An object-oriented implementation of a solver of the time-dependent Schrödinger equation using the CUDA technology

Tomasz Dziubak, Jacek Matulewski *

Instytut Fizyki, Uniwersytet Mikołaja Kopernika, ul. Grudziądzka 5, 87-100 Toruń, Poland

ARTICLE INFO

Article history:
Received 21 October 2010
Received in revised form 23 November 2011
Accepted 28 November 2011
Available online 9 December 2011

Keywords:
Time dependent Schrödinger equation
Quantum dynamics
CUDA
GPGPU computing
Complex Chebyshev polynomial and FFT method

ABSTRACT

We present a set of C++ classes which allow one to use the graphics card processor's cores for quantum ab initio simulations, i.e. a direct solving of the time-dependent Schrödinger equation, gaining the benefits from the parallel architecture of the graphical processor units. We use the Chebyshev polynomial and FFT algorithm. The solution is based on NVIDIA CUDA technology. The speed-up factor in the test runs of our classes performed using the graphics card processor can even be of order of 300 in comparison with the test runs using only the single core of CPU. Not only the Schrödinger equation can be integrated using the presented solver. With only small changes it can be used for solving the nonlinear Gross–Pitaevskii equation of BEC's dynamics, the heat equation, the diffusion equation or other parabolic partial differential equations of second order.1

Program summary

Program title: QnDynCUDA
Catalogue identifier: AELE_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AELE_v1_0.html
Program obtainable from: CPC Program Library, Queen’s University, Belfast, N. Ireland
No. of lines in distributed program, including test data, etc.: 101 359
No. of bytes in distributed program, including test data, etc.: 3 165 228
Distribution format: tar.gz
Programming language: C++, C for CUDA
Computer: Graphics card with CUDA technology recommended
Operating system: No limits (tested on 32-bit and 64-bit Windows and 64-bit Linux)
Has the code been vectorized or parallelized?: Yes, number of processors used – one CPU core and all CUDA cores of the selected processor of graphics card
RAM: Dependent on user's parameters, typically between several tens of megabytes and several gigabytes (this concerns also the graphics card's memory)
Supplementary material: Test input and output files (approx. 3.4 Gigabytes) are available
Classification: 2.7, 6.5
Nature of problem: Solving the time-dependent Schrödinger equation.
Solution method: FFT and Chebyshev polynomial algorithm, CUDA technology.
Running time: Every test example included in the distribution package takes approximately an hour or so if the GPU is engaged and a day or so if only CPU is used.

© 2011 Elsevier B.V. All rights reserved.

* This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding author.
E-mail address: jacek@phys.uni.torun.pl (J. Matulewski).
1 The presented code was prepared as a part of a PhD thesis of TD under the supervision of JM.

0010-4655/– see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2011.11.026
1. Introduction

Every modern computer possesses a large, but quite often not used power of its graphics card processor. This Graphical Processor Unit (GPU) was first designed to accelerate 3D graphics’ rendering, supporting the CPU. However, it evolved into a specialized, programmable processor unit acting independently of CPU, adapted to parallelly executing small pre-defined functions (shaders). In the last twenty years we have been dealing with several generations of GPUs and graphics cards. The direction of their evolution was to make them more and more programmable. The latest editions of GPU, called also General Purpose GPU (GPGPU), apart from their main aim of graphics rendering, got a new ability of executing programs not necessarily connected with the graphics, such as many-particle or soft-body simulations (gases, fluids, galaxies, hair and so on) [1]. Also the possibility of the two-way memory transfer between the main computer (host) and the graphics card is very important in this context. This new ability of the graphics card called CUDA (Compute Unified Device Architecture) in NVIDIA products [2] or ATI Stream in ATI products [3], allows one to use the GPUs for parallel calculations, also in physics.

A set of C++ classes presented below may be used for solving a two-dimensional second order parabolic equation, such as the time-dependent Schrödinger equation. We use it to perform two-dimensional quantum simulations of atomic systems with customized vector and scalar potentials. The Schrödinger equation we implement takes the general form [4]:

\[
i \frac{\partial}{\partial t} \Psi(x, y, t) = S_2 \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \Psi(x, y, t) + S_{1x}(x, y, t) \frac{\partial}{\partial x} \Psi(x, y, t) + S_{1y}(x, y, t) \frac{\partial}{\partial y} \Psi(x, y, t) + S_0(x, y, t) \Psi(x, y, t).
\]

(1)

Below we will refer to the \( \Psi \) function, the solution of the above equation, calling it the wave function. The Schrödinger equation of this form may also be used for three-dimensional simulations in cylindrical coordinates for the systems with an axial symmetry (the angle coordinate is excluded) [5].

The main advantage of the presented solution is using functions written in the C for CUDA language, which allowed us to use the power of the graphics cards compatible with CUDA technology [2].

Since the algorithm we use is rather known [6–9] we limit ourselves just to its concise description. We split the evolution operator into a series of recursively calculated complex Chebyshev polynomials. Note that the Hamiltonian operator which is the argument of the Chebyshev polynomials has to be renormalized, therefore it cannot be unbounded in this method (e.g., the Coulomb potential has to be regularized in order to remove the singularity). The coefficients of the expansion are simply the Bessel functions. This way of expanding the evolution operator has several important advantages: the time step used in the calculations may be relatively large in comparison with other methods [8], the recursive way of calculating the Chebyshev polynomials is favorable. Moreover, the errors of computing the evolution operator are distributed evenly over all eigenstates [6].

For computing the first- and second-order derivatives in the Hamiltonian we use the Fast Fourier Transform (FFT) [7]. Thus we transform the wavefunction into the momentum representation, in which calculating the derivatives is just multiplying by a suitable power of momentum and then we return to the position representation. This means that the \( n \)-th derivative of the function \( \Psi \) is calculated using the following formula:

\[
\left(-i\frac{\partial}{\partial x}\right)^n \Psi(x, y, t) = FFT^{-1}\left(p_x^n FFT(\Psi(x, y, t))\right),
\]

(2)

where \( \Psi \) denotes the wavefunction and the \( p_x \) is the \( x \) component of the momentum.

Note the \((-i)^n\) appearing on the left-hand side of Eq. (2).

2. The structure of the graphics card memory

The graphics card’s architecture was designed for solving problems in which numerous but relatively small portions of data should be treated in an identical way. In this scenario the same functions are executed parallelly many times for different sets of data. As already mentioned, a typical usage of such an approach is performing calculations related to 3D graphics’ rendering (then the data are a single vertex structure), which is of course the reason for which specialized GPU were designed. The possibility of programming of so-called shader functions executed by GPU, appeared already in the 3-rd generation of the graphics cards, allowing one to customize the rendering process. However, CUDA is a general-purpose architecture and its usage no longer needs to be connected with graphics.

Before presenting the implementation details it is important to point out that the graphics cards have their own memory. The consequence of this fact is that the programmer wishing to use the GPU for calculations has to take into account that copying the data from the host computer’s memory to the graphics card’s memory and vice versa is a time consuming process the number of which should be carefully optimized. In fact the main optimization work which should be done in any CUDA implementation is to just minimize the number of such data transfers. Moreover, the memory of the graphics card has its internal structure. All threads running on GPU have access to the global memory of the graphics card. It is the only memory accessible from the host computer. The threads are organized in a grid which contain blocks. The threads in one block share another memory, called simply the shared memory. The global memory is the most capacious one. At the same time the access to this kind of memory is slowest. The shared memory is much smaller, but much faster. Thus not only the data transfer from the host’s memory to the global memory, but also the transfer between the global memory and the shared one (internal within the graphics card) should be optimized.

It should be noted that the most fundamental limitation of the size of the spatial grid comes from the size of the graphics card memory. For 32-bit systems the maximal memory size which can be addressed at all is 4 GB. The memories of comparable sizes are used in current graphics cards. To our knowledge the maximal size in one graphics card is at the moment 6 GB (Tesla M20x0). For such a memory in our implementation the maximal spatial square grid size is \( 2048 \times 2048 \). This is also the reason for which we have limited ourselves to the two-dimensional case (alternatively the three-dimensional one with an axial symmetry).
The user of the presented set of classes (see Fig. 1) has to write its own descendant class inherited from the abstract class `equation_base`. This class contains a set of pure virtual member functions (methods) describing the coefficients of the Hamiltonian (or more generally the coefficient of the partial equation of the second order). There are in particular two methods named `S1x` and `S1y`. Their names correspond to the functions which they implement, standing in front of the first-order derivatives on the right-hand side of Eq. (1). An important pure virtual member function of `equation_base` class is `V`, which describes the binding potential of an atomic system. In addition to that, the class `equation_base` possesses the virtual, although not pure virtual, method `S0`, which by default implements the formula

\[ S_0(x, y, t) = V(x, y, t) + U_{ab}(x, y). \]  

The function `U_abs`, defined in the `equation_base` class and used in the above formula, implements the complex absorbing potentials (standard negative imaginary potential [10] and power negative complex potential [11]). To control it, a set of parameters are defined as field members of `equation_base` class (see Table 1). Their values are read by the member function `read_parameters` from the file of the structure presented in Table 2. The `read_parameters` is not a virtual function, thus it can be hidden by a new version of this method in the descendant class to read the user-defined set of parameters.

In some cases, for example while implementing the below presented Hamiltonian in the velocity gauge (the Hamiltonian takes then the form called `pA`) or using the so-called imaginary optical potential, the member function `S0` needs to be overridden to include additional terms. Another virtual, but not pure virtual, member function defined in `equation_base` is the function `T`, which is used for transforming the wavefunction to the given gauge. This is implemented in the `equation_base` class to return the value 1, but for example in the above-mentioned velocity gauge it should be overridden to implement the function `exp(-iF(t))`.

The user should implement his own `S` functions characterizing the quantum system he want to investigate. To make it easier this function may be implemented in pure C++ with no need to write any kernel's code. An advanced user may move this functions in kernels run on GPU. It will save significant time spent on transferring the data from host's memory to the graphics card memory. However, a knowledge of CUDA technology is then required.

The class `chebyshev_solver` class inherits from the `propagator` class (see Fig. 1). The latter is an abstract class which defines only one member function `step_forward` responsible for performing one step of simulation. The arguments of this method are: the pointer to the table containing the wave function (`complex_array_2d<__FTYPE>* Psi_value`), the current simulation time and the parameter `copyFromDeviceToHost` of `bool` type determining whether the newly calculated wave function should be copied from the graphics card's memory to the host's memory. The `propagator` class makes it possible to easily change the algorithm of solving the time-dependent Schrödinger equation without a need to modify other classes.

The heart of the `chebyshev_solver` class, as well as of the whole package, is the member function `step_forward` overriding the above-mentioned pure virtual method defined in the `propagator` class. It implements the algorithm of Chebyshev polynomials and FFT (see the description above). This method calls the `Hamiltonian` method, which is responsible for calculating the result of acting of the Hamilton operator on the current value of the wavefunction. In turn this method calls the `KineticOperator` method, which yields the results of action of the part of the Hamiltonian containing the derivate on the wavefunction, namely:

\[ S_2 \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \Psi(x, y, t) + S_{1x}(x, y, t) \frac{\partial}{\partial x} \Psi(x, y, t) + S_{1y}(x, y, t) \frac{\partial}{\partial y} \Psi(x, y, t). \]  

Fig. 1. The UML diagram of the presented set of classes. The supporting classes as well as the classes from the example are also included. The bottom row contains type aliases defined in the package for the integer type (`__NTYPE`), the unsigned integer type (`__OTYPE`) and complex numbers type (`fcomplex`). The latter is the instance of the complex template class. Also the `__FTYPE` macro implementing the single or double precision floating point numbers is defined depending on the user choice.
Table 1
Public and protected data members and member functions of the `equation_base` class.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcomplex S2</td>
<td>The value of the coefficient $S_2$ of Eq. (1).</td>
</tr>
<tr>
<td>grid_parameters* gp</td>
<td>The pointer to the instance of the <code>grid_parameters</code> class.</td>
</tr>
<tr>
<td>__OTYPE abs_pot_type</td>
<td>The type of the absorbing potential:</td>
</tr>
<tr>
<td></td>
<td>0 – no absorbing potential,</td>
</tr>
<tr>
<td></td>
<td>1 – negative imaginary potential (see [10]),</td>
</tr>
<tr>
<td></td>
<td>2 – negative complex potential (see [11]).</td>
</tr>
<tr>
<td>__FTYPE Uo_abs_x</td>
<td>The amplitude of the imaginary part of the absorbing potential in the direction of the $x$ axis, respectively.</td>
</tr>
<tr>
<td>__FTYPE Uo_abs_y</td>
<td>Note: The __FTYPE type is defined as float or double depending on DOUBLE_PRECISION_USAGE constant.</td>
</tr>
<tr>
<td>__FTYPE a_abs_x</td>
<td>The distances from the edges of the spatial grid at which the absorbing potentials is effective.</td>
</tr>
<tr>
<td>__FTYPE a_abs_y</td>
<td></td>
</tr>
<tr>
<td>__FTYPE Uor_abs_x</td>
<td>The amplitude of the real part of the complex absorbing potential in the direction of the $x$ axis, respectively.</td>
</tr>
<tr>
<td>__FTYPE Uor_abs_y</td>
<td></td>
</tr>
<tr>
<td>__FTYPE Nr_abs</td>
<td>The exponentials of the real and imaginary parts of the absorbing potential.</td>
</tr>
<tr>
<td>__FTYPE Ni_abs</td>
<td></td>
</tr>
</tbody>
</table>

Public and protected member functions

```cpp
equation_base(grid_parameters* _gp);
virtual ~equation_base();
void read_parameters (ifstream &filename);
virtual fcomplex S1x (__FTYPE x, __FTYPE y, __FTYPE t) const = 0;
virtual fcomplex S1y (__FTYPE x, __FTYPE y, __FTYPE t) const = 0;
virtual fcomplex T (__FTYPE x, __FTYPE y, __FTYPE t) const;
virtual __FTYPE V (__FTYPE x, __FTYPE y, __FTYPE t) const = 0;
fcomplex U_abs (__FTYPE x, __FTYPE y, __FTYPE t) const;
virtual fcomplex S0 (__FTYPE x, __FTYPE y, __FTYPE t) const;
```

The constructor of the class (see listing 1 for example of its use).
The destructor of the class.
The function reading the parameters from the file (see Table 4). It raises the exception of `runtime_error` type if an error has occurred.
The pure virtual functions implementing the coefficients $S_{1x}$ and $S_{1y}$ standing in front of the 1-st order derivatives of Eq. (1).
The transformation operator used for calculating the observables.
The pure virtual member function to be defined in the descendant class describing the binding potential of the system.
The negative complex absorbing potential (see [10,11]).
The virtual method defined in `equation_base` class as a sum of the pure virtual function of the potential $V$ and the predefined absorbing potential $U_{abs}$.

Table 2
The sequence of the parameters which should be placed in the file read by `equation_base` class with additional parameters defined in `equation_pA` class. The meaning of the parameter may be determined from Table 3 (based on the name of the field storing the value of the parameter).

<table>
<thead>
<tr>
<th>The name of the field of <code>equation_pA</code> class storing the value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs_pot_type</td>
<td>See Table 1.</td>
</tr>
<tr>
<td>Uo_abs_x</td>
<td>See Table 1.</td>
</tr>
<tr>
<td>a_abs_x</td>
<td>See Table 1.</td>
</tr>
<tr>
<td>Uo_abs_y</td>
<td>See Table 1.</td>
</tr>
<tr>
<td>a_abs_y</td>
<td>See Table 1.</td>
</tr>
<tr>
<td>Uor_abs_x</td>
<td>See Table 1.</td>
</tr>
<tr>
<td>Uor_abs_y</td>
<td>See Table 1.</td>
</tr>
<tr>
<td>Nr_abs</td>
<td>See Table 1.</td>
</tr>
<tr>
<td>Ni_abs</td>
<td>See Table 1.</td>
</tr>
</tbody>
</table>

The parameters added in `equation_pA` class

<table>
<thead>
<tr>
<th>potential_type</th>
<th>The possible values are:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 – no potential,</td>
</tr>
<tr>
<td></td>
<td>1 – the radial square potential well,</td>
</tr>
<tr>
<td></td>
<td>2 – the model potential $V_0/\sqrt{a^2 + x^2 + y^2}$.</td>
</tr>
<tr>
<td>Vo</td>
<td>The potential depth ($V_0$, should be negative for binding pot).</td>
</tr>
<tr>
<td>a</td>
<td>The potential width ($a$ has to be positive).</td>
</tr>
<tr>
<td>Box</td>
<td>The amplitude of the $x$ element of the electric field vector.</td>
</tr>
<tr>
<td>wx</td>
<td>The frequency of the $x$ element of the electric field vector.</td>
</tr>
<tr>
<td>Boy</td>
<td>The amplitude of the $y$ element of the electric field vector.</td>
</tr>
<tr>
<td>wy</td>
<td>The frequency of the $y$ element of the electric field vector.</td>
</tr>
</tbody>
</table>
In order to do that the first action of KineticOperator method is to calculate the FFT of the wavefunction (see Eq. (2)). Then it multiplies the result, i.e. the wavefunction in the momentum space, separately by the components of the momentum and by its square. Three matrices are then transformed back to the position space and summed up with the proper coefficients. After the thread returns to the Hamiltonian method, the value of the potential (strictly speaking the S0 function) multiplied by the wavefunction is also added.

In the chebyshev_solver class we have defined the wrappers of the equation_base's members S2, S6x, and S1y in order to incorporate the factors coming from the usage of the FFT to calculate the derivatives (see the comment following Eq. (2)). Thus defining them, we multiply the original S2 by minus one and the original S6x and S1y by the imaginary unit.

Additionally, to speed up the calculations, the values of S6x, S1y and S0 member functions for all spatial grid nodes are stored in matrices defined in chebyshev_solver class, which is the only place where they are used. They are updated once at every time step.

The Bessel functions in the chebyshev_solver class were calculated using the Numerical Recipes for C subroutines [12]. The FFT implementation for CPU is the modified version of the Paul Bourke's one (cf. [13]).

4. The kernels

The above-described code was prepared to run on the CPU. However, some of the chebyshev_solver class member functions have their counterparts which use the functions called kernels written in C for CUDA and designed for running on the GPU. In comparison to sub-procedures written for CPU, in the case of kernels the main issue to which one should pay attention is the memory access rather than the complexity of the algorithm. The size of the global memory of the graphics card allocated by the program depends on the size of the spatial grid used in the simulations. It typically uses between several tens of megabytes and several gigabytes. Unfortunately an access to this kind of memory is very slow (it takes 400–800 clock cycles per one operation of memory access) [2]. It is the bottleneck in the CUDA calculations. Thus it is so important to optimize all operations of reading from and writing to this memory. One way to optimize the usage of the global memory is to use the coalesced access, in which whole data sets are transferred in one transaction. We use the scenario of the coalesced memory access in which data are copied to the much faster shared memory. Therefore it contains a copy of the data, which allows us to avoid multiple copying of the same data from the global memory. Next, the calculations are performed and finally the data are copied back to the global memory of the graphics card. However, using the shared memory may cause so-called shared memory bank conflicts, which occur if one tries to simultaneously access the data from two or more memory addresses located in the same part of the shared memory (this parts are called banks). In such a case those memory operations have to be performed sequentially instead of the usual parallel memory access which takes place if the memory addresses belong to different banks. As recommended in [2] (especially the Programming Guide) the simplest and simultaneously the most effective way to avoid bank conflicts is to add an additional column to the matrix located in the shared memory:

```c
__shared__ cufftComplex matrix_name[BLOCK_DIM][BLOCK_DIM+1];
```

In the above declaration the BLOCK_DIM constant, defined in the header file user_definitions.h, keeps the size of the thread block. Thus the number of threads in a block is BLOCK_DIM x BLOCK_DIM. The number of thread blocks is determined at runtime taking into account the spatial grid size and the BLOCK_DIM constant, namely:

```c
dim3 grid(xdim / BLOCK_DIM, ydim / BLOCK_DIM, 1);
```

The xdim and ydim variables store the size of the spatial grid. The speed of the calculations may depend on the value of the BLOCK_DIM constant. Thus it is important for the user to carry out some efficiency tests in order to determine which value of this constant leads to the best performance on a particular user graphics card’s model.

The declarations of the five kernels responsible for calculating the Chebyshev polynomials are assembled in the file 2d-chebyshev_gpu.h. Their definitions may be found in the file named 2d-chebyshev_kernel.cu. The first one, in the order of execution, is the kernel cuMomem-

tumWavefunctionProducts, which multiplies the matrix containing the wavefunction in the momentum representation (i.e. after the forward FFT transformation is performed) by one of the momentum components or by the momentum square. Thus the output of this kernel are three matrices. This kernel's CPU counterpart is the for loop located in the method chebyshev::KineticOperator (see the description above). The second one, named cuKineticOperator, using the previously described kernel, calculates and sums up the second terms of the Hamiltonian acting on the wavefunction, which contain the first- and second-order derivates (in the momentum representation – a multiplication by momentum components and its square). In other words it calculates the expression given by Eq. (4). In the CPU version this is done by the whole member function chebyshev::KineticOperator. Of course in the GPU version the FFT transform is also performed using the CUDA cores. The next kernel cuHamiltonian adds the results of the above operation to the potential part of the Hamiltonian multiplied by the wavefunction, i.e. \( S_0(x,y,t)\psi(x,y,t) \). Therefore the result of the whole Hamiltonian acting on the wavefunction is calculated, which corresponds to the result of the chebyshev::Hamiltonian method in the CPU version.

The goal of the next step is to determine the first and the second Chebyshev polynomials, which is done by kernel cuChebyshevInit. All the rest of the necessary polynomials are recursively calculated by the kernel cuChebyshev.

Two additional kernels, designed for matrix operations, are placed in the file 2d-matrix_kernel.cu. They are the kernel cuScalarMatrixProduct, which multiplies a matrix by a scalar, and cuSchurProduct, which calculates the Schur product of two matrices. In both cases, as well as in the previously described kernels, the elements of the matrices are assumed to be complex numbers of a single or double precision depending on the macro constant DOUBLE_PRECISION_USAGE defined in user_definitions.h file. This file contains also the definitions of several other precompiler macros which determine the way the simulation is performed (see Table 3).

As mentioned above, for calculating the FFT on GPU we use the functions delivered by NVIDIA (they are included in the NVIDIA GPU Computing Toolkit). However, in order to unify their usage for a single and double precision we wrap them by macro __CUFFTEXEC defined in default.h, which selects the proper function based on the value of the above-mentioned constant DOUBLE_PRECISION_USAGE.
## 5. The supporting classes

Two additional classes are included in the presented set of classes. One of them is the `wavefunction` class (see Fig. 1), which is responsible for storing the wavefunction in the memory, saving it to the files, calculating the overlap with the initial state’s wavefunction, the projections onto the wavefunctions stored in the memory (which can be performed using the GPU) and so on. The member fields controlling this class are described in Table 4. The parameters determining the path of the file containing the initial wavefunction, the frequency of calculating the observables’ mean values, of writing them down to the output file and of writing the wavefunction snapshots as well as a list of files with the wavefunctions to project on are read from the file by a private method `read_parameters` (for the structure of the parameters file see Table 5).

In the `wavefunction` class we also define an auxiliary member function `step_forward`. Its main task is calling the `step_forward` method defined in `chebyshev_solver` class (the `propagator` base class pointer is used which enables one to easily replace this solver by another one), but it is also responsible for increasing the time step counter and for calculating and writing the output data. So it is convenient to use it instead of the original method of `chebyshev_solver` class to proceed with the simulation.

In the `wavefunction` class we use the reduction algorithm for parallelly summing up the array elements implemented by NVIDIA in the kernels aggregated in the file `reduction_kernel.cu`. However, we have modified it in order to adapt it to our project (e.g., using the complex type).

We have not yet mentioned two additional classes: `grid_parameters` and `complex_array` (see Fig. 1). The class `grid_parameters` is a simple and compact way to store the parameters of the spatial and time grids used in the calculations (see Table 6 for members of this class). More interesting is the `complex_array` class template which implements the array of complex number. In fact it is the wrapper for the array of floating point numbers arranged as a sequence of real and imaginary parts of the complex numbers. The way it is arranged is important – the same structure is assumed by CUDA functions of complex arguments. Moreover, a verification of the accessed element’s index is implemented to avoid the error of reading or writing the elements outside the range of the array. To mimic the usage of the common arrays we defined the access operator [] which works for complex elements. From the CUDA usage point of view the most important advantage of this class is the possibility of an access to the homogeneous arrays of the floating point numbers (that is filling the continuous part of the memory). This makes copying arrays to the graphics card’s memory easier and more effective with no need to use any intermediate temporary structures.

### Table 3
The set of precompiler macro constants determining the way the code works defined in `user_definition.h` header file.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_USAGE</td>
<td>Should the CUDA technology be used?</td>
</tr>
<tr>
<td>BLOCK_DIM</td>
<td>The size of the GPU thread block.</td>
</tr>
<tr>
<td>BJK_MIN</td>
<td>The “numerical zero” for determining the number of Chebyshev polynomials.</td>
</tr>
<tr>
<td>DEVICE_ID</td>
<td>Zero-based identifier of the device to be used in the calculations.</td>
</tr>
<tr>
<td>DOUBLE_PRECISION_USAGE</td>
<td>If defined, the double precision floating point numbers will be used in the computations.</td>
</tr>
<tr>
<td>TIME_MEASUREMENT</td>
<td>If defined, the duration of solver’s successive operations are written to the file.</td>
</tr>
</tbody>
</table>

### Table 4
Public member functions of the `wavefunction` class.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wavefunction (grid_parameters* _gp,</td>
<td></td>
</tr>
<tr>
<td>equation_base* _eq, propagator* _solver,</td>
<td></td>
</tr>
<tr>
<td>char _filename[FNDLG],</td>
<td></td>
</tr>
<tr>
<td>bool _user_command=false);</td>
<td></td>
</tr>
<tr>
<td>~wavefunction();</td>
<td></td>
</tr>
<tr>
<td>int step_forward();</td>
<td>The constructor of the class (see listing 1 for example of its use). In the case of an error while reading the initial wavefunction from the file the runtime_error exception is thrown.</td>
</tr>
<tr>
<td>The destructor of the class.</td>
<td></td>
</tr>
<tr>
<td>The function which should be called in order to perform one step of the evolution (see listing 1 for the example its use).</td>
<td></td>
</tr>
</tbody>
</table>

### Table 5
The sequence of the parameters which should be placed in the file read by `wavefunction` class.

<table>
<thead>
<tr>
<th>The name of the field of <code>wavefunction</code> class storing the value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wf_filename</td>
<td>The path of the file containing the initial wavefunction.</td>
</tr>
<tr>
<td>timed_saving_interval</td>
<td>The number of time steps between calculating and writing the norm, the correlation with the initial state and the expectation values of position.</td>
</tr>
<tr>
<td>plot_saving_interval</td>
<td>The number of time steps between writing the file containing the modulus square of wavefunction (for plotting).</td>
</tr>
<tr>
<td>out_saving_interval</td>
<td>The number of time steps between writing the raw data of the wavefunction.</td>
</tr>
<tr>
<td>projection_input_filenames_count</td>
<td>The number of files containing the wavefunctions to project on (in order to calculate the occupation of states).</td>
</tr>
<tr>
<td>projection_input_filenames[]</td>
<td>The set of paths to above files.</td>
</tr>
</tbody>
</table>
Table 6
Public data members of the grid_parameters class.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
<th>Is read from file (position in the file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>__OTYPE xdim;</td>
<td>The number of nodes in x direction</td>
<td>Read from file (1)</td>
</tr>
<tr>
<td>__FTYPE xmin;</td>
<td>The minimum of x range</td>
<td>Read from file (2)</td>
</tr>
<tr>
<td>__FTYPE xmax;</td>
<td>The maximum of x range</td>
<td>Read from file (3)</td>
</tr>
<tr>
<td>__OTYPE dx;</td>
<td>The spatial step in x direction</td>
<td>Calculated</td>
</tr>
<tr>
<td>__OTYPE ydim;</td>
<td>The number of nodes in y direction</td>
<td>Read from file (4)</td>
</tr>
<tr>
<td>__FTYPE ymin;</td>
<td>The minimum of y range</td>
<td>Read from file (5)</td>
</tr>
<tr>
<td>__FTYPE ymax;</td>
<td>The maximum of y range</td>
<td>Read from file (6)</td>
</tr>
<tr>
<td>__OTYPE dy;</td>
<td>The spatial step in y direction</td>
<td>Calculated</td>
</tr>
<tr>
<td>__OTYPE tdim;</td>
<td>The number of time steps in the simulation</td>
<td>Read from file (7)</td>
</tr>
<tr>
<td>__FTYPE tmin;</td>
<td>The starting time</td>
<td>Read from file (8)</td>
</tr>
<tr>
<td>__FTYPE tmax;</td>
<td>The final time</td>
<td>Read from file (9)</td>
</tr>
</tbody>
</table>

The public member functions

grid_parameters(char *filename);

The constructor of the class (see listing 1 for example of its use) It raises the exception runtime_error if it is not possible to read the parameters file.

__FTYPE x
(_OTYPE nx) const;

Returns the value of the x coordinate for the node of the given index

__FTYPE y
(_OTYPE ny) const;

Returns the value of the y coordinate for the node of the given index

__FTYPE t
(_OTYPE nt) const;

Returns the value of time for the given time step

OTYPE nx
(_OTYPE arg_x) const;

Determines the index of the node for the given value of the x coordinate

OTYPE ny
(_OTYPE arg_y) const;

Determines the index of the node for the given value of the y coordinate

For storing the complex numbers we use our own template class complex. However, the class complex_array, as well as the rest of the code may also be adapted for the complex template defined in the Standard Template Library of C++. For the sake of convenience we defined the type fcomplex which is the complex template instantiation for float or double types depending on the DOUBLE_PRECISION_USAGE constant. Moreover, we defined the descendant classes: complex_array_1d, complex_array_2d and complex_array_3d to simplify storing one-, two- and three-dimensional arrays.

All the code includes comments in the doxygen format [14]. The documentation in HTML format created on the base of these comments is added to the package.

6. An example

The file 2d-qdyn_cuda_pA.cu contains the main function which is an example of using our set of the classes (listing 1). It starts with creating two objects – the instances of the grid_parameters and of the equation_pA classes. The latter is an example of the descendant class which inherits from the class equation_base. It is defined in the file 2d-equation_pA.h. It contains the overridden functions T, Six, Siy, V and S0 in the way to obtain the Schrödinger equation with the Hamiltonian in the velocity gauge, describing an atomic system interacting with a laser electric field, namely:

\[
\frac{\partial}{\partial t} \Psi(x, y, t) = -\frac{1}{2} \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \Psi(x, y, t) - iA_x(x, y, t) \frac{\partial}{\partial x} \Psi(x, y, t) - iA_y(x, y, t) \frac{\partial}{\partial y} \Psi(x, y, t) + \left( V(x, y, t) + U_{abs}(x, y) + \frac{A_x^2 + A_y^2}{2} \right) \Psi(x, y, t),
\]

(5)

where S2, S1x, S1y, S0 functions are explicitly given (the atomic units, abbreviated as a.u., were used, i.e. \( e = m = \hbar = 1 \)). The vector potential \( \vec{A}(x, y, t) \) corresponds to the plane wave in the dipole approximation given by the laser electric field, i.e. \( \vec{E}(t) = \vec{E}_e \cos(\omega t) + \vec{E}_y \cos(\omega t) \). The parameters of the Hamiltonian, including the electric field amplitudes and frequencies, are determined by the parameters read from the file (see Table 2 for its structure).

The form of Hamiltonian presented in Eq. (5) is common for numerical investigations in quantum optics and attophysics. In particular, we use it for strong field atom-laser interaction simulations [5,15].

The member function V describing the binding potential of equation_pA class contains the switch instruction which allows one, depending on the value of potential_type field (see Table 2), to choose one of the predefined potentials: no binding potential, a radial potential well or a Coulomb potential with a smoothed core.

The pointers to the two above mentioned instances of the classes grid_parameters and equation_pA, are passed as the arguments to the constructor of the chebyshev_solver class. In turn the instance of this class is used by the wavefunction class. Having the object of the latter type, the user should call the wavefunction::step_forward function as many times as necessary. It should be limited by the number of the time steps (see Table 6), but can be additionally dependent on the values obtained during the
simulation, e.g., the norm of the wavefunction. After the main loop has been finished one should remember to release the memory of all four objects.

Listing 1. The function `main` defined in `qndyn_cuda_pA.cu` file – the recipe of how to use the presented set of classes. The time measurement instructions were removed for clearness.

```c
int main(int argc, char* argv[]) {
    #ifdef CUDA_USAGE
    if(!InitCUDA()) return 0;
    #endif
    grid_parameters *gp = NULL;
equation_pA *eq = NULL;
    chebyshev_solver *cs = NULL;
    wavefunction* wf = NULL;
    try
    {
        gp=new grid_parameters(gp_par_filename);
eq=new equation_pA(eq_par_filename,gp);
        cs=new chebyshev_solver(gp,eq,CHEBYSHEV_NO);
        wf=new wavefunction(gp,eq,cs,init_wf_filename,FILE_COMPRES);
        for(__OTYPE index=1; index<=gp->tdim; index++)
        {
            wf->step_forward();
        }
        cout << "OK. " << "\n"
    } catch (runtime_error &error) {
        cerr << "Error: " << error.what( ) << endl;
    }
    if (wf != NULL) { delete wf; wf = NULL; }
    if (cs != NULL) { delete cs; cs = NULL; }
    if (eq != NULL) { delete eq; eq = NULL; }
    if (gp != NULL) { delete gp; gp = NULL; }
    return 0;
}
```

To compile all classes one needs to use an NVIDIA's compiler `nvcc`. We have tested it for Windows and Linux operating systems. In both cases we use the compiler delivered with NVIDIA GPU Computing Toolkit is 3.2. To make the compilation easier in the case of Windows we additionally prepared the Microsoft Visual C++ 2008 project. However, it requires the CUDA Wizard add-on (available at [16]). Alternatively the `makefile_cpu` and `makefile_gpu` files have been prepared, which allow one to compile the source code of the example program using directly `nvcc` for Linux or Windows.

7. Performance

Using the above class `equation_pA` we have tested the performance of the presented solver for several graphics cards. Testing it we have focused on the part of the code we deliver, that is omitting the part which should be implemented by the user, who needs to write the S functions on his own. Depending on the user's choice the latter may be implemented for CPU, as in the above example, or for GPU, which requires an experience in CUDA but will be much faster and no requires the data copying between host's and graphics card's memories. The rest of our solver may work on GPU.

As described above our solver bases on repeatedly executed FFT. For GPU we use its implementation included in the NVIDIA GPU Computing Toolkit. Thus in the best case, our code running on GPU should scale as this implementation of FFT does. Strictly speaking FFT is executed four times for every Chebyshev polynomial used in the expansion of propagator, the number of which can be adjusted in the runtime. In practice this means the FFT is executed about 50–100 times in one time step. Of course performing the forward and inverse FFTs does not take the whole time used in one time step. An additional time is consumed by a transfer of the data from host's memory to graphics card's memory and back, as well as the calculation performed in order to obtain the coefficients of the Chebyshev expansion and other necessary calculations. In order to learn what are the durations of the successive stages of one time step let us trace it in the case of the graphics card Quadro 4000 installed in the host equipped with the CPU Intel Xeon W3680 3.33 GHz and 8 GB of
RAM. We intentionally use only one CPU core to simplify and make it less ambiguously the comparison of GPU and CPU performance and the calculation of the speed-up. The durations presented below are measured for the grid with 1024 × 1024 nodes and a single precision of floating point numbers.

The duration of one step performed by our solver, which means only one execution of the chebyshev_solver::step_forward method without calculating the projections of the wavefunction and other time-dependent quantities and without writing the wavefunction into the file, is about 12 500 ms for a single thread of above CPU. Exactly the same calculations performed using GPU takes only 81 ms, which means that the latter are more than 150 times faster. The above time was calculated as average over 1000 time steps for GPU and 100 time steps for CPU, respectively. To measure the duration of stages of the time step we use the QueryPerformanceCounter function from Windows API using the GPU’s threads synchronization [17]. We have checked that the time taken by measuring the time (so called overhead) is less than 1 μs. This method allows us to measure the times with the resolution of the CPU cycles frequency.

In order to obtain an overall time step duration one should also add to the above the time of calculating the S functions (it is about 197 ms in the above example implementation using CPU). In the case of using CUDA, one should also add a duration of sending the data to graphics card memory, which takes about 8.5 ms. We send to the graphics card three tables containing the S functions, the size of which is in our test case 1024 × 1024 elements. As mentioned above, we also take care of the so-called bank conflicts (see [2,18,17]). In order to avoid them we have included an additional column in two-dimensional tables used to store the S functions, the initial wavefunction and all other tables copied from the host computer's memory.

In the case of a step in which the wavefunction is written down to the file or time-dependent values are calculated, the current wavefunction has to be sent from the graphics card back to the host's memory. This takes one third of the above time consumed by sending the data to the graphics card because only one table of size 1024 × 1024 has to be copied. The times of data transfers strongly depend on the graphics card interface. In the described case it is PCIe x16.

A single execution of the forward out-of-place FFT of the wavefunction on GPU for the above parameters takes about 1 ms (precisely 922 μs). Then the wavefunction in the momentum coordinates is multiplied by the coordinates of the momentum vector and its modulus square and then simultaneously copied into three tables, which are then transformed to the space coordinates using the backward FFT. The former takes about 0.5 ms and the latter takes about 2.5 ms. The whole duration of the calculations of the Hamiltonian is then equal to about 5 ms in which FFT, performed four times, takes 3.5 ms (about 70%). This duration should be multiplied by the number of the Chebyshev polynomials, fifteen in this case, and added to the duration of performing the phase shift (which is in fact negligible, because it takes only 0.5 ms per step) to obtain the overall time step duration 81 ms. In the whole step duration, 57 ms is used by performing the FFTs. In some rare cases the time step duration was much longer than the average values presented above. We suspect it was caused by engaging the graphics card to other tasks than our calculations, i.e. the new versions of Windows operating system uses the graphics card to display its GUI. Note that we do not use the graphics cards dedicated to the calculations only, although we care not to run any other programs at the same time we execute the program performing the tests.

Note that the above durations are not only the times of performing the floating point operations. Time used by a transfer of data from the global memory of the graphics card to the shared memory is significant.

For a double precision of floating point numbers the durations of every part of the time step executed by GPU lengthen (the described graphics card is built in the Fermi technology), but the ratio of the durations of the successive stages is conserved. The total duration of a time step in double precision is 201 ms, while for CPU it is 8500 ms. Note that while using the double precision slows down the calculation on GPU, it speeds up, at least in our case, the calculation performed on CPU (we do not use the streaming SIMD extension (SSE) of CPU instructions). For a discussion of advances of using the single and double precision floating-point numbers see [19] and references therein. In particular, it presents an algorithm of numerical calculations using the mixed precision approach.

It should be stressed that the results obtained using CPU and GPU are in a fairly good agreement. After one tenth of the optical cycle, the differences between the values of the norm and of the square modulus of the correlation with the initial state obtained using CPU and GPU are both equal to 10\(^{-6}\) for single precision floating points. For double precision those differences are of order of 10\(^{-14}\). Thus in both cases the values are close to the numeric precision.

We have tested the scalability of our solver by changing the size of the input data, i.e. the number of the nodes of the grid. Fig. 2 shows the results of those tests, namely an average duration of one step of calculations performed only by GPU: the left column contains the plots for the grid scaled in both directions (case A), while in the right column there are plots for the grid in which the size is changed only in one direction (case B). The upper plots are prepared for single precision, while lower ones – for double one. One can see that the time of the GPU calculations, again taken without the duration of calculating the S functions and its transfer from the host's to graphics card's memory, scales quadric with the scaling factor n (see the description under Fig. 2 for its definition) in case A and linearly in case B. This means that in both cases it scales linearly with the number of the grid nodes (cf. the Gustafson's law for number of processor used).

The total duration of the FFT scales in the same way (not shown).

It is worth to note, that the calculations performed on CPUs scale in a similar way (not shown), that is linearly with the number of grid nodes in both cases of single and double precision, however the CPU again proves to work better for double precision. Of course the inclinations of the plots vary much for various hardware configurations of the host computers. Also the relative duration of a time step performed by CPU for single and double precision differs significantly for various configurations (in the extreme case the duration difference was equal to 30% while usually it is less than 5–10%). In the case of GPU, comparing the efficiency of calculations for single and double precision (Fig. 3) one can draw a simple conclusion: for graphics cards from Fermi family the duration of a time step for double precision is twice longer than for single precision floating point numbers, while for older graphics card this ratio is larger than three.

As mentioned, we focus on measuring the performance of the parallel code executed by CPU, from which the time taken by calculating the S functions and the data transfer is excluded. In this way we make the measurements independent of a large set of parameters of the host computer, which influence the time of calculations but do not depend on the code optimization we can make. A good example is the graphics card's interface or driver's version.

Such an approach, in which we focus only on the parallel part of the code, made it easier to check whether our code scales with the number of cores according to the Amdahl's law for the expected speed-up, namely
Fig. 2. The time of the GPU calculations for the grids the size of which changes by the factor of 2 in one or both directions (respectively, left and right column), i.e. the grid’s size is equal to $128n \times 128n$ or $128 \times 128n$, where the scaling factor $n$ takes values of 1, 2, 4 and 8. The upper plots are prepared for single precision, while the lower ones for double precision.

Fig. 3. A comparison of the time step duration for various graphics cards and for single versus double precision. In the brackets the number of CUDA cores is specified. The graphics cards built in Fermi technology are marked with letter “F”.

\[
\frac{1}{(1 - P) + \frac{N}{P}},
\]

where $P$ is the part of the code, which is parallelized and $N$ is the number of processor cores. The latter value can be found in Table 7. Since we measure only the duration of the parallel code’s execution, the value of $P$ is equal to unity. Therefore the Amdahl’s law simplifies to $1/N$, which describes the way in which the duration should shorten with a growing number of the CUDA cores. Note however, that since one cannot control the number of CUDA cores used for calculations, the value of $N$ may be changed only by physically changing the whole graphics card. However, the condition, under which the comparison of two cards with different numbers of CUDA cores is reasonable, is that those two graphics card belong to the same family, in which the GPU architecture is similar and thus single GPU cores have a comparable efficiency. As visible in Table 7 this condition is rather difficult to fulfill outside the specialized testing labs. Thus we decided to use the extended Amdahl’s law in order to take into account both the number of cores and its efficiency measured simply by its nominal frequency multiplied by the number of instructions which may be executed simultaneously. The latter parameter, different for single and double precision, is unfortunately usually not listed in the graphics cards specifications. The product of these three parameters is called the theoretical efficiency of the whole GPU. It is expressed in GFLOP units. In the best case the speed-up of our code (calculated comparing the durations obtained for particular GPU with the duration for the Intel Xeon W3680 3.33 GHz processor and 8 GB RAM) should scale as the theoretical efficiency. Its values for several tested graphics cards can be also found in Table 7. We have checked that such a computed theoretical efficiency agrees well with the efficiency measured in benchmarks realized for single precision and widely available in the Internet.
The speed-up of the parallel part of the code for various graphics cards scales very well with the theoretical efficiency of the GPU (see Fig. 4). Note that the speed-up is in fact expressed in arbitrary units since it depends on an arbitrary selected CPU – this is however a weakness of any attempts to the speed-up calculations. Only the ratios of two speed-ups calculated for various graphics cards do not depend on the parameters of the chosen CPU. In particular, the CPU used by us as the reference is a very powerful one and thus in fact it lowers our speed-up values.

In the case of the best graphics card in the tested set, that is GeForce GTX 580, the speed-up is equal to about 350 if the duration of the GPU calculations is compared with the duration of the calculations performed by a single thread of the Intel Xeon W3680 3.33 GHz processor. As mentioned, we intentionally do not parallelize the code performed by CPU. However, in order to get more a reliable image of benefits of using the CUDA technology, one should take into account such a possibility. In an extreme, but not realistic case, one should divide the speed-up visible in Table 7 by the number of cores of CPU. The number of cores in the above processor is equal to 6, which gives the value of such a “renormalized” speed-up equal to about 60. It is still a great value, especially taking into account the economic factor. However, such a speed-up of CPU parallel calculations is purely theoretical. In practice its value is much lower. Thus the benefits of using CUDA and massively parallel processors of a graphics card are still greater that those of the newest powerful CPUs.

Acknowledgements

This work was partially supported by the Nicolaus Copernicus University (grant. No. UMK 386-F/2010) and by the European Social Fund and the budget of Poland within the Integrated Regional Operational Programme, measure 2.6 “Regional Innovation Strategies and transfer of knowledge” under the project of Kujawsko–Pomorskie voivodeship “Scholarships for Ph.D. students 2008/2009 – IROP”. We would like also to thank Ł. Pepłowski and M. Sylwestrzak for making their graphics cards available to our tests.
Table 8
The structure of the output file containing the durations of the subroutines of the solver.

<table>
<thead>
<tr>
<th>The name in GPU measurements</th>
<th>The description for GPU code</th>
<th>The name in CPU measurements (the code corresponding to that from the left column)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer1 (S functions)</td>
<td>The duration of the S functions computation</td>
<td>Timer1 (S functions)</td>
</tr>
<tr>
<td>Timer2 (H2D)</td>
<td>The time taken by the transfer of the tables containing the computed values of S functions from the host’s memory to the graphics card’s global memory</td>
<td>N/A</td>
</tr>
<tr>
<td>Timer3 (1xFFTfor)</td>
<td>The duration of forward FFT execution</td>
<td>Timer2 (1xFFTfor)</td>
</tr>
<tr>
<td>Timer4 (cpuMWP)</td>
<td>The duration of execution of kernel cuMomentumWavefunctionProducts</td>
<td>Timer3 (cpuMWP)</td>
</tr>
<tr>
<td>Timer5 (3xFFTinv)</td>
<td>The duration of backward FFT execution</td>
<td>Timer4 (3xFFTinv)</td>
</tr>
<tr>
<td>Timer6 (cpuKO)</td>
<td>The duration of execution of kernel cuKineticOperator</td>
<td>Timer5 (cpuKO)</td>
</tr>
<tr>
<td>Timer7 (Ham)</td>
<td>The duration of execution of kernel cuHamiltonian</td>
<td>Timer6 (cpuHam)</td>
</tr>
<tr>
<td>Timer8 (cuCh)</td>
<td>The duration of execution of kernel cuChebyshevInit</td>
<td>Timer7 (cpuChInit)</td>
</tr>
<tr>
<td>Timer9 (cuCh)</td>
<td>The duration of execution of kernel cuChebyshev</td>
<td>Timer8 (cpuCh)</td>
</tr>
<tr>
<td>Timer10 (cuSMP)</td>
<td>The duration of execution of kernel cuScalarMatrixProduct</td>
<td>Timer9 (cpuSMP)</td>
</tr>
<tr>
<td>Timer11 (ChM)</td>
<td>The time taken by execution of Chebyshev method step and of FFTs (duration of method chebyshev_solver::step_forward without duration of data transfer in both ways and of computation of S functions)</td>
<td>Timer10 (ChM)</td>
</tr>
<tr>
<td>Timer12 (D2H)</td>
<td>The time taken by the transfer of the new calculated wave function from the graphics card’s global memory to the host’s RAM memory</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Appendix A. Test runs

In order to enable the tests of the presented classes and CUDA kernels we have prepared several sets of input files, namely the parameter files together with the initial wavefunction files, aggregated in the folder test_runs. All of them use the above-described example of the descendant class equation_pA, i.e. an implementation of the Hamiltonian describing an atom interacting with a laser field in the velocity gauge.

In all cases we use the spatial grid of size 1024 × 1024. The time step is equal to Δt = π/500 and the number of time steps is 1000. In subfolders reference_output one can find the reference output of these simulation, namely the file 2d-timed.dat containing the calculated mean values of the observables x and y, the overlap with the initial state as well as the norm of the wavefunction, additionally in the file 2d-timed.dat file.

In subfolders reference_output one can find the reference output of these simulation, namely the file 2d-timed.dat containing the calculated mean values of the observables x and y, the overlap with the initial state as well as the norm of the wavefunction, additionally in the file 2d-timed.dat file.

Table 8 contains the description of each column written in this file. The last row contains the averaged values calculated for all the above rows. The time measurements are switched off by default.

For our test runs performed for single precision floating point numbers we used the computer equipped with processor Intel Core Quad Q6600 2.40 GHz, 8 GB of RAM memory and 64-bit operating system Microsoft Windows 7. The graphics card is GeForce GTX 480 with 480 CUDA cores and 1536 MB GDDR5 of main memory. The version of the graphics card drivers is 267.24. The precise version of NVIDIA GPU Computing Toolkit is 3.2.16.

To assure the full portability we have compiled the above code using switches -arch sm_10 and -arch sm_13 for double precision.

A.1. Test 1 (free electron)

For the first test we use no potential or external field. The initial wavepacket is a two-dimensional Gaussian packet of width equal to 1 a.u., located at the origin of the coordinate system. While the simulation begins, the wavepacket starts to spread out. After a few cycles the absorbing potential acts, which leads to a decrease of the norm of the wavepacket.

Subfolder 1_free_electron.

A.2. Test 2 (Volkov state)

For the initial parameters identical as in previous case we add a laser field along the x axis \( \epsilon(t) = \epsilon_0 \cos(\omega t) \). The laser frequency is \( \omega = 1 \) a.u. and the laser field amplitude is \( \epsilon_0 = 1 \) a.u. The laser field causes the wavepacket, apart its spreading, to perform oscillations. The expectation value of the position’s x component oscillates in the range of between 0 and \( -2 \) a.u. according to formula \( \langle x(t) \rangle = \cos(\omega t) - 1 \) (compare the fourth column of the 2d-timed.dat file). The wavepacket oscillation’s manifestation is the comb-like structure of the initial state occupation (the third column of the 2d-timed.dat file).

Subfolder 2_Volkov_state.
A.3. Test 3 (photodetachment in the presence of a strong laser field)

In the last of the presented tests we simulate a two-dimensional ion-laser system in the photodetachment scenario. The binding potential is a radial potential well, the width of which is $a = 1$ a.u. and the depth $V_0 = -2$ a.u. The initial state is the ground (and only) state of this potential. This system is disturbed by a laser field described $\mathbf{E}(t) = \mathbf{\hat{x}} \varepsilon_0 \cos(\omega t + \pi)$, where $\omega = 1$ a.u. and $\varepsilon_0 = 1$ a.u., which belongs to the regime of ultra-strong field attosecond ionization. Thus the ionization occurs quickly and the test does not need to last very long.

Subfolder 3_photodetachment.

References

[1] See webpage http://www.nvidia.com/object/cuda_apps_flash_new.html for examples of implementations using GPGPU (it includes the speed-up factors). Moreover, there are some attempts to simulate soft-body systems in real time, which is possible only due to GPGPU.


