Computer Physics Communications 185 (2014) 1791–1807

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

A GPGPU based program to solve the TDSE in intense laser fields through the finite difference approach☆

Cathal Ó Broin*, L.A.A. Nikolopoulos

School of Physical Sciences, Dublin City University and National Centre for Plasma Science and Technology, Ireland

ARTICLE INFO

Article history:
Received 9 August 2013
Received in revised form
11 February 2014
Accepted 15 February 2014
Available online 25 February 2014

Keywords:
GPGPU
TDSE
OpenCL
Parallel
Finite difference
Taylor
Runge–Kutta
Lanczos

ABSTRACT

We present a *General-purpose computing on graphics processing units* (GPGPU) based computational program and framework for the electronic dynamics of atomic systems under intense laser fields. We present our results using the case of hydrogen, however the code is trivially extensible to tackle problems within the single-active electron (SAE) approximation. Building on our previous work, we introduce the first available GPGPU based implementation of the Taylor, Runge–Kutta and Lanczos based methods created with strong field *ab-initio* simulations specifically in mind; CLTDSE. The code makes use of finite difference methods and the OpenCL framework for GPU acceleration. The specific example system used is the classic test system; Hydrogen. After introducing the standard theory, and specific quantities which are calculated, the code, including installation and usage, is discussed in-depth. This is followed by some examples and a short benchmark between an 8 hardware thread (i.e. logical core) Intel Xeon CPU and an AMD 6970 GPU, where the parallel algorithm runs 10 times faster on the GPU than the CPU.

Program summary

Program title: CLTDSE

Catalogue identifier: AESM_v1_0

Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AESM_v1_0.html

Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland

Licensing provisions: GNU General Public License, version 3 and above

No. of lines in distributed program, including test data, etc.: 15 988

No. of bytes in distributed program, including test data, etc.: 233 518

Distribution format: tar.gz

Programming language: C99 and OpenCL C. C99 conformance is ensured through use of C99 and pedantic flags under GCC and Clang.

Computer: Single compute node.

Operating system: GNU/Linux. It should, in principle, work with little modification for other Unix-Like systems.

Has the code been vectorised or parallelised?: OpenCL is a parallel language. Thus CLTDSE can use all cores on a processor or GPU. OpenCL supports using all available compute units (GPUs and CPUs etc.), although this is not currently implemented in CLTDSE.

RAM: Negligible RAM and GPU global memory (20MiB)

Classification: 2.5.

External routines: An OpenCL library; the current major packages are APP by AMD (CPU and AMD GPU), the NVIDIA driver (GPU), and the Intel SDK for OpenCL Applications (CPU and Intel HD Graphics). libconfig for processing configuration files.

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
E-mail address: cathal.obroin4@mail.dcu.ie (C. Ó Broin).

http://dx.doi.org/10.1016/j.cpc.2014.02.019

Nature of problem:
Describing the dynamics of electrons under intense laser fields in atoms or molecules.

Solution method:
Solving the discretised system through finite difference using the Lanczos, Runge–Kutta, and Taylor methods.

Restrictions:
The example problem which is implemented is under the single active electron approximation.

Unusual features:
Focused on GPGPU acceleration through OpenCL.

Running time:
The running time depends on the nature of the pulse. This can vary from seconds to tens of minutes.

1. Introduction

Currently, the exploration of fundamental processes that atomic and molecular systems undergo when under the action of intense laser fields is a major research area. Such processes are studied, through experiment, using infrared wavelengths [1] or at short wavelengths by free electron lasers (FELs) [2]. Modelling, *ab initio*, the time-dependent dynamics of multielectron systems under intense time dependent fields to high accuracy is a computationally demanding task. Nevertheless, many *ab initio* methods, which are outside the perturbation regime, have been developed to solve the TDSE with few electrons, such as [3–17]. Many methods use, incorporate or are derivatives of the finite difference approach.

Computational tractability, combined with few approximations, and accuracy are the most desirable properties when studying multielectron systems, *ab-initio*, under intense ultrashort electromagnetic fields. Many approaches have been developed in atomic and molecular physics, most notably, Time-Dependent Hartree–Fock (TDHF) [18] and Time-Dependent Density Functional Theory (TDDFT) [19].

An alternative approach, with a long and considered history, is the adoption of the single-active-electron (SAE) approximation for atoms [18]. Models based on the SAE approximation, which reduces the dimensionality of the problem by freezing the tightly bound inner electrons, have a proven track record in cases where multiple electron excitations are not important and where single electron effects, such as above threshold ionisation (ATI) and high harmonic generation (HHG), dominate.

In the present work, we present a parallel finite-difference code (CLTDSE) which can run on arbitrary computing devices (CPU, GPU, FPGAs etc.), provided OpenCL support is available, and which works within the SAE context. Although we present results on hydrogen, it is a simple matter to employ an effective single-electron potential and treat the code within the SAE approximation.

CLTDSE is a free software, C99 and OpenCL C based, GPGPU package which implements the action of the time evolution operator on the state of a system described by the Time Dependent Schrödinger Wave Equation (TDSE). The code implements three types of integration methods; embedded and non-embedded Runge–Kutta methods, the arbitrary order Taylor propagator, and Lanczos based propagation. The Lanczos propagator does not explicitly prescribe a specific integration method, but rather it facilitates the integration process by changing the system of equations one needs to integrate to a smaller one. In the current case, the Lanczos method is combined with a Taylor propagator.

In our previous paper [20] we introduced the topic of OpenCL based GPU acceleration of the Time-Dependent Schrödinger Equation (TDSE) within the context of strong field physics. The suitability of performing *ab initio* calculations for GPU acceleration was focused on the basis case where a significant performance improvement was achieved against a standard serial implementation for the case of the Runge–Kutta methods and an arbitrary order Taylor series expansion. To contrast with the previous case, we add the Lanczos method to the methods implemented, and the discussion and code released is centred around finite difference methods. In the context of *ab initio* calculations, this involves very sparse matrices.

GPU parallelisation efforts are not in accelerating the integration method itself; the Runge–Kutta, Lanczos and Taylor methods are serial, and the calculation of each derivative is done sequentially. That is, the methods require multiple derivatives to be calculated, where each new derivative calculation requires information from the previous derivative. Rather than using a parallel integration algorithm, the matrix–vector calculation that is performed at each time step is, itself, parallelised. GPUs are the natural partner to this form of embarrassingly parallel linear algebra acceleration.

QPROP [21] is a C++ library which helps to hide the complexity of split-operator propagation for TDSE and TDDFT systems. That is, it exposes propagator functionality, but the specific application is built by the end user, albeit at greatly reduced complexity. QPROP is under a non-commercial use licence. Non-commercial licences do not meet the open source definition of the Open Source Initiative® or the free software requirements of the Free Software Foundation®. PyProp [22] is a free software, grid based, TDSE propagator written in a mixture of C++, Fortran and Python. It provides a finite difference grid based method, amongst other grid types, as well as providing Krylov subspace based propagation. PyProp is under a copyleft licence, but does not support GPUs directly, and public development appears to have stopped.

The HELIUM code, is a Fortran based code for 2 electron atomic systems using finite difference. In terms of its code, it takes a complex, multi-layered approach, appearing to have 5 relatively independent logical layers, and thus at least 4 separate levels of function wrapping [23]. The HELIUM code does not appear to be freely available. The approach taken in the current case for CLTDSE has been to tie the propagator functionality to OpenCL; being too generic in operation has its own performance penalty. In principle the system could be quite easily adapted to the language CUDA and much of the propagator code was originally written in pure C99 for the basis case. This has been spun off now, as a stand alone separate program.

ALTDSE (Arnoldi–Lanczos TDSE) [11] is another non-commercial licenced package. While it is a basis method, it is noteworthy because it uses a Krylov method; the more general Arnoldi method, written in Fortran 95. 2dhf is a recently updated [24] Fortran 77

electronic structure program. It makes use of a high order finite difference scheme combined with a mixed Self consistent fields (SCF) and coloured successive over-relaxation (SOR, a variant of standard Gauss–Seidel iteration) approach.

QnDynCUDA is a C++ class available under the CPC non-profit use licence agreement, which provides functionality to solve the TDSE using a method which relies on FFT and Chebyshev polynomials [25]. The code has been extended to take advantage of CUDA. CUDA is limited only to Nvidia discrete GPU technology. Thus the future of any CUDA code is dependent on future fortunes of Nvidia, and its future within high performance computing.

In the last decade high performance computational infrastructure has moved away from custom fully integrated systems towards distributed computing models based on highly interconnected commodity machines. At present, these distributed systems are being augmented with various forms of accelerators. Accelerators focus on optimising a specific class of problem. The strength of GPUs is in optimising throughput focused and computationally heavy problems. Currently GPGPU usage is mostly of discrete cards, but with the current ongoing convergence between CPUs and GPUs, it is expected that improved GPGPU performance with double precision will be available on integrated chips. Two benefits of this integration will be the lower latency and higher bandwidth between the GPU and CPU.

GPUs use a Single-Instruction Multiple-Data (SIMD) layout, or very long instruction words (VLIW), or a combination of the two. In a SIMD, a processor takes one instruction with multiple data arguments and executes the instructions on multiple processing elements. A VLIW is similar in operation except that each processing element can be fed a different instruction. The AMD FirePro V7800, discussed in our previous paper, had multiple VLIW units arranged in a SIMD design.

The Arithmetic Logic Unit (ALU) performs basic arithmetic operations. Often ALU without further descriptors refers to integer arithmetic operations specifically. A Floating point unit (FPU) is an ALU that performs arithmetic operations with floating point formats. FPUs typically work with the single and double precision formats from the *IEEE Standard for Floating-Point Arithmetic 754* [26]. The processing elements of a GPU are essentially FPUs. When GPGPU first became available the GPUs were not fully compliant with the floating point specification, but recent models claim compliance.

GPUs operate as a collection of independent SIMD-like units, so they operate well with the OpenCL model. CPUs typically have a SIMD ability, though generally on a much smaller scale than on a GPU. A primary difference of interest is the CPU's focus on data/intra-thread locality while the GPU focuses on inter-thread locality. That is, on a CPU, data for one thread is logically co-located into a contiguous block of memory, but for a GPU the memory is ordered to suit a single memory access by each thread concurrently. With a symmetric (filled) block tridiagonal matrix, for example, it is trivial to experiment with both of these methods to see which has the best performance; one simply has to switch the blocks and change a minor piece of calculation logic.

C, specifically the C99 standard, was chosen over C++ and Fortran. The chief reason for choosing C99 was that OpenCL C is a C99 variant, and the OpenCL specification is primarily defined as a C interface. As a language, C is conceptually simple with a small amount of syntax to learn, yet very powerful. The heavy numerical processing in this case was performed by OpenCL C code; the advantages within Fortran for numerical processing are thus not relevant.

In the following sections we lay out the necessary details for the *ab-initio* calculation of hydrogenic systems under intense linearly polarised laser fields. In Section 2 a description of the underlying theory is given, followed by Section 3 which discusses the form of the electromagnetic fields. The mathematics for calculating the observables and gauge dependent quantities are discussed in Section 4. The remainder of the paper focuses on the specifics of installing (Section 6.1), using (Section 6), analysing (Section 5), and modifying (Section 8) the code. An overview of the settings is provided in Appendix A, and the algorithms for the Lanczos (Appendix B) and Runge–Kutta approaches (Appendix C) are also given to provide further context.

2. Finite-difference formulation of the time-dependent Schrödinger equation

The field-free Hamiltonian of the atomic system $\hat{h}(\mathbf{r})$ reads,

$$\hat{h}(\mathbf{r}) = -\frac{1}{2}\nabla^2 + V(r), \tag{1}$$

where $\mathbf{r}$ is the position vector and the spherically symmetric potential $V(r)$ is assumed to satisfy $rV(r \to \infty) \longrightarrow const$. For example, for pure hydrogenic systems, the potential is given as $V(r) = -Z_{nuc}/r$, where Z_{nuc} is the atomic number.

The Hamiltonian of this system in a linearly-polarised radiation field $\mathbf{E}(t) = \hat{e}\mathcal{E}(t)$, where $\hat{e}$ is the polarisation vector, can be expressed in different forms depending on the chosen gauge used to represent the atom–field interaction operator. In the present implementation we have adopted the length (L) and the velocity (V) gauges in the dipole approximation, expressed as:

$$\hat{d}^{(G)}(\mathbf{r}, t) = \begin{cases} \frac{1}{c}\mathbf{A}(t)\cdot\mathbf{p}, & G = V, \\ \mathbf{E}(t)\cdot\mathbf{r}, & G = L, \end{cases} \tag{2}$$

where $\mathbf{p} = -i\nabla$ represents the momentum operator and $\mathbf{A}(t) = -c\int_{-\infty}^{t} dt'\mathbf{E}(t')$ is the vector potential of the field.

In this case, the Hamiltonian becomes gauge dependent, since $\hat{H}^{(G)}(\mathbf{r}, t) = \hat{h} + \hat{d}^{(G)}(\mathbf{r}, t)$, with the time evolution of the system's wavefunction which is given by,

$$i\frac{\partial}{\partial t}\psi_G(\mathbf{r}, t) = \left[\hat{h} + \hat{d}^{(G)}(\mathbf{r}, t)\right]\psi_G(\mathbf{r}, t). \tag{3}$$

Based on the above, it is concluded that the time-dependent wavefunction also becomes gauge dependent and it is known that the two forms of the wavefunction are related through the gauge transformation:

$$\psi_L(\mathbf{r}, t) = e^{-i\mathbf{A}(t)\cdot\mathbf{r}}\psi_V(\mathbf{r}, t). \tag{4}$$

In our present numerical implementation we choose an orthogonal coordinate system with the z-axis along the polarisation vector of the field; $\hat{z} = \hat{e}$. Moreover, although in principle a partial wave expansion of the wavefunction should include an infinite summation of the quantised angular momentum terms, in an actual calculation we truncate the expansion of the spherical harmonics at some maximum $l = l_{\max}$. The maximum angular momentum $l_{\max}$ should be chosen such the expectation values of the observables under study have sufficiently converged as $l_{\max}$ is increased, while it is also important to note that an unnecessarily large $l_{\max}$ increases the runtime of the simulation. The time-dependent wavefunction $\psi_G(\mathbf{r}, t)$ is expanded on a basis of spherical harmonics $Y_{lm_l}(\theta, \phi)$:

$$\psi_G(\mathbf{r}, t) = \sum_{l,m_l}\frac{1}{r}f^{(G)}_{l,m_l}(r, t)Y_{lm_l}(\theta, \phi), \quad G = L, V \tag{5}$$

where the partial waves $f^{(G)}_{l,m_l}(r, t) = r\langle lm_l|\psi_G\rangle$ are the time dependent quantities to be evolved in time. We proceed by the substitution of Eq. (5) into Eq. (3), followed by a projection onto the

spherical harmonic conjugates $Y^{\star}_{lm_l}(\theta,\phi)$ and integration over the solid angle $d\Omega = \sin\theta d\theta d\phi$. After employing the orthonormalization properties of the spherical harmonics, we arrive at the following propagation scheme for the partial waves [13,16]:

$$i\frac{\partial}{\partial t}f^{(G)}_{l,m_l}(r,t) = \hat{h}_{lm_l}(r)f_{l,m_l}(r,t) + \sum_{l'm_{l'}}\hat{d}^{(G)}_{lm_l,l'm_{l'}}(r,t)f^{(G)}_{l',m_{l'}}(r,t), \quad (6)$$

where the diagonal $\hat{h}_{lm_l}$ and non-diagonal $d^{(G)}_{lm_l;l'm_{l'}}$ radial operators are defined by

$$\begin{aligned}\hat{h}_{lm_l}(r) &= \langle lm_l|\hat{h}(\mathbf{r})|l'm_{l'}\rangle \\ &= \left[-\frac{1}{2}\frac{\partial^2}{\partial r^2} + \frac{l(l+1)}{2r^2} + V(r)\right]\delta_{ll'}\delta_{m_lm'_l}, \end{aligned} \quad (7)$$

$$\hat{d}^{(G)}_{lm_l;l'm_{l'}}(r,t) = \langle lm_l|\hat{d}(\mathbf{r},t)|l'm'_l\rangle = \hat{t}^{(G)}_{ll'}(r,t)k_{l,l'}(m_l) \quad (8)$$

$$k_{l,l'}(m_l) = \langle lm_l|\cos\theta|l'm_{l'}\rangle = \delta_{ll'\pm1}\delta_{m_lm'_l}\sqrt{\frac{l_>^2 - m_l^2}{4l_>^2 - 1}} \quad (9)$$

with $l_> = \max(l,l')$ and $\hat{t}^{(G)}_{ll'}$ the radial coupling operator of each gauge:

$$\hat{t}^{(G)}_{ll'}(r,t) = \begin{cases} -\dfrac{i}{c}A(t)\left[\dfrac{\partial}{\partial r} + (l-l')\dfrac{l_>}{r}\right], & G = V \\ r\mathcal{E}(t), & G = L. \end{cases} \quad (10)$$

As seen from the above expressions, the electromagnetic interaction operator only couples states with equal magnetic quantum number, $m_l = m_{l'}$ and angular momentum numbers that differ by one unit, $l' = l \pm 1$. Therefore the TDSE in the two gauges, length and velocity are finally written as:

$$\begin{aligned} i\frac{\partial}{\partial t}f^{(L)}_{l,m_l}(r,t) &= \hat{h}_{lm_l}(r)f^{(L)}_{l,m_l}(r,t) \\ &+ r\mathcal{E}(t)\left[\kappa_{l,m_l}f^{(L)}_{l-1,m_l}(r,t) + \kappa_{l+1,m_l}f^{(L)}_{l+1,m_l}(r,t)\right] \end{aligned} \quad (11)$$

$$\begin{aligned} i\frac{\partial}{\partial t}f^{(V)}_{l,m_l}(r,t) &= \hat{h}_{lm_l}(r)f^{(V)}_{l,m_l}(r,t) \\ &- i\frac{A(t)}{c}\left[\left(\frac{\partial}{\partial r} + \frac{l}{r}\right)\kappa_{l,m_l}f^{(V)}_{l-1,m_l}(r,t)\right. \\ &+ \left.\left(\frac{\partial}{\partial r} - \frac{l+1}{r}\right)\kappa_{l+1,m_l}f^{(V)}_{l+1,m_l}(r,t)\right], \end{aligned} \quad (12)$$

with $\kappa_{l,m_l} = k_{l,l-1}(m_l)$ and $\kappa_{l+1,m_l} = k_{l,l+1}(m_l)$.

Up to this point the derivation of the time-dependent equations is general and no reference to any particular integration scheme has been made. One choice is the expansion of the partial waves $f^{(G)}_{l,m_l}(r,t)$ on some known basis, which will transform the above partial-differential equation into a coupled system of ordinary differential equations for the expansion coefficients [27]. This approach is known as the basis representation method of the time dependent wavefunction. In the present case we have chosen the grid representation of the wavefunction where the quantities that are treated numerically are the values of the partial wavefunctions $f^{(G)}_{l,m_l}(r,t)$ on a specific radial grid. Considering this grid based approach, the differential operators, which appear in the TDSE, are expressed through a finite difference scheme. In general there are two different ways to conceptualise a grid based problem: in terms of the individual elements communicating between neighbours, or in terms of a linear algebra problem where the derivative is represented by a matrix–vector calculation and where the primary calculation is that of an exponential operator acting on a vector. In the present formulation the latter approach is used for the time evolution of the system. An evenly spaced grid is chosen from the origin up to a distance R, which represents the spatial region where the system is located. Denoting the grid as $r_j = (j-1)h, j = 1,\ldots,N$, where $h = R/(N-1)$ is the constant grid spacing, and dropping the indexes l, m_l temporarily, then the partial wavefunctions and their spatial derivatives are expressed as

$$f_j(t) = f^{(G)}_{l,m_l}(r,t) \quad (13)$$

$$\begin{aligned}\frac{d}{dr}f_j(t) &= \frac{1}{12h^2}[-f_{j+2}(t) + 8f_{j+1}(t) \\ &- 8f_{j-1}(j-1) + f_{j-2}(t)] + O(h^4) \end{aligned} \quad (14)$$

$$\begin{aligned}\frac{d^2}{dr^2}f_j(t) &= \frac{1}{12h^2}[-f_{j+2}(t) + 16f_{j+1}(t) \\ &- 30f_j(t) + 16f_{j-1}(t) - f_{j-2}(t)] + O(h^4) \end{aligned} \quad (15)$$

where the first and the second derivative have been approximated with fourth order difference formulae.

Formally, the system of equations is written in a compact way by using matrix notation. Since the magnetic quantum number index m_l is a constant of the motion, we may define a vector such that $\mathbf{F}^{(G)}_{m_l}(t) = \{[f^{(G)}_{0,m_l}(r_1,t),\ldots,f^{(G)}_{0,m_l}(r_j,t),\ldots,f^{(G)}_{0,m_l}(r_N,t)],\ldots,[f^{(G)}_{l_{\max},m_l}(r_1,t),\ldots,f^{(G)}_{l_{\max},m_l}(r_j,t),\ldots,f^{(G)}_{l_{\max},m_l}(r_N,t)]\}$ and the matrix,

$$\mathbf{H}^{(G)}_{m_l}(t) = \begin{bmatrix} \mathbf{h}_0 & \mathbf{d}^{(G)}_{01}(t) & 0 & \ldots & 0 \\ \mathbf{d}^{(G)}_{10}(t) & \mathbf{h}_1 & \mathbf{d}^{(G)}_{12}(t) & \ldots & 0 \\ 0 & \mathbf{d}^{(G)}_{21}(t) & \mathbf{h}_2 & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \mathbf{h}_{l_m-1} & \mathbf{d}^{(G)}_{l_m-1,l_m}(t) \\ 0 & 0 & \ldots & \mathbf{d}^{(G)}_{l_m,l_m-1}(t) & \mathbf{h}_{l_m} \end{bmatrix},$$

with $\mathbf{h}_l$ and $\mathbf{d}^{(G)}_{ll'}$ being the $N \times N$ matrix representations of the operators $\hat{h}_{l,m_l}(r)$ and $\hat{d}^{(G)}_{l,m_l}(r,t)$ on the chosen finite-difference scheme respectively. In this case the time-dependent coupled equations, Eqs. (6), are expressed in their final matrix form as:

$$\dot{\mathbf{F}}^{(G)}_{m_l}(t) = -i\mathbf{H}^{(G)}_{m_l}(t)\mathbf{F}^{(G)}_{m_l}(t). \quad (16)$$

The structure of the propagation matrix $\mathbf{H}^{(G)}_{m_l}$ for the length and the velocity gauge is shown in Fig. 1.

Note that this block structure is similar to the block structure obtained with a basis representation of the wavefunction as described in [20] (see Fig. 2). The difference with the present finite-difference formulation is that now the diagonal blocks will not be diagonal themselves but be multi-diagonal. The off-diagonal blocks themselves are now diagonal (length gauge) or multi-diagonal (velocity gauge), while in the basis representation case [20] they are dense.

2.1. Absorbing potential

Up to this point, the formulation has been presented by artificially placing the atomic system inside a sphere of radius R, by forcing the wavefunction to be zero after the fixed boundary. That is, the terms for the finite difference are assumed to be zero after the fixed boundary, much like in the case of an infinitely high box potential. This might be a source of artificial complications, such as the unphysical reflection of parts of the time-dependent wavefunction at the boundaries. To overcome this problem, an optional absorbing boundary has been added throughout the inner region of the box. This is implemented through the addition of an

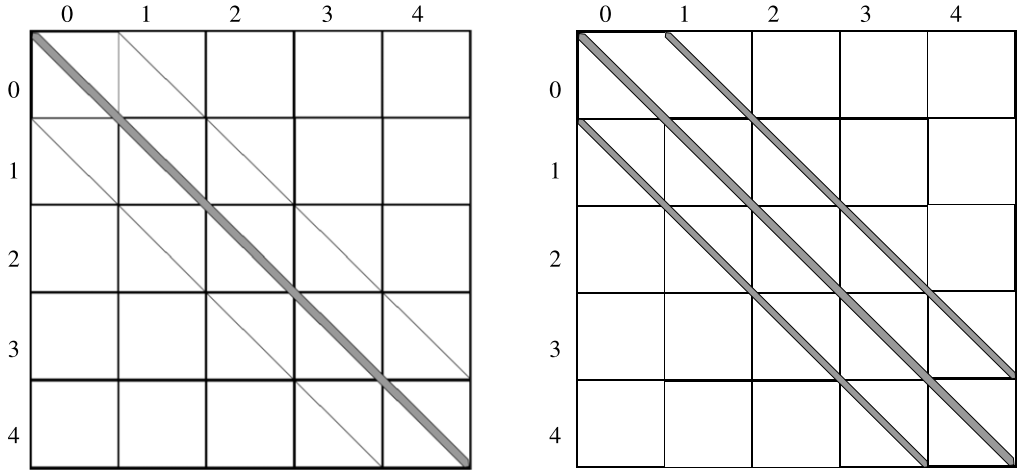

Fig. 1. The banded structure of the finite difference Hamiltonian in the length gauge (left figure) and the velocity gauge (right figure) for $l_{\max} = 4$ is shown. For the case of the velocity gauge, the sub diagonal, super diagonal and diagonal blocks are all multi-diagonal to account for the particular finite difference scheme used for the first and second derivatives.

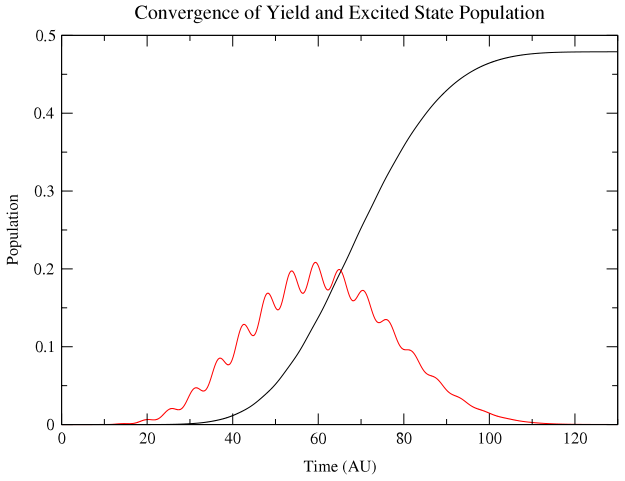

Fig. 2. The sine-squared vector potential returns to zero after approximately 140 au. The solid black line is the yield, while the solid red line is the calculated excited state population. Only the final population values are physically meaningful; the ionised population requires time post-propagation to be radially separated from the excited state population. The yield is calculated with an inner boundary that ignores the first 50 finite difference grid points (r_{Ion}). The excited state population is calculated by subtracting the ground state population and the yield from unity. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

imaginary potential into the Hamiltonian, through the substitution $V(\mathbf{r}) \to V(\mathbf{r}) + W(r)$, where $W(r)$ is [28]

$$W(r) = \frac{i}{dt} \ln\left\{1 - \cos^p\left[\frac{\pi}{2}\left(1 - \frac{r}{R}\right)\right]\right\}, \tag{17}$$

where p controls the *smoothness* of the imaginary potential. Absorption is, essentially, absent at the central region of the system, $\lim_{r\to+0} W(r) = 0$, while it approaches complete absorption at the box boundaries, $\lim_{r\to R} W(r) = -i\infty$.

The imaginary potential not only solves the technical problem of the artificial scattering but it can also be used as a method to quantify the ionisation yield. This imaginary potential smoothly removes probability flux as it approaches the boundaries, affecting only the outgoing part of the wavefunction. Therefore, the loss of norm of the wavefunction could be interpreted as ionisation of the system, although in practice this should be used in conjunction with the spatial integration method of calculating the yield, which is discussed later, so that the yield calculation can be done practically without waiting for all outgoing parts of the wavefunction to reach the box boundaries.

3. Electric field and vector potential

In the present implementation the electric field $\mathcal{E}(t)$ and the vector potential $A(t)$ of the pulse are represented as

$$\mathcal{E}(t) = \mathcal{E}_0 f(t) \cos(\omega_0 t + \phi_{cep}(t)), \tag{18}$$

$$A(t) = -c \int_{t_0}^{t} dt' \mathcal{E}(t'), \tag{19}$$

where ω_0 is the photon energy, $\mathcal{E}_0$ is the maximum of the envelope, $f(t)$ is the pulse envelope and ϕ_{cep} is the carrier–envelope phase, which is the phase difference between the carrier wave and the envelope. Since we have adopted the dipole approximation, it is assumed that both the electric field and the vector potential are approximately constant over the extent of the box at some instant in time; that is, the spatial variation of the fields are ignored [29].

At present, two forms of the envelope are implemented; the Hann function envelope and the Gaussian envelope.

The Hann function envelope (sine squared pulse) adopted is given by

$$f(t) = \sin^2\left(\frac{\omega_0}{2n_c} t\right),$$

where n_c is the number of cycles of the carrier wave.

In particular, for the Gaussian pulse we have also implemented the case where the electric field phase is not necessarily constant, i.e. the pulse can be chirped:

$$f(t) = \frac{\mathcal{E}_0 z_0}{z_d^{\frac{1}{4}}} e^{-\frac{z_0^2 t^2}{2z_d}} \times \cos\left(\omega_0 t + \phi_{cep} + \frac{d}{2z_d} t^2 - \frac{\arctan(d/z_0^2)}{2}\right), \tag{20}$$

$$z_0 = \frac{\tau_d}{2\sqrt{2\ln 2}}, \qquad z_d^2 = z_0^4 + d^2. \tag{21}$$

For a pulse without chirp $d = 0$, τ_d represents the FWHM of the pulse intensity, and the above equation simplifies to

$$f(t) = \mathcal{E}_0 e^{\left(-2\sqrt{2\ln(2)}\frac{t^2}{\tau_d^2}\right)} \cos(\omega_0 t). \tag{22}$$

Having chosen the form of the electric field $\mathcal{E}(t)$ we numerically calculate the potential field $A(t)$, for each time step, as:

$$A(t_{n+1}) = A(t_n) - c\int_{t_n}^{t_n+h} d\tau \mathcal{E}(\tau),$$

where the integral is performed using a standard 5-point Gaussian quadrature. The speed of light factor, c, does not need to be explicitly included in the actual code since there is a $\frac{1}{c}$ factor in the transition operator.

For example, for a potential field $A(t)$ with a Hann function shape, the electric field has the form:

$$\mathcal{E}(t) = \mathcal{E}_0 \sin(\Omega t)\,(\omega_0 \sin(\Omega t)\cos(\omega_0 t) + 2\Omega \cos(\Omega t)\sin(\omega_0 t))$$

where $\Omega_0 = \frac{\omega_0}{2n_c}$. This can be viewed as a combination of two electric fields which are individually of the form given in Eq. (18). The vector potential in the velocity gauge is thus given by

$$A(t) = -\frac{\mathcal{E}_0}{c}\sin^2 \Omega t \sin \omega_0 t.$$

Although the Gaussian envelope represents a very realistic description of the actual experimental fields, the Hann form of the envelope is used very frequently as it has some numerical advantages against the Gaussian envelope. First, in contrast to the Gaussian envelope, which is non-zero for all times, the Hann envelope is strictly zero at precisely known end points $t_i = 0$ and $t_f = \pi/(n\omega_0)$. In addition, the Fourier transform of the Hann envelope results in a sharper cut-off in the spectral distribution for frequencies other than the carrier frequency. This is a very important property for few-cycles pulses where bandwidth effects can have appreciable effects on the final results. This is particularly true when the photon energy is near resonance with transition energies of the system. Finally, in comparison with a Gaussian pulse with the same total electromagnetic energy offered in the system, there is an appreciable amount of computing time that is saved because the extent of the pulse is more limited.

4. Calculation of observables

4.1. Ground state calculation

The *diffusion* equation may be used to locate the eigenstates and associated eigenenergies for a stationary system. In this case, we assume that no field is present and the corresponding Time Independent Schrödinger Equation (TISE) is further modified by performing the replacement $-i \rightarrow 1$ [30]. The resulting partial differential equation is the well known *diffusion equation*. The replacement $-i \rightarrow 1$ breaks the ability of the state evolution operator to maintain unitarity. The partial waves $f_{lm_l}(r,t)$ are re-normalised after each time step to ensure that the wavefunction does not grow overly large and compromise numerical accuracy. The magnitude of the break from unitarity can be used to calculate the expectation value of the Hamiltonian operator, through the following expression [28]:

$$\langle E(t)\rangle\, dt = -ln\,\langle\psi(t+dt)|\psi(t+dt)\rangle\,. \tag{23}$$

It is important to note that the method does not find the true physical ground state, but instead it finds the ground state corresponding to the discretised numerical system. If we start with the analytical ground state, the system will quickly approach the ground state of the numerical approximation of the system. If the time step used is too large, numerical accuracy will be compromised and the algorithm may not converge on the ground state. If the time step is too small, more computational time than is necessary may be used in reaching the ground state. Using the time steps of the standard time dependent calculation provides reasonable values. This should not be used in combination with the complex potential term, from Eq. (17), enabled.

It is also important to bear in mind that the symmetry properties of the initial state also determine which energy eigenstate the system settles into. That is, the method is effective for finding the lowest energy eigenstate of an uncoupled state subspace as it does not require information about any other subspace. For example, if only states of a particular angular momentum l are initially populated then the system can only settle into the lowest energy eigenstate with the same angular momentum l, provided that there are no non-field dependent couplings among different partial waves (as for example in the molecular case). This is because there is no field to couple states which differ in angular momentum.

4.2. Populations

During and after the interaction of the atomic system with the radiation field, the vector of the radial wavefunction $f_{lm_l}(r,t)$ is used within the finite difference approach and so can also be used for the calculation of other quantities. Having the data for the evolution of the radial wavefunctions to hand, the most straightforward quantities to calculate are the ground state population, the excited state population and the total ionisation yield.

The ground state population $p_g(t)$ is calculated by the projection of the initial state $\psi(\mathbf{r},0)$ onto the wavefunction at time t, $\psi(\mathbf{r},t)$,

$$p_g(t) = \langle\psi(0)|\psi(t)\rangle = \int_0^R dr f^{\star}_{0,0}(r,0) f_{0,0}(r,t). \tag{24}$$

The population of the bound states can be calculated by direct spatial integration of the probability values at all grid points inside a carefully chosen radius, say r_b:

$$p_b(t) = \langle\psi(t)|\psi(t)\rangle_{r<r_b} = \sum_{lm_l}^{l_{\max}} \int_0^{r_b} dr |f_{l,m_l}(r,t)|^2. \tag{25}$$

The population of the excited states, $p_e(t)$, can be deduced by knowing the ground state population, Eq. (24), and the bound state population, Eq. (25), as

$$p_e(t) = p_b(t) - p_g(t). \tag{26}$$

The two methods of calculating the yield, below, require post-pulse propagation. This means that the ionised population will be counted as the excited state population during the run of the pulse. That is, the excited state population is estimated by the quantity within a subset of the box which is not counted towards ionisation, and which is not the ground state. As such, one estimates the excited states population as $p_e(t) = p_b(t) - p_g(t)$. Note that it is not until the post propagation, when the ionised population moves away from the central potential and the yield value asymptotically approaches a value as in Fig. 2, that the excited state population becomes meaningful.

4.3. Ionisation yield and angular distribution

In the present case we use two different methods to calculate the ionisation yield; (a) direct spatial integration of the probability values at all grid points outside a chosen radius, say r_b, and (b) use of the probability current directed radially through a sphere of radius r_b, as shown later.

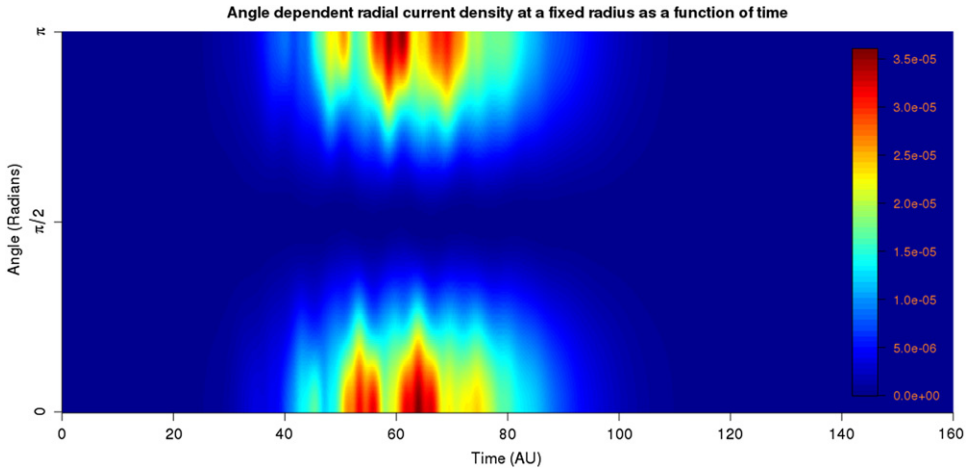

Fig. 3. The strength of the time-dependent radial probability current $j_r(r_b, \Omega, t)$ at a fixed radius $r_b = 9.98$ au and for θ from 0 to π, over the duration of the pulse described in Fig. 8. Since $m = 0$, the angle, ϕ, plays no role.

Spatial integration of the wavefunction. Assuming spatial integration for radius $r > r_b$ we obtain

$$p_i(r_b, t) = \langle \psi(t) | \psi(t) \rangle_{r>r_b} = \sum_{lm_l}^{l_{\max}} \int_{r_b}^{R} dr |f_{l,m_l}(r, t)|^2$$
$$= Norm - \sum_{lm_l}^{l_{\max}} \int_{0}^{r_b} dr |f_{l,m_l}(r, t)|^2. \quad (27)$$

To ensure that a reasonable part of the contributions from continuum energy-eigenstates are counted in the ionisation yield post-calculation, the wave equation is propagated forward in time. This allows the continuum contributions to move away from the central boundary so that the yield can be calculated by counting the probabilities after a certain cut-off radius r_b. The radius should be chosen so that a minimal amount of bound state probability is included in the ionisation yield calculation. This calculation is only valid when the complex potential term is not used to syphon off the probability current heading towards the boundaries. If the complex absorbing potential term is present, the break from the norm should also be added to the yield. When no absorbing potential is present then

$$p_i + p_b = 1, \quad (28)$$

assuming perfect numerical accuracy.

Probability current. The radial probability current of the time-dependent wavefunction is obtained from the probability current $\mathbf{j}(\mathbf{r}, t)$:

$$j_r(\mathbf{r}, t) = \hat{\mathbf{r}} \cdot \mathbf{j}(\mathbf{r}, t) = \hat{\mathbf{r}} \cdot \mathrm{Re}\left\{ \psi^\star(\mathbf{r}, t) \left[\mathbf{p} + \frac{1}{c}\mathbf{A}(t) \right] \psi(\mathbf{r}, t) \right\} \quad (29)$$

$$= -\mathrm{Im}\left\{ \frac{1}{r} \sum_{lm_l} \sum_{l'm'_l} f^\star_{l,m_l}(r, t) \times \frac{\partial}{\partial r}\left[\frac{1}{r} f_{l',m'_l}(r, t) \right] Y^\star_{lm_l}(\Omega) Y_{l'm'_l}(\Omega) \right\}$$
$$+ \frac{1}{r^2}\frac{A(t)}{c}\cos(\theta) \left| \sum_{lm_l} f_{l,m_l}(r, t) Y_{lm_l}(\Omega) \right|^2 \quad (30)$$

with the unit radial vector $\hat{r} = \mathbf{r}/r$ where $\mathbf{r} = (r, \Omega)$. By use of the continuity equation $(\nabla \cdot \mathbf{j}(\mathbf{r}, t) + \partial\rho(\mathbf{r}, t)/\partial t = 0)$, it follows that $j_r(r_b, \Omega, t) r_b^2 d\Omega$ provides the number of electrons moving outwards, per unit time, within a solid angle $d\Omega$ through a spherical surface placed at some distance $r_b < R$. For the differential probability we have:

$$\frac{dP}{d\Omega}(r_b, \Omega, t) = \int_0^t dt' r_b^2 j_r(r_b, \Omega, t'). \quad (31)$$

The time t is chosen to be large enough such that all radial outgoing flux has passed the point of observation r_b. The distance r_b is chosen neither too close to the central region nor to the box boundaries. This is to ensure two things respectively; that no flux corresponding to the system moving into an excited state is included and so that the absorbing potential $W(t)$ has no significant discernible impact, at the radial distance of interest, if it is enabled. Where the absorbing potential is non-zero, the continuity equation is modified as $\nabla \cdot \mathbf{j}(\mathbf{r}, t) + \partial\rho(\mathbf{r}, t)/\partial t = 2W(t)\rho(\mathbf{r}, t)$. We make the choice of r_b in order to avoid any complications introduced by the modification of the continuity equation.

For a fixed radius r_b, the quantity $j_r(r_b, t)$, shown in Fig. 3, can be used to calculate the ionisation current. It is self evident that the square of the time integral of the probability current flowing through the sphere of radius r_b is exactly the population outside the sphere, considering that the initial population outside the sphere is effectively zero. For this method, adding the break from the norm is not required. Therefore, by integrating Eq. (31) over the angular variables we obtain the ionisation yield $p_i(t)$ at time t as,

$$p_i(r_b, t) = \int_0^t dt' \sum_{l,m_l} \left\{ \mathrm{Im}\left[f^\star_{l,m_l}(r_b, t') \frac{\partial}{\partial r} f_{l,m_l}(r_b, t') \right] + \frac{2}{c} A(t) \kappa_{l+1,m_l} f^\star_{l,m_l}(r_b, t') f_{l+1,m_l}(r_b, t') \right\}. \quad (32)$$

The above quantity, over the lifetime of the simulation, provides a separate measure of the ionisation yield.

4.4. High harmonic generation

In the particular case where $m_l = 0$, we also calculate the expectation value of the dipole moment, dipole velocity, and dipole acceleration, using the following formulae [31]:

$$\langle z(t) \rangle = 2 \sum_l \kappa_{l+1,0} \mathrm{Re} \int_0^R dr r f^\star_{l,0}(r, t) f_{l+1,0}(r, t) \quad (33)$$

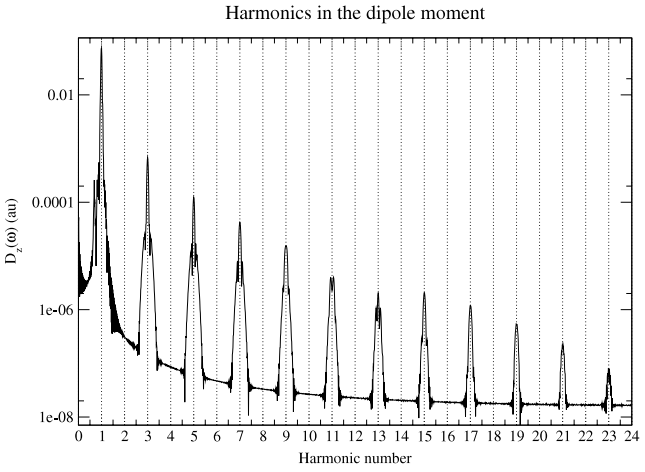

Fig. 4. The Fourier transform of the dipole moment for a hydrogen system under the action of a pulse of intensity 1×10^{15} W/cm^2, central photon energy 15 eV, 40 cycles and with a sine-squared shape for the vector potential. Multiples of the photon frequency, on the *x* axis, are plotted against the dipole moment strength in atomic units on the *y* axis.

$$\langle \dot{z}(t) \rangle = \sum_l \kappa_{l+1,0} \mathrm{Im} \left[\int_0^R dr r f^\star_{l+1,0}(r,t) \frac{\partial}{\partial r} \left(\frac{1}{r} f_{l,0}(r,t) \right) - l \int_0^\infty dr f^\star_{l+1,0}(r,t) \frac{1}{r} f_{l,0}(r,t) \right] \quad (34)$$

$$\langle \ddot{z}(t) \rangle = -E(t) - 2 \sum_l \kappa_{l+1,0} \mathrm{Re} \int_0^R dr \frac{1}{r^2} f^\star_{l,0}(r,t) f_{l+1,0}(r,t). \quad (35)$$

Since the wavefunction is set so as to be finite in extent, the integration can be terminated at the end of the box R instead of at infinity. Fig. 4 is an example plot that clearly shows odd harmonics spaced by twice the photon frequency. Clearly if part of the wavefunction is removed by the complex potential, or reflected from the box boundary in the alternative situation, the result can be influenced. Thus, care must be taken to ensure that box length is adequate for the system under study.

5. Description of the code

OpenCL is a specification that attempts to standardise the use of co-processors. The current version of the standard is 2.0 (Rev 19), and was released by the Khronos OpenCL group, the multi-vendor committee responsible for the specification. It was at the suggestion of Apple® that the initial specification came about.

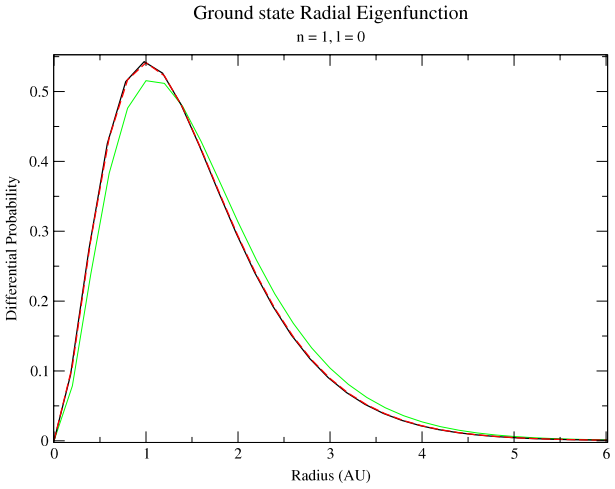

Fig. 6. The differential probability for the radial function for the 1s ground state of hydrogen is calculated with a grid of size 800 au and $dr = 0.2$ au. The initial calculation without an offset is given by the green line. The system with an offset is given by the black line. The dashed red line indicates the analytical radial probability given as $4r^2e^{-2r}$ in atomic units. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

CLTDSE is provided as a stand alone program, and not a library. See Fig. 5 for the first level of the call graph.

Code reusability and extensibility. The overarching concept of CLTDSE 's framework design is *reusability* of all functions. That is, the functions that are lowest on the function tree should also be the most generic in their operation. Functions should have narrowly defined, but generically implemented, functionality. Tasks such as the opening of files, or the allocation of memory, are generally passed to functions higher up the tree where reasonable.

Extensibility is also important because not all possible use cases can be anticipated, and the alteration and generation of code at a high level up the function tree is desired, That is, lower level functionality should be able to be called upon in unforeseen circumstances, but within the scope of the function, and still be expected to work.

For example, CLTDSE was initially written for the basis method [20], but was extended to duplicate calculations for the TDDM system described in [32], and also to work for the present finite difference implementation. The intention of the framework is for it to be suitable for use cases which rely on a single compilation of the OpenCL code. It has not been tested for use cases involving repeated initialisation and deallocation of the framework.

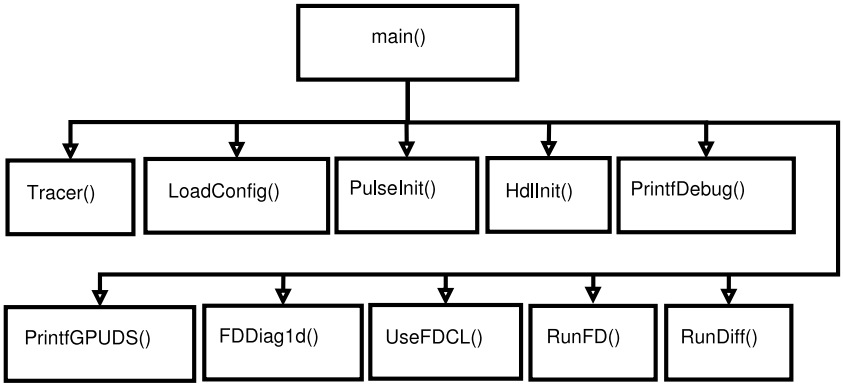

Fig. 5. The first level of the call graph for the main() C function. *Tracer*, *PrintfDebug*, *PrintGPUDS* provide simple debugging. *LoadConfig* is the entry point into the *Configuration file* reading. *PulseInit* initialises the Pulse structures. *HdlInit* initialise the OpenCL structures. *FDDiag1d*, and *UseFDCL* initialise the necessary structures, compile the specific OpenCL C code, and allocate the required OpenCL memory objects. *RunFD* is the entry point into running the TDSE propagator and *RunDiff* is the entry point into the Diffusion equation propagator.

With initialisation, performance considerations are generally not a primary concern, as almost all of the total running time is dictated by the specific computation. Only in the smallest computations does initialisation bear any reasonable performance penalty. As such, in very short but repeated computations, tailoring specific code to handle the repeated computation is more desirable than scripted re-runs of the program. Scripted re-reruns will involve repeated initialisation and repeated compilation. The extensibility of the code plays an important factor here.

There are 5 broad areas within the code;

1. **Input**: *ConfigReader.c*, *ConfigReader2.c*
2. **Output**: *Output.c*
3. **Pulse**: *PulseInit.c*, *Pulse.c*, *Integrators.c*
4. **OpenCL**: *clFunc.c*, *clDebug.c*, *clInit.c*
5. **TDSE**: *FiniteDiff.c*, *UseFDCL.c*, *Taylor.c*, *Lanczos.c*, *RungeKutta.c*

The **OpenCL** and **TDSE** sections rely on the *OpenCL framework*. The **Input** section is independent of the computing framework used for the propagation, while the **Output**, and **Pulse** sections make use of the OpenCL double precision datatypes for compatibility but are otherwise easily separated from the framework with only minimal effort required.

5.1. Input

Initially the *configuration file* (examples are shown later), which is specified as an argument to the program, is loaded into structures in memory. The configuration file provides the necessary information to specify the problem fully. The ability to read the configuration file for CLTDSE comes from the *libconfig* library [33]. Thus the syntax is that listed by the libconfig maintainers. libconfig based configuration files are easy to follow in their structure and also extensible to unanticipated configurations. In order for the configuration file to be read, CLTDSE contains two sections (logically separated and in separate files):

ConfigReader2.c provides generic functionality to read configuration files and place the data into the appropriate memory location based on a passed array of structures. Each member of the array represents a specific setting. Each array of structures is terminated with a NULL structure.

ConfigReader.c is set to pass the specific arrays of structures required by *ConfigReader2.c* for reading settings. This array of structures approach is only used where feasible. It is sometimes more practical for the functions in *ConfigReader.c* to access the libconfig functions directly, such as for the necessary translation of strings to enumerated types which are used internally within CLTDSE. The tedious conversion of converting character arrays to enumerated types is done to facilitate the ease of future comparisons and the ease of using switch statements etc. Performing the necessary conversion in a single location provides clarity.

This configuration file handling is an example of the *reusability* design of the code. The functions of *ConfigReader2.c* provide extensible configuration reading, whilst *ConfigReader.c*, higher in the function tree, has more problem specific and less generic functionality. This then lends itself to modularity and possible code re-use for parts of the code.

5.2. Output

The produced output files are given in Table 1. In particular, the file *output.c*, contains the following externally facing (in terms of the API) functions:

Table 1
Output files after the propagation of the CLTDSE executable.

File name	Description
laser.dat	The laser pulse, including the electric field, vector potential and the intensity.
output.dat	Time dependent functions which return only a single value per time step.
current.dat	The angle dependent probability current through a fixed radius.
radialt.dat	The time dependent radial distribution at a particular time t.
ProbCurr_ll.dat	The angle dependent probability current density for a particular partial wave value l.

PrintFD() outputs the angle-integrated population, as well as the population of the angle integrated partial waves $\left|f_{l0}(r_j, t)\right|^2$, at each radius r_j. Currently this is only used in the diffusion equation function *clDiffusion*.

jPrintCurrentFD() calculates the angle-dependent radial probability current, as given by Eq. (31) at a particular radius r_b (determined by the configuration file setting IonPos). For each time step it is printed to a file. Graphs such as Fig. 3 can be produced with the data (in this instance using R).

PrintHarmonic() prints the population of the gauge dependent partial waves $\left|\frac{1}{r}f^{(G)}_{l,0}(r, t)Y_{l0}(\theta, \phi)\right|^2$ in terms of θ and r. The data is printed into separate files of the form ProbCurr_ll.dat, where l is the angular momentum.

PrintOut() is responsible for printing all the time dependent functions which return a single value at each time step to a file. Values are printed in the following order: the time, the ground state population (Eq. (24)), the ionisation yield estimate (Eq. (27)), the excited state estimate (Eq. (26)), the dipole moment (Eq. (33)), the dipole velocity (Eq. (34)), the dipole acceleration (Eq. (35)), the ionisation current at a fixed radius (Eq. (32)), the norm (Eq. (28)), and the difference between unitarity and the actual norm (with the difference being due to numerical limitations and the absorbing potential, if applicable). These values are printed to the file *output.dat*.

PrintPulse() is also an output function but unlike the others it is located in Pulse.c. For each time step, the time t, the electric field $\mathcal{E}(t)$, the vector potential $A(t)$, and the intensity $I(t)$ are printed out to the specified file.

5.3. Pulse

After the configuration file has been loaded, the pulse is initialised:

PulseInit() in *PulseInit.c* is the entry point into pulse initialisation. The pulse initialisation includes (a) specifying the gauge of the problem, (b) the form of the pulse(s), (c) conversion of settings to atomic units, if applicable, and (d) the calculation of the length of time required for propagation.

During the simulations, when the electric field $\mathcal{E}(t)$ (*PulseE()* in *Pulse.c*) or the velocity gauge vector potential $A(t)$ (*A()* in *Pulse.c*) is required, the pulse is calculated on the CPU within the host code and transferred to the specific compute device.

5.4. OpenCL

Next we describe some important functions and structures that are essential for the establishment of the OpenCL framework, and important for queueing kernels during execution. The actual calculations are performed through OpenCL C code. The C code merely deals with management issues such as copying arrays

and passing in new pulse values, as well as queueing kernels for execution. All mentioned functions are located in *clFunc.c* unless otherwise noted.

5.4.1. Initialisation

clHandle is a structure used to store handles and objects returned from OpenCL API function calls. Initialisation functions return relevant handles to relevant devices, queues, contexts etc. These handles are stored within a single structure, designated as type clHandle. This allows the construction of an API where internal layout is irrelevant to those writing functions which make use of the framework. After pulse initialisation, OpenCL is initialised and a *clHandle* structure is allocated for use by calling the function *HdlInit()*.

allocHdl() follows the specific sequence of function calls to initialise OpenCL which are laid out in the OpenCL specification. Prior to calls to the OpenCL API, *allocHdl* is called to allocate memory for the new handle, the platform and device Ids. *allocHdl* also initialises the unused pointers and variables within the *clHandle* structure to *NULL* and *zero* respectively. The number of programs that will be compiled must also be specified at this point. A program is defined, in this context, as a set of OpenCL kernels which are executed to perform one task; for example in the current case there are *Taylor*, *Lanczos*, and *Runge–Kutta* programs. allocHDL relies on the following OpenCL library function calls:

- *clGetPlatformIDs()* A platform ID is acquired through the OpenCL function *clGetPlatformIDs()*. In principle this function can be used to select between different implementations of the OpenCL library.
- *clGetDeviceIDs()* This allows compute devices of a specific type to be selected. The three device types are CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_GPU and CL_DEVICE_TYPE_ACCELERATOR. In the present case the latter has not been tested as an option (due to the lack of non-GPU accelerators available). Devices can be further split into sub-devices, although for the present case this is undesirable and so has not been performed.
- *clCreateContext()* associates the selected devices together into a single context. Later, when kernels are compiled, they will be available for execution on any device within the context. A callback function can be associated with the context to provide error handling. The necessary function pointer is passed to the *clCreateContext()* function.
- *clCreateCommandQueue()* creates a queue. For every compute device within the context, a separate queue is created. Kernels are added to this queue for execution on the associated compute device. Currently only one compute device at a time is supported in CLTDSE.

UseFDCL() in *FiniteDiff.c* allocates and initialises the structures necessary for the finite difference approach.

5.4.2. Kernel execution

clExecuteSizedKernel() is a minimal function which calls *KernInit()* to ensure the memory objects for the kernel of interest are initialised, and to also call the OpenCL function *clEnqueueNDRangeKernel()* to enqueue a kernel. Kernel execution is asynchronous; that is, when you enqueue a kernel to be executed it is not necessarily executed before the *clEnqueueNDRangeKernel()* function returns. This ensures that the GPU can be throughput dependent rather than being latency dependent; it can build up a batch of kernels and then pass the data over the PCI bus. During kernel execution, at occasional intervals, the radial grid array is read and processed for output. When the kernels are being executed a number of functions provide information on the dimensions of the problem, while other functions provide the specific place of each instance of the kernel within the problem. The OpenCL function *clEnqueueNDRangeKernel()* is always called with the number of work groups and the number of work items as arguments.

5.4.3. Work items

SetupGroups() dictates the work group size. When OpenCL kernels are executed they are not executed in a single thread but simultaneously by multiple "threads" known as work items; i.e. an instance of a kernel is called a work item. Work items execute instructions in a SIMD fashion with a collection of other work items called a work group. In the host code, an ideal number of work groups and work items must be passed as arguments to *SetupGroups()*. In *SetupGroups()*, through a call to *clReqSizeTInfo()* to get the maximum allowed work group size, the ideal value is modified to take into account the work group size limits. The function *SetupGroups()* sets up a default configuration of work groups and their sizes for the calls to *clExecuteKernel()*. This default configuration can be overlooked by executing kernels through *clExecuteSizedKernel()*.

Within an executing OpenCL kernel:

get_global_id() provides a number identifying a particular work item with respect to the other work items. The integer it returns ranges from 0 to *get_global_size()* −1.

get_local_id() provides the number of the work item with respect to the other work items in its work group. The integer returned ranges from 0 to *get_local_size()* −1.

A work group is simply a logical grouping of work items that also share a common local memory. Since problems can be multi-dimensional, the parameter of the above functions is an integer indicating the particular dimension. For the parallelisation of the calculation of a new vector only 1 dimension is required, so 0 is used as the argument to refer to that 1 dimension, e.g. *get_global_id(0)*.

$\mathbf{F}_{l0}(r_j, t)$ is dependent on a value of l and j, both of which are integers. The simple approach of calculating l and j can be expressed in 2 very simple lines. l indicates the angular momentum and it is simply calculated by dividing the global id by the number of grid points per l block (*GRID_SIZE*):

$$l \leftarrow get\_global\_id(0)/GRID\_SIZE$$

Note that in this division we are relying on a property of integer division. In C, division of integers is not like the standard floating point division. Integer division rounds down after dividing two integers, i.e. $(int)3/(int)2 = 1$.

If we know the particular angular momentum l and if the value is multiplied by the number of states $l * GRID\_SIZE$, that gives the global id corresponding to the start of the block. If we subtract that value from the global id:

$$j \leftarrow get\_global\_id(0) - l * GRID\_SIZE$$

we find the spatial coordinate j. If this property of integer division was not used then it is clear that $j \equiv 0$.

Previously, when the kernels were executed the assignment of work was calculated according to the algorithms, as discussed in [20], which ensured that for each coefficient to be calculated (via a dot product) there is one thread. It also ensured that the threads

were logically grouped into work groups by the part of the array they are calculating. Although the complexity of the method added no tangible computational burden for the case of the basis set, a much simpler approach is used for finite difference because there is no large Hamiltonian stored in memory to read from.

5.4.4. Compilation and object initialisation

As well as setting up the work group structure in the host code, the specific kernels that will be used for finite difference must be compiled. Several functions help to wrap up the OpenCL build functions, and to manage the OpenCL memory objects:

BuildInitKern() is the highest function in the tree for compilation, it calls the necessary subfunctions which handle compilation (*clBuildFile()*), kernel object allocation and initialisation (*SetupKerns()*).

clBuildFile() takes the specified OpenCL C source files and any additional strings specified and calls the necessary functions to load and compile them. Programs, and thus kernels, are compiled for every device within a specified OpenCL context. clBuildFile is associated with a specific program with the total number of programs having been decided by *HdlInit()*. *clBuildFile()* relies on the following OpenCL functions:

- *clCreateProgramWithSource()* prepares the source file for compilation and returns a program handle.
- *clBuildProgram()* performs the actual compilation associated with a program handle.

clBuildInfo() provides any errors or warnings returned by the compilation attempt.

SetupKerns() allocates the structures that hold the kernel memory objects, and the kernel handles as well. The allocation and initialisation of kernel handles is done through the function *AllocKern()* which, in turn, relies on the OpenCL function *clCreateKernel()*. *AllocKernArgs()* allocates memory in the clHandle structure to link to specific memory objects which are then initialised by *SetupObjs()*.

The specific memory objects and forms of the kernels for the various methods varies greatly; the initialisation functions in *clInit.c*: *clBuildLanczos()*, *clBuildTaylor()*, *clBuildRungeKutta()* are distinct, as well as the code for kernel enqueuing for the Taylor expansion (*Taylor.c*), the Lanczos method (*Lanczos.c*) and the Runge–Kutta methods (*RungeKutta.c*). They share the above functions for initialising, compiling and building which have been implemented and the Pulse API.

Currently the code assumes that each angular momentum block of the radial vector contains the same number of coefficients, since this suits the particular use case discussed here. To modify this would only require modifying the two lines of code which correspond to defining l and j, and possibly the number of work items enqueued via *clEnqueueNDRangeKernel()*.

5.5. TDSE

The actual calculations are performed through OpenCL C code. The C code for each method merely deals with management issues such as copying arrays and changing the pulse values, as well as queueing kernels for execution. For the propagation schemes implemented in the code, the main bottleneck operation is the matrix–vector multiplication. In particular, we have implemented (a) the Runge–Kutta methods, (b) the Taylor series, and (c) the Krylov based Lanczos algorithm methods. The Taylor and Runge–Kutta kernels described here, are the same as those described in [20], except that the kernels have now been re-purposed for finite difference.

Taylor.c. The main feature of the Taylor series propagator is its simplicity. Unfortunately as a method, it is not always stable, although it converges quite well when propagating the hydrogen system. For helium systems, as in [23], the Taylor series was also noted to be very reliable. Considering the propagation of Eq. (16), we evaluate the vector $\mathbf{F}_{n+1} = \mathbf{F}(t_n + dt)$ by evaluating the following Taylor expansion:

$$\mathbf{F}_{n+1} = \sum_{p=0}^{P} \mathbf{F}_n^{(p)} \qquad \mathbf{F}_n^{(p+1)} = \frac{-idt}{p} \mathbf{H} \cdot \mathbf{F}_n^{(p)}. \tag{36}$$

Since the Taylor expansion of the exponential operator is such a simple expression, it also leads to very simple OpenCL C code. Only one kernel is required, which performs the calculation given by Eq. (36) and adds it to the solution for $\mathbf{F}(r, t + dt)$.

RungeKutta.c. The Runge–Kutta methods are of the form:

$$\begin{aligned} \mathbf{F}_{n+1} &= \mathbf{F}_n + h \sum_{i=1}^{S} b_i \mathbf{k}_i, \\ \mathbf{k}_i &= \mathbf{F}\left(t_n + c_i h, \mathbf{F}_n + h \sum_{j=1}^{i-1} a_{ij} \mathbf{k}_j \right). \end{aligned} \tag{37}$$

where b_i, c_i, and a_{ij} are values dependent on the specific method.

The Runge–Kutta algorithm is broken into 3 kernels. One kernel performs the summation on the RHS of (37), while the second kernel performs the derivative calculation. Finally, the third kernel adds the linear combination of multiple derivatives. The pseudocode for each time step is:

$i \leftarrow 0$
while $i < P$ **do**
  $\mathbf{g} \leftarrow \mathbf{f}_n + h \sum_{j=1}^{i-1} a_{ij} \mathbf{k}_j$
  $\mathbf{k}_i \leftarrow \mathbf{f}(t_n + c_i h, \mathbf{g})$
  $i \leftarrow i + 1$
end while
$\mathbf{f}_{n+1} \leftarrow \mathbf{f}_n + h \sum_{i=1}^{S} b_i \mathbf{k}_i$

Since, for each derivative calculation, a vector potential is calculated with a Gaussian quadrature and then added to the previous vector potential, care must be taken to ensure the vector potential is handled correctly. Subtle errors can be introduced by an incorrect sequence. Since we are not concerned about explicitly including the speed of light term c, using the substitution $\tilde{A}(t) = A(t)/c$, the calculation performed during the derivative calculation to be approximated is: $\tilde{A}(t_n + c_i h) = \tilde{A}(t_n) + \int_{\tau=t_n}^{t_n + c_i h} d\tau E(\tau)$. Ignoring the c term, after each time step a calculation much like Eq. (3) has been performed.

Lanczos.c. Lanczos propagation, here, refers to the application of the Lanczos method to generate a smaller system that can then be propagated with other standard methods. The kernel for the Lanczos method is also more complicated in comparison to the RK and Taylor kernels.

The Lanczos algorithm is broken into 5 kernels. Multiple kernels are required because global memory synchronisation only occurs between all work groups after the execution of a kernel. That is, only local synchronisation, which occurs between members of the same work group, can occur during kernel execution. This restriction is present to easily allow for different execution models with no impact on the results. Different methods of execution include queueing the kernels to execute serially, in parallel, or both, depending on the compute device.

The first kernel, *LanczosLoop1()* performs the necessary power-iteration matrix–vector calculation; although in the current context the matrix is not explicitly stored. The kernel also starts the two-kernel process of calculating α through reduction. Two kernels are required since we need to perform a reduction

across work groups. In the first function a local reduction is performed for each work group. Then a second reduction is done in *LanczosLoop2()* with each work group to independently calculate α.

For a GPU, there is a standard method for performing a reduction. It is a method of performing a dot product that relies on as little communication, and thus synchronisation, as possible. Local memory is used for the reduction where possible because access latency is lower.

Initially items are read from the global array pArr[] into the local Work[] in such a way that if the global memory is larger than the local memory each work item adds the correct items. Abbreviating get_local_id(0) by LID, and get_local_size(0) with LSZ, and where Len is the length of the array to be added, we have:

```
1  int NumTile = Len / LSZ;
2
3  Work[LID] = (LID < Len ? pArr[LID] : 0.0);
4  for (int i = 1; i < NumTile; i++)
5      Work[LID] += pArr[i*LSZ + LID];
6
7  int Rem = (NumTile + 1) * LSZ + LID;
8  if (Rem < Len)
9      Work[LID] += pArr[Rem];
```

After this, a for loop is entered, where at each step, the number of active work items is reduced by half. This is done by bit shifting the value of an integer *i* initially set to half the work size down 1 bit each step in a for loop, which is equivalent to division by 2. On each loop, each work item checks if its local Id is less than the loop iterator value *i*. If the local Id is less than the loop iterator, then the work item grabs an array element which is *i* further down the array from the element in position *LID*, and adds it to the element in position *LID*:

```
1  for (unsigned int i = LSZ >> 1; i > 0; i >>= 1)
2  {
3      barrier(CLK_LOCAL_MEM_FENCE);
4
5      if (LID < i)
6          Work[LID] += Work[LID + i];
7  }
8  barrier(CLK_LOCAL_MEM_FENCE);
```

The work items in the work group, some of which are performing no work, must still all reach the local memory synchronisation barrier.

As well as performing the α calculation in *LanczosLoop2()*, we perform the orthogonalisation procedure on the new vector in the Krylov subspace, as well as performing the first step in a reduction to calculate β.

To complete the Krylov subspace algorithm, in *NormKrylov()* the value β is calculated through the parallel normalisation of the new vector by reduction.

The function *Taylor1()* implements a simple parallel algorithm for the Taylor series which uses only one work group, while Taylor2 calculates $\mathbf{f}(t+dt)$ from $\tilde{\mathbf{f}}(t+dt)$ (the expansion from the subspace back to the state space).

6. Using the program

6.1. Installation

The Installation of OpenCL is hardware and distribution dependent. For standard CPU and AMD GPU usage, the AMD Accelerated Parallel Processing (APP) package should be used (currently on v2.8). Alternatively the Intel SDK can be used for CPU only. The NVIDIA OpenCL driver only works on Nvidia GPUs. OpenCL support is built into the standard Nvidia driver. Nvidia Optimus systems may also require the *Bumblebee* program since the GPU switching technology is currently not implemented for Linux based systems. The installation of Bumblebee is system dependent.

The package *libconfig* should also be installed for the configuration file processing. This package is available for most common GNU/Linux distributions.

The distributed archive contains the source code (*src*), the OpenCL source code (*osrc*), a directory for the binary (*bin*) and a run directory (*run*). CLTDSE should be run from the run directory. The run directory contains a symbolic link to the OpenCL source code directory of the same name, a directory for holding input configuration files *inp*, a directory for holding the generated ground state and for output *out*.

In the *src* directory, the commands

```
cltdse_fd/src> make clean ; make cleansrc
cltdse_fd/src> make CLTDSE
```

remove the object files and most of the temporary files generated by text editors and those files generated by clang during analyse mode, and then compiles the code.

For the makefile, *gcc.cfg* and *clang.cfg* are provided. CFLAGS dictates the compiler flags used. Two options are given for each compiler, one for debugging, and an option for regular optimised compilation. Uncomment the desired flag (and comment out the other flag when applicable).

Execution of the program is quite simple. The binary is run with the location of a specific configuration file as a parameter, e.g.:

```
cltdse_fd/run> bin/CLTDSE inp/Diff1.cfg
```

The configuration file contains all of the relevant information that describes the problem to be simulated.

Each example given is a progression from the previous example to add additional complexity. In the configuration files, a number of settings are specified to handle the location of files for output, the pulse, the size of the system, as well as the number of discrete points represented and settings for the propagator itself.

6.2. Example 1: Example1.cfg

In the example configuration file *inp/Example1.cfg*, a hydrogen system is subjected to an electric field. This file consists of a number of separate entries called *Groups* that serve to configure the run. The following provides a brief description of the various configuration groups included in this example:

Group **Files**: The various setting included in this group define the location of various files as follows:
- Setting *osrc*: sets the location of the OpenCL C source code directory
- Setting *Data*: sets the location to store the ground state function
- Setting *Laser*: sets the location to print laser output
- Setting *Plot*: sets the 2d plot of partial waves
- Setting *Radial*: sets the location to print the square of the radial wavefunction
- Setting *Time*: sets the location to print time dependent outputs excluding the radial wavefunction
- Option *Divisions*: indicates the approximate number of discrete points that will be in the time dependent data set, with a minimum value being the same number of points in the propagation scheme.

Group **Output**:
- Setting *TimeDep*: 1 indicates that time dependent output should be printed to file, while 0 indicates no time dependent output is to be printed.

Since this output is currently not performed asynchronously, this can provide a decrease in total runtime when the time dependent outputs are not desired.

Group **System**:
- Setting *Type*: set to *"Finite"* for the finite difference simulation.
- Setting *Equation*: set to *"TDSE"* for the TDSE calculations or *"Diff"* for the case of the diffusion equation approach for generating the ground state.

Group **Matrix**:
- Setting *Gauge*: selects the gauge by setting *"Length"* or *"Velocity"*.

Group **Propagators**: is a group that provides the settings for the time integration.
- Setting *Device*: can be set to *CPU* or *GPU* depending on whether you wish the simulation to be run on the CPU or GPU
- Setting *WorkItems* indicates the number of work items per work group you wish for the simulation. It is generally recommended that the number of work items have at least 32 as a divisor due to general GPU structures. In the case of AMD graphics cards 64 is the minimum granularity for executing work items; numbers of work items less than 64 simply result in wasted execution cycles (this is referred to as a "wavefront" in AMD terminology)
- Setting *Method*: is set to *"Taylor"*. The other settings are *"RungeKutta"* and *"Lanczos"*. (Since *"Taylor"* is selected in the current example, the group *Taylor* is also present here, for which the *setting Order* is set to the order of the Taylor propagator.

Group **FD**: The properties of the finite difference grid are specified in this group.
- Setting *NumBlocks*: sets the number of angular momentum blocks.
- Setting *Numdx*: sets the number of grid points per angular momentum block.
- Setting *Len*: is an array of length 1, indicating the radial dimension of the box in atomic units. Typically while pulse parameters are specified in standard units, the box dimensions are in atomic units. The system is in a "box" of radius 800 au and 4000 grid points, meaning $dr = 0.2$ au.
- Setting *Start*: Since the finite difference starting location is offset from zero, this indicates the number of grid points the system is shifted away from $r = 0$. For example, if $Start = 1$; is specified then the initial grid point $r_0 = dx$.
- Setting *Offset*: allows a general offset to be specified to apply to the grid, to offset the grid by a specific amount specified in atomic units. Note that if the offset is beyond a small tweak it will mean that a different integration technique would need to be used which is more sophisticated than Simpson's rule.

After having configured the configuration file for the time-dependent run, we have to prepare a configuration file in order to generate the ground state of the atomic system, in the present case of hydrogen. To this end, the *inp/Diff1.cfg* configuration file provides the required settings. For the ground state calculation changing *Equation* = *"Diff"* provides enough settings to generate the ground state for the system. Note that the field does not need to be specified at this point.

```
cltdse_fd/run> CLtdse inp/Diff1.cfg
```

Running CLTDSE with the configuration file *inp/Diff1.cfg* will generate and save the radial eigenfunction for the ground state. Without tweaking the inner boundary, the ground state energy is given as approximately −0.47 au, and the radial function is as shown in Fig. 6. It may be noticed that, compared to the analytical ground state, the calculated state is a rather poor match. By setting *Offset* within the group *FD* as is done in *inp/Diff2.cfg*, one can introduce a slight modification. An offset of −0.02 au returns a radial function much closer to the analytical function, and also with a closer energy value of approximately −0.5 au. From the perspective of an *ab-initio* calculation of the ground state this tweaking to fit the result is undesirable, but considering that we are interested in the time dependent modification of the states, rather than the *ab-initio* generation of states, this approach is fairly standard.

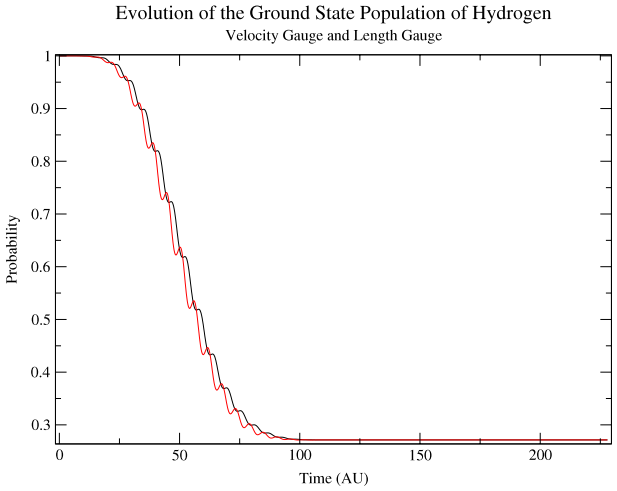

Fig. 7. The population of the ground state of a hydrogen system as it is under the action of the sine-squared vector potential. Shown in red is the length gauge result, while in black the velocity gauge. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

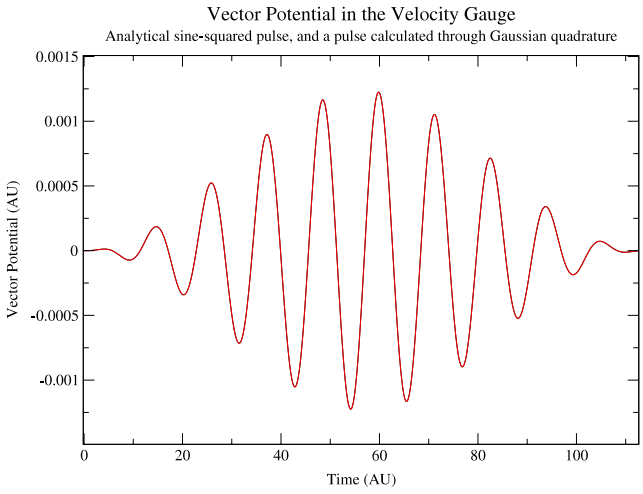

Fig. 8. The analytical vector potential compared to the vector potential calculated through a 5 point Gaussian quadrature. The pulse properties are 1×10^{15} W/cm^2, 10 eV and 10 cycles.

At this stage we run the executable with argument the Example1.cfg configuration file. As explained in detail the settings in this file have been selected to support a time-dependent propagation:

```
cltdse_fd/run> CLtdse inp/Example1.cfg
```

In the particular case of the present example, the vector potential generated from the implemented 5-point Gaussian quadrature when compared to the analytical form agrees very well as shown in Fig. 8. The output vector potential does not include the

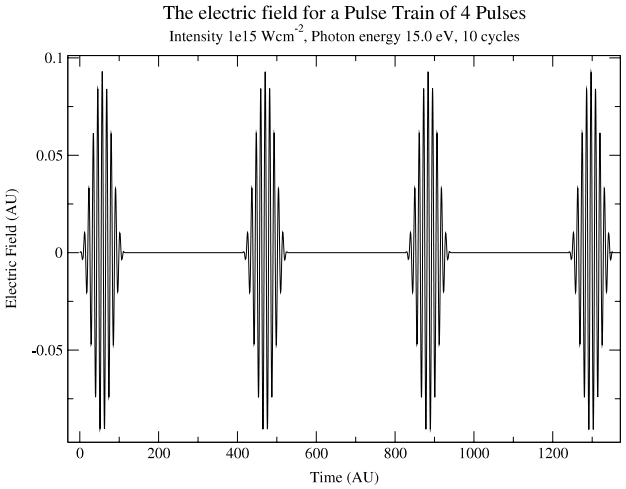

Fig. 9. The electric field for a pulse train separated by 10 fs peak to peak.

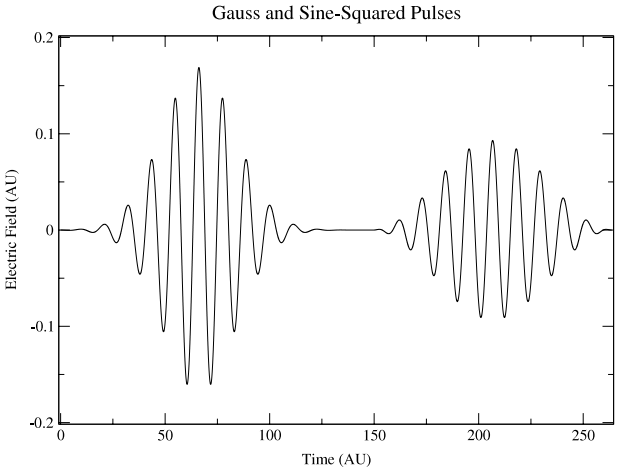

Fig. 10. The electric field for a Gaussian and a sine-squared pulse. The Gaussian pulse has the same intensity and photon energy as the Sine-Squared pulse, 1×10^{15} W/cm^2 and 15 eV respectively, but the full-width half maximum is 2 fs.

multiplication by the fine structure constant required for atomic units; this multiplication can be performed after the fact.

To generate a pulse of this form, in the group *Field*, an array of groups is defined called *Laser*. Each entry in this array defines the parameters for a particular pulse. In the current example, where the array contains only one group:

Units is set to *"Standard"* to indicate that the numbers given are in standard units for the laser parameters (eV, W/cm^2, femtoseconds (fs), etc.).
Intensity is set to *1e15*, which denotes an intensity of 1×10^{15} W/cm^2.
W is set to 15.0 for 15 eV. The number is written in floating point style to indicate that it is not an integer. Internally the parameters are converted to atomic units (by *UnitsConvert()* in *PulseInit.c*).
Shape is *SineA* to indicate that the vector potential should be of sine squared form, thus using Eq. (3).
SineSqr is the subgroup that provides further details about $\sin^2(x)$ pulse envelopes. The setting used within this group is:
Cycles which indicates the number of cycles present in the pulse and is set to 10.

During execution, the pulse generated by the above parameters is printed to the directory specified by *Laser* in the *Files* group. Multiple columns are printed: the time, the electric field, the vector potential, and the intensity.

The evolution of the ground state population, calculated using Eq. (24) is shown in Fig. 7. While the final ground state population is 0.27145, if the gauge is changed to the length gauge by changing the *Gauge* setting to *"Length"*, the calculated population is 0.27144.

During the calculation the outputted value for the ionisation yield should only be used as an approximate indicator of the fastest components of the yield and thus has limited utility. This is due to the yield calculation requiring the system to continue propagating post pulse so that the continuum components can be spatially separated from the bound states. The yield output is shown in Fig. 2.

6.3. Other example cases

By the choice of various settings in the *configuration files* one can investigate different cases. For example in *inp/Example2.cfg* a train of pulses is used, which is shown in Fig. 9. This is specified by adding the settings *PulseTrain* and *Separation*.

To change the pulse given in *inp/Example1.cfg* to a Gaussian form, *Shape* must be changed to *"Gauss"* and the settings *TauD* and *d* must be specified under the subgroup *Gauss* within the *Laser* entry, as shown in *inp/Gauss.cfg*. A Gaussian and a sine-squared pulse can also both be present at the same time as in Fig. 10, by simply combining the two separate groups from the *Laser* entries in *inp/Example1.cfg* and *inp/Gauss.cfg* as is done in *inp/Example3.cfg*. The extra *TimeShift* entry is used to specify the shift of the peak of the second pulse from the $t = 0$ point.

A clear high harmonic spectrum can be produced, as shown in Fig. 4, by increasing the pulse length in *inp/Example1.cfg* to 40 and lowering the intensity. The spectrum is calculated as the Fourier transform of the Dipole moment $D_z(t)$ output (the Fourier transform was carried out using xmgrace). The frequency range can be increased by increasing the number of data points printed. The dipole velocity and dipole acceleration frequency components can, similarly, be plotted.

7. Benchmarking

It is useful to calculate the relative performance of the CPU to the GPU. To select between GPU and CPU, *Device* in the respective configuration files is changed to *"GPU"* to *"CPU"*. Since real world performance is of most interest, a CPU with several cores and a GPU are compared. For an example benchmark, 2 compute devices are compared: an Intel Xeon E5430 with 4 cores and 8 hardware threads operating at 2.66 GHz and an AMD 6970 GPU with 1536 execution units operating at 880 MHz. The Xeon E5430 was released at the end of 2007, and the 6970 was released at the end of 2010. The system used here has 4000 grid points per l, and the hydrogen system is subject to the pulses shown in Fig. 10 consisting of 263 698 time steps, with a 15th order Taylor propagator. The 6970 GPU runs the simulation approximately 10 times faster than the Xeon E5430 as shown in Fig. 11 on average. The exact improvement in execution time is dependent on the system, the more complex the system the better the performance would be expected to be.

While benchmarking, we are interested in the total execution time of the program with a specific configuration. Example configuration files, named *benchmark/Bench4.cfg* to *benchmark/Bench15.cfg* contain configurations for systems with 4–15 angular momenta present. The same stepsize is chosen in each system so that the work load scales linearly with l. It is expected that the CPU is at a disadvantage since it must execute the simulation, perform the standard operations of the host operating system, and also share time slices with other programs. This disadvantage should become less of an issue as the number of cores, and thus the computing power, is increased.

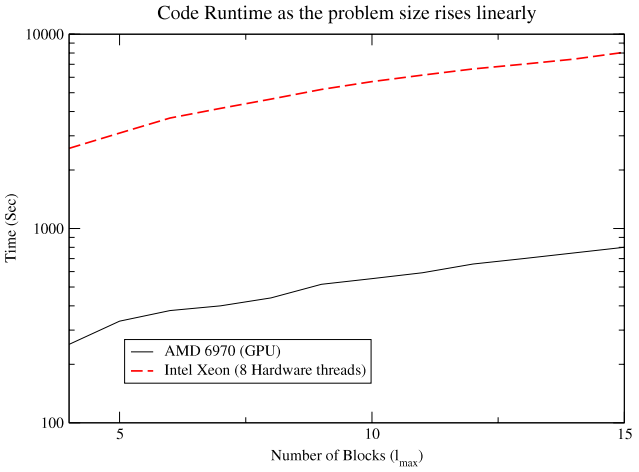

Fig. 11. A performance comparison between the OpenCL code running in parallel on an 8 hardware thread Intel Xeon (dashed red line) against an AMD 6970 (black solid line). The average reduction in runtime is 10 times. The graph shows how the runtime increases as $l_{\max}$ is increased. The y axis is on a log scale, while the x axis is the number of blocks $l_{\max}$.

8. Modifying the code

For each modification, if a new setting is added, you must add support in the configuration file, where the specific function is dependent on what group you are modifying. New configuration settings will require the addition of code to *ConfigReader.c* so as to support the reading of the setting from the configuration file. If the new setting is to be read, one may need to add a new variable to the configuration structures in *ConfigReader.h*. Reading the manual for libconfig is advised, or alternatively one can use the existing code as a template.

To add a new pulse type, three files must be changed. Firstly, the new Pulse must be added to *Pulse.c* with the same function declaration, and with the function prototype added to *Pulse.h*. The prototype for the pulses must match: *PulsePrec PulseName(RPrec, void *)*. If the pulse requires initialisation these should be handled in *PulseInit.c* using the existing pulses as a template.

Changing the structure of the Hamiltonian in the OpenCL C code in *FD.cl* does not necessarily require any change to the configuration options. Since one element of the solution vector is handled by one work item, changes can be made to the matrix–vector calculation relatively easily with little impact outside that function. The function *MatVec()* performs the matrix–vector calculation $i\dot{\Phi}(r,t) = H\Phi(r,t)$. To modify the static potential, the function *Vm()* is modified.

If any new memory objects are added to a specific propagator, they are allocated by adding lines into the structure *Objs* in *clBuildTaylor()*, *clBuildRungeKutta()* or *clBuildLanczos()*. Depending on the circumstances, changes to *ReadyMethod()* may also be required. These functions are located in *clInit.c*. If one intends to read or write data to the new memory objects during the time propagation, then changes to *Taylor.c*, *RungeKutta.c* and *Lanczos.c* may be required. A new propagator can be added by duplicating and then tailoring the host code of an existing propagator. New time dependent outputs can be added by modifying *PrintOut()* in *Output.c*.

9. Conclusions

In this paper we have presented, CLTDSE, an OpenCL based program for solving the TDSE using the grid-based finite difference approach. We have included an overview of the underlying theory used to describe and solve the hydrogen system, as well as providing information on the electric field and the vector potential. We have given a short description of the equations used within the code for the calculation of observables and other quantities. Instructions have been included on how to install and use the code. A discussion of the structure of the code has also been provided, where we have elaborated on the specifics of the implementation of the Taylor, Runge–Kutta, and Lanczos methods for OpenCL. We have also provided a benchmark to provide a flavour of the performance improvement obtained through GPGPU programming, achieving a runtime reduction of about 10 times when comparing parallel execution on a CPU and a GPU. GPU acceleration opens up the possibility to study problems that are highly computationally demanding, but can be solved in a reasonable timeframe. CLTDSE can help with this goal.

Acknowledgements

This work has been funded under a Seventh Framework Programme (FP7) project HPCAMO/256601, and with support from the Irish Research Council 'New Foundations' 2012 scheme. LAAN is supported by the Science Foundation Ireland (SFI) Stokes Lectureship Program, and Cost actions MP1203 and CM1204.

Appendix A. The configuration file settings

In the configuration file there are a number of settings to specify. These settings are expressed in terms of *Groups* and the associated *Options*. If an *Options* subgroup is not in use it will not be accessed.

Group Files. It is desirable for the location of input and output files to be fully customisable. As such the Files group allows locations for key input/output operations to be specified.

The Standard settings are:

Option	Values	Notes
osrc	Location string	The location of the OpenCL C files
Data	Location string	The location of the input data (generated by diffusion equation)
Laser	Location string	The output location for the laser
Plot	Location string	The output location for 2D plots
Radial	Location string	The output location for 1D plots of the radial probability
Time	Location string	The output location for time dependent quantities
Divisions	Integer	The approximate number of time dependent outputs that should be made

Group System. The Standard setting is:

Option	Values	Notes
Equation	"TDSE", "Diff"	Whether the equation is the TDSE or the diffusion equation (Ground state calculator)

Group Matrix. The Standard setting is:

Option	Values	Notes
Gauge	"Length", "Velocity"	The gauge for the system

Group Pulse. The Standard setting is:

Option	Values	Notes
Laser	Array of Groups	Each group specifies a separate laser pulse, or train of pulses.

The subgroup Laser has the further settings:

Option	Values	Notes
Units	"Atomic", "Standard"	The form of the laser units, atomic units or standard Wcm^{-2}, eV, fs units.
Intensity	Floating point	Intensity in the respective units
W	Floating point	Photon energy in the respective units
Shape	"SineSqr", "Gauss", "SineA", "FileE"	The shape of the pulse
SineSqr	Group	This group is required if "SineSqr" or "SineA" is selected for Shape
Gauss	Group	This group is required if "Gauss" is selected for Shape

The subgroup SineSqr has the further settings:

Option	Values	Notes
FineShape	"cos", "sin"	The fine shape of the pulse is cosine or sine
Cycles	Integer	The number of cycles in the pulse

The subgroup Gauss has the further settings:

Option	Values	Notes
TauD	Floating point	The full width half maximum of the pulse
d	Floating point	The strength of pulse chirping

Group Propagator. The Standard settings are:

Option	Values	Notes
Device	"CPU" or "GPU"	CPU or GPU execution
Method	"Taylor", "Runge–Kutta", "Lanczos"	Chooses between the different propagators
StepSize	Floating point	Indicates the initial step size in atomic units
WorkItems	Integer	The number of work items in a work group
Taylor	Group	This group is required if the Taylor method is specified
RungeKutta	Group	This group is required if the Runge–Kutta method is specified
Lanczos	Group	This group is required if the Lanczos method is specified

The subgroup Taylor has the further settings:

Option	Values	Notes
Order	Integer	The order of the Taylor propagator

The subgroup RungeKutta has the further settings:

Option	Values	Notes
Method	"RKF", "CashKarp", "ClassicRK4", "Euler", "DormandPrince", "Merson"	These settings represent a variety of standard Runge–Kutta methods

The subgroup Lanczos has the further setting:

Option	Values	Notes
Dim	Integer	The dimension of the Krylov subspace

Appendix B. Lanczos methods

The Lanczos method, assuming perfect precision, is given by [34,35]:

$Q_1 \leftarrow \frac{F(t)}{\|F(t)\|}$
$\beta \leftarrow 0$
$Q_0 \leftarrow 0$
for $p = 0$ to P **do**
  $Q_p \leftarrow \mathbf{H}Q_{p-1}$
  $\alpha_p \leftarrow Q_{p-1}^T Q_p$
  $Q_p \leftarrow F - \alpha_p Q_{p-1} - \beta Q_{p-2}$
  $\beta \leftarrow \|f_p\|$
  if $\beta = 0$ **then**
    quit
  end if
  $Q_{p+1} \leftarrow \frac{Q_{p+1}}{\beta}$
  $\tilde{H}_{p,p+1} = \tilde{H}_{p,p+1} = \beta$
  $\tilde{H}_{p,p} = \alpha$
end for

Appendix C. Runge–Kutta

A number of common Runge–Kutta integration methods have been implemented: Euler, Classic RK4, Runge–Kutta Fehlberg (*RKF 4(5)*), Dormand–Prince, Cash Karp, and Merson methods. For the classic RK4 there is only one solution, but embedded pair methods provide two solutions, one of higher order $\mathbf{F}^{(2)}$, and another solution of lower order $\mathbf{F}^{(1)}$:

$$\mathbf{k}_i = \mathbf{f}\left(t_n + c_i h, \mathbf{F}_n + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j\right)$$

$$\mathbf{F}_{n+1}^{(1)} = \mathbf{F}_n + h\sum_{i=1}^{S} b_i^{(1)}\mathbf{k}_i \tag{C.1}$$

$$\mathbf{F}_{n+1}^{(2)} = \mathbf{F}_n + h\sum_{i=1}^{S} b_i^{(2)}\mathbf{k}_i. \tag{C.2}$$

As discussed at [36], the difference between the lower order and higher order solutions,

$$\boldsymbol{\Delta} = \mathbf{F}_{n+1}^{(2)} - \mathbf{F}_{n+1}^{(1)}$$

provides an estimate of the average deviation from truncation over the entire solution vector,

$$\sigma_n \approx \frac{1}{S}|\boldsymbol{\Delta}|^2 = \frac{1}{S}\sum_{i=1}^{S}\Delta_i^2.$$

This local truncation deviation between the two methods is used to approximate the truncation between the approximate solution and the real solution, and can be used to extrapolate the value of the global truncation error. This could be used for dynamical step size support, but it is currently not implemented. At present, the sum of the local truncation error is used passively to provide the global truncation error:

$$\sigma = \sum_n \sigma_n.$$

References

[1] J.H. Posthumus, Rep. Progr. Phys. 67 (2004) 623.
[2] H. Wabnitz, A.R.B. de Castro, P. Gürtler, T. Laarmann, W. Laasch, J. Schulz, T. Möller, Phys. Rev. Lett. 94 (2005) 023001.
[3] D. Dundas, J.F. McCann, J.S. Parker, K.T. Taylor, J. Phys. B 33 (2000) 3261.
[4] L.-Y. Peng, D. Dundas, J.F. McCann, K.T. Taylor, I.D. Williams, J. Phys. B 36 (2003) L295.
[5] S. Barmaki, S. Laulan, H. Bachau, M. Ghalim, J. Phys. B 36 (2003) 817.
[6] M. Awasthi, Y.V. Vanne, A. Saenz, J. Phys. B 38 (2005) 3973.
[7] A. Palacios, S. Barmaki, H. Bachau, F. Martín, Phys. Rev. A 71 (2005) 063405.
[8] A. Palacios, H. Bachau, F. Martín, Phys. Rev. Lett. 96 (2006) 143001.
[9] J.L. Sanz-Vicario, H. Bachau, F. Martín, Phys. Rev. A 73 (2006) 033410.
[10] J. Caillat, J. Zanghellini, M. Kitzler, O. Koch, W. Kreuzer, A. Scrinzi, Phys. Rev. A 71 (2005) 012712.
[11] X. Guan, C.J. Noble, O. Zatsarinny, K. Bartschat, B.I. Schneider, Comput. Phys. Commun. 180 (2009) 2401–2409.
[12] H. Muller, Laser Phys. 9 (1999) 138–148.
[13] K.C. Kulander, K.J. Schafer, J.L. Krause, Adv. At. Mol. Opt. Phys. Suppl. 1 (1992) 247–300.
[14] D. Dundas, K.J. Meharg, J.F. McCann, K.T. Taylor, Eur. Phys. J. D. 26 (2003) 51–57.
[15] L.R. Moore, M.A. Lysaght, L.A.A. Nikolopoulos, J.S. Parker, H.W. van der Hart, K.T. Taylor, J. Modern Opt. 58 (2011) 1132–1140.
[16] L.A.A. Nikolopoulos, J.S. Parker, K.T. Taylor, Phys. Rev. A 78 (2008) 063420.
[17] L.-Y. Peng, A.F. Starace, J. Chem. Phys. 125 (2006) 154311.
[18] K.C. Kulander, Phys. Rev. A 36 (1987) 2726–2738.
[19] G. Onida, L. Reining, A. Rubio, Rev. Modern Phys. 74 (2002) 601–659.
[20] C.Ó. Broin, L.A.A. Nikolopoulos, Comput. Phys. Commun. 183 (2012) 2071–2080.
[21] D. Bauer, P. Koval, Comput. Phys. Commun. 174 (2006) 396–421.
[22] T. Birkeland, Ph.D. Thesis, University of Bergen (Dec 2009).
[23] E.S. Smyth, J.S. Parker, K.T. Taylor, Comput. Phys. Commun. 114 (1998) 1–14.
[24] J. Kobus, Comput. Phys. Commun. 184 (2013) 799–811.
[25] T. Dziubak, J. Matulewski, Comput. Phys. Commun. 183 (2012) 800–812.
[26] M. Cowlishaw (Ed.), IEEE Std 754-2008 (Revision of IEEE Std 754-1985), IEEE Inc., 2008.
[27] H. Bachau, E. Cormier, P. Decleva, J.E. Hansen, F. Martín, Rep. Progr. Phys. 64 (2001) 1815.
[28] L.A.A. Nikolopoulos, T.K. Kjeldsen, L.B. Madsen, Phys. Rev. A 76 (2007) 033402.
[29] B.H. Bransden, C.J. Joachain, Physics of Atoms and Molecules, second ed., Pearson Education Limited, 2003.
[30] H. Flocard, S.E. Koonin, M.S. Weiss, Phys. Rev. C 17 (1978) 1682–1699.
[31] A. Bandrauk, S. Chelkowski, D.J. Diestler, J. Manz, K.J. Yuan, Phys. Rev. A 79 (2009) 023403.
[32] L.A.A. Nikolopoulos, T.J. Kelly, J.T. Costello, Phys. Rev. A 84 (2011) 063419.
[33] M. Lindner, Libconfig–C/C++ Configuration File Library, Tech. Rep. URL: http://www.hyperrealm.com/libconfig/libconfig_manual.html.
[34] N. Mohankumar, S.M. Auerbach, Comput. Phys. Commun. 175 (2006) 473–481.
[35] J.W. Demmel, Applied Numerical Linear Algebra, Society for Industrial Mathematics, 1997.
[36] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes, third ed., Cambridge University Press, 2007.