL);
557     v_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,2*N*sizeof(double),NULL
          ,NULL);
558     vold_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,2*N*sizeof(double),
          NULL,NULL);
559     w_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,2*N*sizeof(double),NULL
          ,NULL);
560     ab_buff=clCreateBuffer(context,CL_MEM_WRITE_ONLY,N*sizeof(double),NULL,
          NULL);
561 
562     for(i=0;i<N;i++){
563 
564        beta=b;
565 
566        clEnqueueWriteBuffer(queue, M_buff, CL_TRUE, 0,2*N*N*sizeof(double),
              M, 0, NULL, NULL);
567        clEnqueueWriteBuffer(queue, w_buff, CL_TRUE, 0,2*N*sizeof(double), w
              , 0, NULL, NULL);
568        clSetKernelArg(lanczos1, 0, sizeof(cl_double), &beta);
569        clSetKernelArg(lanczos1, 1, sizeof(cl_mem), &M_buff);
570        clSetKernelArg(lanczos1, 2, sizeof(cl_mem), &v_buff);
571        clSetKernelArg(lanczos1, 3, sizeof(cl_mem), &w_buff);
572        clSetKernelArg(lanczos1, 4, sizeof(cl_mem), &ab_buff);
573        clEnqueueNDRangeKernel(queue,lanczos1,1,NULL,&glob_size,&loc_size,0,
              NULL,NULL);
574        clEnqueueReadBuffer(queue,v_buff, CL_TRUE, 0,2*N*sizeof(double), v,
              0, NULL, NULL);
575        clEnqueueReadBuffer(queue,ab_buff, CL_TRUE, 0,N*sizeof(double), ab,
              0, NULL, NULL);
576 
577        a=ab[0];
578        for(j=1;j<N;j++){
579           a+=ab[j];
580        }
581        diag[i]=a;
582 
583        if(i==0)
584           beta=0;
585 
586        alpha=a;
587 
```

```
588         clEnqueueWriteBuffer(queue, vold_buff, CL_TRUE, 0,2*N*sizeof(double)
                , vold, 0, NULL, NULL);
589         clSetKernelArg(lanczos2, 0, sizeof(cl_double), &alpha);
590         clSetKernelArg(lanczos2, 1, sizeof(cl_double), &beta);
591         clSetKernelArg(lanczos2, 2, sizeof(cl_mem), &v_buff);
592         clSetKernelArg(lanczos2, 3, sizeof(cl_mem), &vold_buff);
593         clSetKernelArg(lanczos2, 4, sizeof(cl_mem), &w_buff);
594         clSetKernelArg(lanczos2, 5, sizeof(cl_mem), &ab_buff);
595         clEnqueueNDRangeKernel(queue,lanczos2,1,NULL,&glob_size,&loc_size,0,
                NULL,NULL);
596         clEnqueueReadBuffer(queue,w_buff, CL_TRUE, 0,2*N*sizeof(double), w,
                0, NULL, NULL);
597         clEnqueueReadBuffer(queue,ab_buff, CL_TRUE, 0,N*sizeof(double), ab,
                0, NULL, NULL);
598
599         b=ab[0];
600         for(j=1;j<N;j++){
601            b+=ab[j];
602         }
603         b=sqrt(b);
604         subdiag[i]=b;
605
606         for(j=0;j<N;j++){
607            P[2*j*N+2*i]=v[2*j];
608            P[2*j*N+2*i+1]=v[2*j+1];
609            vold[2*j]=v[2*j];
610            vold[2*j+1]=v[2*j+1];
611         }
612      }
613
614      clReleaseMemObject(M_buff);
615      clReleaseMemObject(v_buff);
616      clReleaseMemObject(vold_buff);
617      clReleaseMemObject(w_buff);
618      clReleaseMemObject(ab_buff);
619      clReleaseKernel(lanczos1);
620      clReleaseKernel(lanczos2);
621
622      free(w);
623      free(v);
624      free(vold);
625
626
627
628      //QR algorithm
629
630      int chk=0;
631      double mu,conf,conf2=0;
632      double *Q,*A;
633
634      cl_mem A_buff;
635      cl_mem B_buff;
636      cl_mem C_buff;
637
638      glob_size=N*N;
639      glob_size2=N;
640      A_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,N*N*sizeof(double),NULL
             ,NULL);
641      B_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,N*N*sizeof(double),NULL
             ,NULL);
642      C_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,N*N*sizeof(double),NULL
```

```
        ,NULL);
643     v_buff=clCreateBuffer(context,CL_MEM_WRITE_ONLY,N*sizeof(double),NULL,
            NULL);
644     w_buff=clCreateBuffer(context,CL_MEM_WRITE_ONLY,N*sizeof(double),NULL,
            NULL);
645
646     mu=diag[N-1];
647     A=malloc(sizeof(double)*N*N);
648     for(i=0;i<N;i++){
649        for(j=0;j<N;j++){
650           if(i==j){
651              A[i*N+j]=diag[i];
652           }
653           else if(i==j+1){
654              A[i*N+j]=subdiag[i-1];
655           }
656           else if(i==j-1){
657              A[i*N+j]=subdiag[i];
658           }
659           else{
660              A[i*N+j]=0;
661           }
662        }
663        diag[i]-=mu;
664     }
665
666     Q=QR(diag,subdiag);
667     while(chk==0){
668        double *tdiag,*tsubdiag,*Qnew;
669
670        //temp=A*Q
671        clEnqueueWriteBuffer(queue, A_buff, CL_TRUE, 0,N*N*sizeof(double), A
               , 0, NULL, NULL);
672        clEnqueueWriteBuffer(queue, B_buff, CL_TRUE, 0,N*N*sizeof(double), Q
               , 0, NULL, NULL);
673        clSetKernelArg(mat_mul, 0, sizeof(cl_mem), &A_buff);
674        clSetKernelArg(mat_mul, 1, sizeof(cl_mem), &B_buff);
675        clSetKernelArg(mat_mul, 2, sizeof(cl_mem), &C_buff);
676        clEnqueueNDRangeKernel(queue,mat_mul,1,NULL,&glob_size,NULL,0,NULL,
               NULL);
677
678        //Qt=trasp(Q)
679        clEnqueueWriteBuffer(queue, B_buff, CL_TRUE, 0,N*N*sizeof(double), Q
               , 0, NULL, NULL);
680        clSetKernelArg(transpose, 0, sizeof(cl_mem), &B_buff);
681        clSetKernelArg(transpose, 1, sizeof(cl_mem), &A_buff);
682        clEnqueueNDRangeKernel(queue,transpose,1,NULL,&glob_size,NULL,0,NULL
               ,NULL);
683
684        //Ak=Qt*temp
685        clSetKernelArg(mat_mul, 0, sizeof(cl_mem), &A_buff);
686        clSetKernelArg(mat_mul, 1, sizeof(cl_mem), &C_buff);
687        clSetKernelArg(mat_mul, 2, sizeof(cl_mem), &B_buff);
688        clEnqueueNDRangeKernel(queue,mat_mul,1,NULL,&glob_size,NULL,0,NULL,
               NULL);
689
690        tdiag=malloc(sizeof(double)*N);
691        tsubdiag=malloc(sizeof(double)*N);
692
693        clSetKernelArg(transfer, 0, sizeof(cl_mem), &B_buff);
694        clSetKernelArg(transfer, 1, sizeof(cl_mem), &v_buff);
```

```
695         clSetKernelArg(transfer, 2, sizeof(cl_mem), &w_buff);
696         clEnqueueNDRangeKernel(queue,transfer,1,NULL,&glob_size2,NULL,0,NULL
                ,NULL);
697         clEnqueueReadBuffer(queue,v_buff, CL_TRUE, 0,N*sizeof(double), tdiag
                , 0, NULL, NULL);
698         clEnqueueReadBuffer(queue,w_buff, CL_TRUE, 0,N*sizeof(double),
                tsubdiag, 0, NULL, NULL);
699
700         mu=tsubdiag[N-1];
701         tdiag[N-1]=0;
702         Qnew=QR(tdiag,tsubdiag);
703
704         //Q=Q*Qnew
705         clEnqueueWriteBuffer(queue, A_buff, CL_TRUE, 0,N*N*sizeof(double), Q
                , 0, NULL, NULL);
706         clEnqueueWriteBuffer(queue, B_buff, CL_TRUE, 0,N*N*sizeof(double),
                Qnew, 0, NULL, NULL);
707         clSetKernelArg(mat_mul, 0, sizeof(cl_mem), &A_buff);
708         clSetKernelArg(mat_mul, 1, sizeof(cl_mem), &B_buff);
709         clSetKernelArg(mat_mul, 2, sizeof(cl_mem), &C_buff);
710         clEnqueueNDRangeKernel(queue,mat_mul,1,NULL,&glob_size,NULL,0,NULL,
                NULL);
711         clEnqueueReadBuffer(queue,C_buff, CL_TRUE, 0,N*N*sizeof(double), Q,
                0, NULL, NULL);
712         free(Qnew);
713
714         conf=tdiag[0];
715         for(i=0;i<N;i++){
716            if(tdiag[i]<conf)
717               conf=tdiag[i];
718         }
719         free(tdiag);
720         free(tsubdiag);
721         conf+=mu;
722         if(fabs(conf-conf2)< 1e-13)
723            chk=1;
724         conf2=conf;
725         //printf("%f\n",conf2);
726      }
727      *eig0=conf2;
728
729      //free memory
730      free(P);
731      free(diag);
732      free(subdiag);
733      free(A);
734      free(Q);
735
736      //Deallocate resources
737      clReleaseMemObject(A_buff);
738      clReleaseMemObject(B_buff);
739      clReleaseMemObject(C_buff);
740      clReleaseKernel(mat_mul);
741      clReleaseKernel(transpose);
742      clReleaseKernel(transfer);
743      clReleaseCommandQueue(queue);
744      clReleaseProgram(program);
745      clReleaseContext(context);
746
747      end = clock();
748      *time=((double) (end - start)) / CLOCKS_PER_SEC;
```

```
749     printf("OpenCL (GPU) finished!\n");
750     printf("Execution time: %f\n",*time);
751     printf("Lowest eigenvalue: %f\n\n",*eig0);
752 }
753
754
755
756
757 //OpenCL CPU
758
759 void openclCPU(double *time ,double *eig0 ,double *M){
760
761     printf("OpenCL (CPU) started!\n");
762     clock_t start, end;
763     start = clock();
764
765     //opencl initialization
766
767     FILE *f;
768     int i,j;
769     char str[40];
770     char name[40];
771     char *source_str;
772      size_t source_size;
773     sprintf(str,"-cl-std=CL1.2 -D N=%d",N);
774
775     int dev=2;
776     cl_platform_id *platforms;
777     cl_uint num_platforms;
778      cl_device_id device;
779      cl_uint num_devices;
780     cl_context context;
781     cl_command_queue queue;
782     cl_program program;
783     size_t glob_size;
784     size_t glob_size2;
785     size_t loc_size=N;
786
787     clGetPlatformIDs(5,NULL,&num_platforms);
788     platforms=(cl_platform_id*) malloc(sizeof(cl_platform_id)*num_platforms
        );
789     clGetPlatformIDs(num_platforms,platforms,NULL);
790     //printf("Number of Platforms detected: %d\n",num_platforms);
791     for(i=0;i<num_platforms;i++){
792         clGetPlatformInfo(platforms[i],CL_PLATFORM_NAME,sizeof(name),&name,
            NULL);
793         //printf("%s\n",name);
794     }
795
796
797     if(dev==1){
798         clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,1,&device,&
            num_devices);
799         for(i=1;loc_size>256;i++){
800             if(loc_size%i==0)
801                 loc_size=N/i;
802         }
803         //printf("\nAMD GPU selected\n");
804     }
805     else if(dev==2){
806         clGetDeviceIDs(platforms[1],CL_DEVICE_TYPE_CPU,1,&device,&
```

```
            num_devices);
807         for(i=1;loc_size>8192;i++){
808            if(loc_size%i==0)
809               loc_size=N/i;
810         }
811         //printf("\nIntel CPU selected\n");
812      }
813
814      context=clCreateContext( NULL, 1,&device, NULL, NULL, NULL);
815      queue=clCreateCommandQueueWithProperties(context,device,NULL,NULL);
816
817       f=fopen("kernels.cl", "r");
818       source_str = (char*)malloc(MAX_SOURCE_SIZE);
819       source_size = fread( source_str, 1, MAX_SOURCE_SIZE, f);
820       fclose(f);
821      program=clCreateProgramWithSource(context,1,(const char **)&source_str
          ,(const size_t *)&source_size,NULL);
822      clBuildProgram(program,1,&device,str,NULL,NULL);
823      free(source_str);
824
825      cl_kernel lanczos1 = clCreateKernel(program, "lanczos1", NULL);
826      cl_kernel lanczos2 = clCreateKernel(program, "lanczos2", NULL);
827      cl_kernel transpose = clCreateKernel(program, "transpose", NULL);
828      cl_kernel mat_mul = clCreateKernel(program, "mat_mul", NULL);
829      cl_kernel transfer = clCreateKernel(program, "transfer", NULL);
830
831
832
833      //lanczos algorithm
834
835      double a,b=1;
836      double *P, *v, *vold,*w;
837      double *diag,*subdiag,*ab;
838
839      cl_double alpha;
840      cl_double beta;
841      cl_mem M_buff;
842      cl_mem v_buff;
843      cl_mem vold_buff;
844      cl_mem w_buff;
845      cl_mem ab_buff;
846
847      diag=malloc(sizeof(double)*N);
848      subdiag=malloc(sizeof(double)*N);
849      w=calloc(2*N,sizeof(double));
850      v=malloc(sizeof(double)*2*N);
851      vold=calloc(2*N,sizeof(double));
852      P=calloc(2*N*N,sizeof(double));
853      ab=malloc(N*sizeof(double));
854      P[0]=1;
855      P[1]=0;
856      w[0]=P[0];
857      w[1]=P[1];
858
859      glob_size=N;
860      M_buff=clCreateBuffer(context,CL_MEM_READ_ONLY,2*N*N*sizeof(double),
          NULL,NULL);
861      v_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,2*N*sizeof(double),NULL
          ,NULL);
862      vold_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,2*N*sizeof(double),
          NULL,NULL);
```

```
863     w_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,2*N*sizeof(double),NULL
            ,NULL);
864     ab_buff=clCreateBuffer(context,CL_MEM_WRITE_ONLY,N*sizeof(double),NULL,
            NULL);
865
866     for(i=0;i<N;i++){
867
868        beta=b;
869
870        clEnqueueWriteBuffer(queue, M_buff, CL_TRUE, 0,2*N*N*sizeof(double),
               M, 0, NULL, NULL);
871        clEnqueueWriteBuffer(queue, w_buff, CL_TRUE, 0,2*N*sizeof(double), w
               , 0, NULL, NULL);
872        clSetKernelArg(lanczos1, 0, sizeof(cl_double), &beta);
873        clSetKernelArg(lanczos1, 1, sizeof(cl_mem), &M_buff);
874        clSetKernelArg(lanczos1, 2, sizeof(cl_mem), &v_buff);
875        clSetKernelArg(lanczos1, 3, sizeof(cl_mem), &w_buff);
876        clSetKernelArg(lanczos1, 4, sizeof(cl_mem), &ab_buff);
877        clEnqueueNDRangeKernel(queue,lanczos1,1,NULL,&glob_size,&loc_size,0,
               NULL,NULL);
878        clEnqueueReadBuffer(queue,v_buff, CL_TRUE, 0,2*N*sizeof(double), v,
               0, NULL, NULL);
879        clEnqueueReadBuffer(queue,ab_buff, CL_TRUE, 0,N*sizeof(double), ab,
               0, NULL, NULL);
880
881        a=ab[0];
882        for(j=1;j<N;j++){
883           a+=ab[j];
884        }
885        diag[i]=a;
886
887        if(i==0)
888           beta=0;
889
890        alpha=a;
891
892        clEnqueueWriteBuffer(queue, vold_buff, CL_TRUE, 0,2*N*sizeof(double)
               , vold, 0, NULL, NULL);
893        clSetKernelArg(lanczos2, 0, sizeof(cl_double), &alpha);
894        clSetKernelArg(lanczos2, 1, sizeof(cl_double), &beta);
895        clSetKernelArg(lanczos2, 2, sizeof(cl_mem), &v_buff);
896        clSetKernelArg(lanczos2, 3, sizeof(cl_mem), &vold_buff);
897        clSetKernelArg(lanczos2, 4, sizeof(cl_mem), &w_buff);
898        clSetKernelArg(lanczos2, 5, sizeof(cl_mem), &ab_buff);
899        clEnqueueNDRangeKernel(queue,lanczos2,1,NULL,&glob_size,&loc_size,0,
               NULL,NULL);
900        clEnqueueReadBuffer(queue,w_buff, CL_TRUE, 0,2*N*sizeof(double), w,
               0, NULL, NULL);
901        clEnqueueReadBuffer(queue,ab_buff, CL_TRUE, 0,N*sizeof(double), ab,
               0, NULL, NULL);
902
903        b=ab[0];
904        for(j=1;j<N;j++){
905           b+=ab[j];
906        }
907        b=sqrt(b);
908        subdiag[i]=b;
909
910        for(j=0;j<N;j++){
911           P[2*j*N+2*i]=v[2*j];
912           P[2*j*N+2*i+1]=v[2*j+1];
```

```
913             vold[2*j]=v[2*j];
914             vold[2*j+1]=v[2*j+1];
915         }
916     }
917 
918     clReleaseMemObject(M_buff);
919     clReleaseMemObject(v_buff);
920     clReleaseMemObject(vold_buff);
921     clReleaseMemObject(w_buff);
922     clReleaseMemObject(ab_buff);
923     clReleaseKernel(lanczos1);
924     clReleaseKernel(lanczos2);
925 
926     free(w);
927     free(v);
928     free(vold);
929 
930 
931 
932     //QR algorithm
933 
934     int chk=0;
935     double mu,conf,conf2=0;
936     double *Q,*A;
937 
938     cl_mem A_buff;
939     cl_mem B_buff;
940     cl_mem C_buff;
941 
942     glob_size=N*N;
943     glob_size2=N;
944     A_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,N*N*sizeof(double),NULL
        ,NULL);
945     B_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,N*N*sizeof(double),NULL
        ,NULL);
946     C_buff=clCreateBuffer(context,CL_MEM_READ_WRITE,N*N*sizeof(double),NULL
        ,NULL);
947     v_buff=clCreateBuffer(context,CL_MEM_WRITE_ONLY,N*sizeof(double),NULL,
        NULL);
948     w_buff=clCreateBuffer(context,CL_MEM_WRITE_ONLY,N*sizeof(double),NULL,
        NULL);
949 
950     mu=diag[N-1];
951     A=malloc(sizeof(double)*N*N);
952     for(i=0;i<N;i++){
953         for(j=0;j<N;j++){
954             if(i==j){
955                 A[i*N+j]=diag[i];
956             }
957             else if(i==j+1){
958                 A[i*N+j]=subdiag[i-1];
959             }
960             else if(i==j-1){
961                 A[i*N+j]=subdiag[i];
962             }
963             else{
964                 A[i*N+j]=0;
965             }
966         }
967         diag[i]-=mu;
968     }
```

```
969
970     Q=QR(diag,subdiag);
971     while(chk==0){
972         double *tdiag,*tsubdiag,*Qnew;
973
974         //temp=A*Q
975         clEnqueueWriteBuffer(queue, A_buff, CL_TRUE, 0,N*N*sizeof(double), A
                , 0, NULL, NULL);
976         clEnqueueWriteBuffer(queue, B_buff, CL_TRUE, 0,N*N*sizeof(double), Q
                , 0, NULL, NULL);
977         clSetKernelArg(mat_mul, 0, sizeof(cl_mem), &A_buff);
978         clSetKernelArg(mat_mul, 1, sizeof(cl_mem), &B_buff);
979         clSetKernelArg(mat_mul, 2, sizeof(cl_mem), &C_buff);
980         clEnqueueNDRangeKernel(queue,mat_mul,1,NULL,&glob_size,NULL,0,NULL,
                NULL);
981
982         //Qt=trasp(Q)
983         clEnqueueWriteBuffer(queue, B_buff, CL_TRUE, 0,N*N*sizeof(double), Q
                , 0, NULL, NULL);
984         clSetKernelArg(transpose, 0, sizeof(cl_mem), &B_buff);
985         clSetKernelArg(transpose, 1, sizeof(cl_mem), &A_buff);
986         clEnqueueNDRangeKernel(queue,transpose,1,NULL,&glob_size,NULL,0,NULL
                ,NULL);
987
988         //Ak=Qt*temp
989         clSetKernelArg(mat_mul, 0, sizeof(cl_mem), &A_buff);
990         clSetKernelArg(mat_mul, 1, sizeof(cl_mem), &C_buff);
991         clSetKernelArg(mat_mul, 2, sizeof(cl_mem), &B_buff);
992         clEnqueueNDRangeKernel(queue,mat_mul,1,NULL,&glob_size,NULL,0,NULL,
                NULL);
993
994         tdiag=malloc(sizeof(double)*N);
995         tsubdiag=malloc(sizeof(double)*N);
996
997         clSetKernelArg(transfer, 0, sizeof(cl_mem), &B_buff);
998         clSetKernelArg(transfer, 1, sizeof(cl_mem), &v_buff);
999         clSetKernelArg(transfer, 2, sizeof(cl_mem), &w_buff);
1000        clEnqueueNDRangeKernel(queue,transfer,1,NULL,&glob_size2,NULL,0,NULL
                ,NULL);
1001        clEnqueueReadBuffer(queue,v_buff, CL_TRUE, 0,N*sizeof(double), tdiag
                , 0, NULL, NULL);
1002        clEnqueueReadBuffer(queue,w_buff, CL_TRUE, 0,N*sizeof(double),
                tsubdiag, 0, NULL, NULL);
1003
1004        mu=tsubdiag[N-1];
1005        tdiag[N-1]=0;
1006        Qnew=QR(tdiag,tsubdiag);
1007
1008        //Q=Q*Qnew
1009        clEnqueueWriteBuffer(queue, A_buff, CL_TRUE, 0,N*N*sizeof(double), Q
                , 0, NULL, NULL);
1010        clEnqueueWriteBuffer(queue, B_buff, CL_TRUE, 0,N*N*sizeof(double),
                Qnew, 0, NULL, NULL);
1011        clSetKernelArg(mat_mul, 0, sizeof(cl_mem), &A_buff);
1012        clSetKernelArg(mat_mul, 1, sizeof(cl_mem), &B_buff);
1013        clSetKernelArg(mat_mul, 2, sizeof(cl_mem), &C_buff);
1014        clEnqueueNDRangeKernel(queue,mat_mul,1,NULL,&glob_size,NULL,0,NULL,
                NULL);
1015        clEnqueueReadBuffer(queue,C_buff, CL_TRUE, 0,N*N*sizeof(double), Q,
                0, NULL, NULL);
1016        free(Qnew);
```

```
        conf=tdiag[0];
        for(i=0;i<N;i++){
            if(tdiag[i]<conf)
                conf=tdiag[i];
        }
        free(tdiag);
        free(tsubdiag);
        conf+=mu;
        if(fabs(conf-conf2)< 1e-13)
            chk=1;
        conf2=conf;
        //printf("%f\n",conf2);
    }
    *eig0=conf2;

    //free memory
    free(P);
    free(diag);
    free(subdiag);
    free(A);
    free(Q);

    //Deallocate resources
    clReleaseMemObject(A_buff);
    clReleaseMemObject(B_buff);
    clReleaseMemObject(C_buff);
    clReleaseKernel(mat_mul);
    clReleaseKernel(transpose);
    clReleaseKernel(transfer);
    clReleaseCommandQueue(queue);
    clReleaseProgram(program);
    clReleaseContext(context);

    end = clock();
    *time=((double) (end - start)) / CLOCKS_PER_SEC;
    printf("OpenCL (CPU) finished!\n");
    printf("Execution time: %f\n",*time);
    printf("Lowest eigenvalue: %f\n\n",*eig0);
}
```

A.2 Kernels

```
//enable support to double type
#pragma OPENCL EXTENSION cl_khr_fp64: enable


//2 component vector to hold the real and imaginary parts of a complex
    number:
typedef double2 cdouble;

#define I ((cdouble)(0.0, 1.0))


/*
 * Return Real (Imaginary) component of complex number:
 */
inline double  real(cdouble a){
     return a.x;
```

```
16 }
17 inline double  imag(cdouble a){
18      return a.y;
19 }
20
21 inline cdouble cc(cdouble z){
22    z.s1=-z.s1;
23    return z;
24 }
25
26
27 /*
28  * Multiply two complex numbers:
29  *
30  *  a = (aReal + I*aImag)
31  *  b = (bReal + I*bImag)
32  *  a * b = (aReal + I*aImag) * (bReal + I*bImag)
33  *        = aReal*bReal +I*aReal*bImag +I*aImag*bReal +I^2*aImag*bImag
34  *        = (aReal*bReal - aImag*bImag) + I*(aReal*bImag + aImag*bReal)
35  */
36 inline cdouble  cmult(cdouble a, cdouble b){
37     return (cdouble)( a.s0*b.s0 - a.s1*b.s1, a.s0*b.s1 + a.s1*b.s0);
38 }
39
40
41
42
43 // first part of the Lanczos algorithm iteration
44
45 __kernel void lanczos1(const double beta, __global cdouble *A, __global
       cdouble *v, __global cdouble *w, __global double *alpha)
46 {
47    int id=get_global_id(0);
48    int gid=get_group_id(0);
49    int ls=get_local_size(0);
50    int lid=get_local_id(0);
51    int ngr=get_num_groups(0);
52
53    cdouble temp1,temp2;
54    temp2=0;
55    temp1=w[id]/beta;
56    v[id]=temp1;
57
58    int i;
59    barrier(CLK_LOCAL_MEM_FENCE);
60
61    for(i=0;i<N;i++){
62       temp2+=cmult(A[N*id+i],v[i]);
63    }
64    w[id]=temp2;
65
66    alpha[id]=real(cmult(cc(temp2),temp1));
67 }
68
69
70 // second part of the Lanczos algorithm iteration
71
72 __kernel void lanczos2(const double alpha,const double beta, __global
       cdouble *v, __global cdouble *v2, __global cdouble *w, __global double
        *ab)
73 {
```

```
74
75      int id=get_global_id(0);
76      int gid=get_group_id(0);
77      int ls=get_local_size(0);
78      int lid=get_local_id(0);
79      int ngr=get_num_groups(0);
80
81      cdouble temp1;
82      temp1=w[id]-alpha*v[id]-beta*v2[id];
83      w[id]=temp1;
84
85      ab[id]=real(cmult(cc(temp1),temp1));
86  }
87
88
89
90  // transpose NxN matrix
91
92  __kernel void transpose(__global double *A,__global double *At)
93  {
94      int id=get_global_id(0);
95
96      int i,j;
97      i=id/N;
98      j=id-i*N;
99      At[j*N+i]=A[id];
100 }
101
102
103 // multiply two NxN matrices
104
105 __kernel void mat_mul(__global double *A,__global double *B,__global
        double *C)
106 {
107     int id=get_global_id(0);
108
109     int i,j,k;
110     double temp=0;
111     i=id/N;
112     j=id-i*N;
113     for(k=0;k<N;k++){
114        temp+=A[i*N+k]*B[k*N+j];
115     }
116     C[id]=temp;
117 }
118
119
120 // auxiliary function for QR iteration
121
122 __kernel void transfer(__global double *A,__global double *diag,__global
        double *subdiag)
123 {
124     int id=get_global_id(0);
125
126     double temp,mu;
127     mu=A[N*N-1];
128     temp=A[id*N+id]-mu;
129     diag[id]=temp;
130     temp=A[id*N+id+1];
131     subdiag[id]=temp;
132     if(id==N-1)
```

```
133         subdiag[id]=mu;
134 }
```

B Schrödinger equation code

```
1
2  ////////////////////////////////////////////////////////////////////////
3  //
4  // The code generate the matrix corresponding to the 2D schrodinger
5  // equation with any potential.
6  // The matrix is written in the file mh.mtx in the matrix market
7  // format. Then the Lanczos algorithm from ViennaCL is called and
8  // the lowest eigenpair is found. The eigenvector is written in a
9  // suitable form to the file "psi.txt" so as to be plotted.
10 //
11 ////////////////////////////////////////////////////////////////////////
12
13
14 // include necessary system headers
15 #include <iostream>
16 #include <string>
17 #include <iomanip>
18 #include <fstream>
19 #include <cmath>
20
21 // tell viennaCL to use OpenCL
22 #define VIENNACL_WITH_OPENCL
23
24 // include basic scalar and vector types of ViennaCL
25 #include "viennacl/scalar.hpp"
26 #include "viennacl/vector.hpp"
27 #include "viennacl/matrix.hpp"
28 #include "viennacl/matrix_proxy.hpp"
29 #include "viennacl/compressed_matrix.hpp"
30 #include "viennacl/linalg/lanczos.hpp"
31 #include "viennacl/io/matrix_market.hpp"
32
33 // If you GPU does not support double precision, use `float` instead of `
      double`:
34 typedef double ScalarType;
35
36
37 viennacl::vector<ScalarType> ground_state();
38
39
40 int main()
41 {
42    //Schrodinger equation generator
43
44    int N,i,j;
45    double delta, lim, temp1, V, x, y;
46    std::ofstream pot,sch;
47
48    N=401;
49    lim=4;
50    delta=2*(double)lim/(N-1);
51
52    pot.open("pot.txt");
```

```
53      sch.open("sch.mtx");
54      sch << N*N << "   " << N*N << "  " << N*N*5-2*(N-1)-4 << std::endl;
55
56      for(i=0;i<N;i++){
57          for(j=0;j<N;j++){
58
59              x=-lim+i*delta;
60              y=-lim+j*delta;
61              temp1=(double)1/(delta*delta);
62
63              //defining the potential energy
64
65              //V=0.5*x*x+0.5*y*y;                          //2D decoupled harmonic
                    oscillator
66              V=-(x*x+y*y)+0.125*pow(x*x+y*y,2);          //central mexican hat
67              //V=-x*x+0.125*pow(x,4)-y*y+0.125*pow(y,4);     //double mexican
                    hat
68              /*if(sqrt(x*x+y*y)>0.1)                       //regularised
                    electrostatic potential
69                 V=-(double)3/sqrt(x*x+y*y);
70              else
71                 V=-10;*/
72
73
74              if(i+1+j*N+1<=N*N)
75                 sch << i+j*N+1 << "  " << i+1+j*N+1 << "  " << temp1 << std::
                       endl;
76              if(i+j*N>0)
77                 sch << i+j*N+1 << "  " << i-1+j*N+1 << "  " << temp1 << std::
                       endl;
78              if(i+(j+1)*N+1<=N*N)
79                 sch << i+j*N+1 << "  " << i+(j+1)*N+1 << "   " << temp1 << std
                       ::endl;
80              if(i+(j-1)*N+1>0)
81                 sch << i+j*N+1 << "  " << i+(j-1)*N+1 << "   " << temp1 << std
                       ::endl;
82              sch << i+j*N+1 << "  " << i+j*N+1 << " " << -V-4*temp1 << std::
                    endl;
83
84              pot << x << "  " << y << " " << V << std::endl;
85          }
86      }
87
88      sch.close();
89      pot.close();
90
91
92      //ground state computation
93      viennacl::vector<ScalarType> psi;
94      psi=ground_state();
95      std::cout << "Remember to change sign to the eigenvalue!" << std::endl;
96
97
98      //output
99      std::ofstream out;
100     out.open("psi.txt");
101     for(i=0;i<N;i++){
102         for(j=0;j<N;j++){
103             out << -lim+i*delta << "   " << -lim+j*delta << "  " << pow(psi[i
                    +j*N],2)/pow(delta,2) << std::endl;
104         }
```

```
105     }
106     out.close();
107
108     return 0;
109 }
110
111
112
113
114
115 /*
     ============================================================================

116     Copyright (c) 2010-2016, Institute for Microelectronics,
117                              Institute for Analysis and Scientific
                                     Computing,
118                              TU Wien.
119     Portions of this software are copyright by UChicago Argonne, LLC.
120                              -----------------
121                   ViennaCL - The Vienna Computing Library
122                              -----------------
123     Project Head:    Karl Rupp                   rupp@iue.tuwien.ac.at
124     (A list of authors and contributors can be found in the PDF manual)
125     License:         MIT (X11), see file LICENSE in the base directory
126 ============================================================================
      */
127
128 viennacl::vector<ScalarType> ground_state(){
129
130     viennacl::vector<ScalarType> approx_eigenvector;
131     std::vector< std::map<unsigned int, ScalarType> > host_A;
132     if (!viennacl::io::read_matrix_market_file(host_A, "sch.mtx"))
133     {
134        std::cout << "Error reading Matrix file" << std::endl;
135        return approx_eigenvector;
136     }
137
138     viennacl::compressed_matrix<ScalarType> A;
139     viennacl::copy(host_A, A);
140     viennacl::linalg::lanczos_tag ltag(0.75,     // Select a power of 0.75
         as the tolerance for the machine precision.
141                                        1,       // Compute (approximations to) the 1
                                              largest eigenvalues
142                                        1,       // use full reorthogonalization
143                                        400);    // Maximum size of the Krylov space
144
145     std::cout << "Running Lanczos algorithm (with eigenvectors). This might
          take a while..." << std::endl;
146     viennacl::matrix<ScalarType> approx_eigenvectors_A(A.size1(), ltag.
         num_eigenvalues());
147     std::vector<ScalarType> lanczos_eigenvalues = viennacl::linalg::eig(A,
         approx_eigenvectors_A, ltag);
148
149     for (std::size_t i = 0; i< lanczos_eigenvalues.size(); i++)
150     {
151        std::cout << "Approx. eigenvalue " << std::setprecision(7) <<
             lanczos_eigenvalues[i]<<std::endl;
152        // test approximated eigenvector by computing A*v:
153        approx_eigenvector = viennacl::column(approx_eigenvectors_A,
             static_cast<unsigned int>(i));
154        viennacl::vector<ScalarType> Aq = viennacl::linalg::prod(A,
```

```
            approx_eigenvector);
155         std::cout << " (" << viennacl::linalg::inner_prod(Aq,
            approx_eigenvector) << " for <Av,v> with approx. eigenvector v)"
             << std::endl;
156     }
157     return approx_eigenvector;
158 }
```