Solving the Maxwell-Bloch equations for resonant nonlinear optics using GPUs

G. Demeter[a]

[a] *Wigner Research Center for Physics, Hungarian Academy of Sciences Konkoly Thege Miklós út 29-33, H-1121 Budapest, Hungary*

Abstract

We solve the Maxwell-Bloch equations of resonant nonlinear optics using GPUs and compare the computation times with traditional single- and multithreaded programs. A detailed benchmarking of programs as a function of various parameters shows how the massive parallelism built into GPUs becomes more and more advantageous as the physical problem becomes more and more demanding. For the case of multimode light propagating through an inhomogeneously broadened medium of many-level quantum systems, the program executing on GPUs can be over 20 times faster than that executing on all cores of a modern CPU. The methods presented can be applied in a wide area of atomic physics where the time evolution of atomic ensembles is to be computed.

Keywords: Pulse propagation, Maxwell-Bloch equations, resonant nonlinear optics, quantum ensembles, GPU, CUDA

1. Introduction

Interesting nonlinear phenomena that occur when resonant laser pulses propagate through atomic media have been studied extensively, more lately driven by possible applications in quantum communication and quantum computing. Perhaps the best known such phenomenon is the so-called self-induced transparency and the associated area-theorem [1, 2], but numerous other important effects have been investigated, such as electromagnetically induced transparency, slow light [3, 4], matched pulses [5, 6, 7, 8] or photon echo [9, 10], which can be used to create quantum memories designed for single photons [11].

The fundamental difficulty of this field is the determination of the coupled dynamical evolution of the atomic systems and the laser fields in space and time. The basic equations to be solved are the so-called Maxwell-Bloch equations, which arise when one considers the propagation of one or more electromagnetic fields (light pulses) through a medium composed of "atoms". In this context "atoms" are, in general, quantum systems with discrete internal states. They may be real atoms (such as those found in dilute vapors of alkali atoms), impurity ions embedded in host crystals, or quantum dots in semiconductor crystals whose excitons give rise to discrete absorption lines. The propagating fields are quasi-resonant with some of the atomic transitions and intensive enough to significantly change the atoms' internal state, whose evolution must thus be calculated by explicitly solving the Schrödinger equation. (The equations derived from the Schrödinger equation for atoms in optical fields in this context are also known as the Optical Bloch-Equations). At the same time, the medium of atoms is assumed to be dense enough to affect the pulses, whose variation must be obtained from the Maxwell equations, containing a source term due to the macroscopic polarization of the atomic medium. Because the evolution of the atomic states follows a nontrivial time dependence, the polarization term cannot be reduced to a set of linear and nonlinear susceptibilities, one has to consider explicitly the coupled dynamical system of light and atoms. These equations have been derived in a variety of settings, starting from the classical case of two-level atoms [2, 1] to more complex situations

Email address: demeter.gabor@wigner.mta.hu (G. Demeter)

[5, 6, 12]. Being a set of partial differential equations, they admit analytical solutions only is some very special cases.

In general, the solution of the Maxwell-Bloch equations is possible only numerically. Unless the simplest physical settings are considered, even the numerical solution is a laborious computational task. Sometimes it is necessary to consider the propagation of several fields simultaneously and/or use complex models for the atoms, with many quantum states. One such case is when we consider the full hyperfine level structure of alkali atoms whose dilute gases are often used as a medium for resonant nonlinear optical experiments. Another case is when impurity ions in dielectric host crystals are considered, where electronic transitions are coupled to the vibrational motion of the ion in the crystal lattice (the so-called localized vibrational modes) [13, 14]. A further complication arises when the medium is composed of atoms whose transition lines are inhomogeneously broadened, i.e. whose transition frequencies are not identical, but are distributed randomly around an average according to some probability distribution function. This is inevitably the case in solid state media such as ion-doped crystals or quantum dots and requires the explicit solution of the Shrödinger equation for an ensemble of atoms to obtain the macroscopic polarization of the medium. In all these cases, solving the equations numerically is a demanding problem.

Graphical processing units (GPUs) have proven to be very useful in solving various numerical problems in recent years [15, 16, 17, 18]. The massive parallelism that GPUs offer can be very advantageous also when solving demanding problems in resonant nonlinear optics, most obviously when the response of the medium is to be determined by using a statistical ensemble of atoms. We have tested the extent to which GPUs may be useful for these calculations by programming, running and benchmarking three implementations of the same basic calculation. A single-thread version, a multithreaded version coded using OpenMP and executing on all four cores of a modern CPU, and finally a massively parallel version utilizing GPUs coded with the CUDA technology of Nvidia.

This paper is organized as follows. The formulation of the physical problem and the equations to be solved are presented in section 2. The general computation scheme is presented in section 3. The peculiarities of the various implementations of one particular step, solving the Optical Bloch Equations for an ensemble of atoms are described in section 4. In the present case, that is by far the most time-consuming step of the integration. The results are presented and discussed in section 5 and summarized in 6.

2. Formulation of the problem

To be specific, we consider quantum systems that can be used to model rare-earth-ion dopants in crystals, with vibrational motion coupled to an electronic transition [12]. They have N_v ground state sublevels $|g,m\rangle$, and the same number of excited state sublevels $|e,n\rangle$ ($m,n \in \{0,\ldots,N_v-1\}$) with a dipole transition between any of the ground and excited states with transition element $\langle e,n|\hat{d}|g,m\rangle = d_{eg}\mathcal{F}_{n,m}$ (here $\mathcal{F}_{n,m}$ is a Franck-Condon factor coming from vibrational wave-function overlap). The basic transition frequency between $|g,0\rangle \rightarrow |e,0\rangle$ is denoted by ω_{eg} and is assumed to be inhomogeneously broadened. The level spacing of the ground and excited state sublevels is taken to be the same for each atom $\nu = \nu_g = \nu_e$.

We consider light composed of equidistant quasi-monochromatic modes with frequency spacing $\delta = \nu$ propagating through the medium. Restricting ourselves to a single spatial dimension z, the electric field can be written as

$$E(z,t') = \frac{1}{2}\sum_l \varepsilon_l(z,t')e^{-i(\omega_0+l\cdot\delta)t'+ik_l z} + c.c. \quad (1)$$

The complex mode amplitudes ε_l vary little on the length scale of k_l^{-1} and timescale of ω_0^{-1}, and $\delta \ll \omega_0$ is assumed. For convenience we introduce the retarded time $t = t' - z/c$, which is equivalent to using a reference frame that moves with the speed of light across the medium. In the slowly varying envelope approximation the wave equation for the mode amplitudes will then become [12]:

$$\frac{\partial}{\partial z}\varepsilon_l(z,t) = i\frac{k_0}{2\epsilon_0\epsilon}\mathcal{P}_l(z,t), \quad l \in \{l_{min},\ldots,l_{max}\} \quad (2)$$

where we have made use of $k_l \approx k_0$ that follows from previous considerations. The source terms on the right-handside $\mathcal{P}_l(z,t)$ are the macroscopic polarizations due to the atoms excited by the fields.

The state of an atom at any z and t is given by $2N_v$ complex probability amplitudes contained in the vector $\underline{\psi}$, whose time evolution is governed by the Hamiltonian $\hat{H}$ via the Schrödinger equation:

$$\partial_t\underline{\psi}(\Delta;z,t) = -i\hat{H}(\Delta;z,t)\underline{\psi}(\Delta;z,t) \quad (3)$$

where $\Delta = \omega_0 - \omega_{eg}$ is the detuning between the frequency of the $l = 0$ mode and the basic resonance. We have an inhomogeneously broadened atomic ensemble, so Δ is slightly different for each atom. The inhomogeneous line shape function $g(\Delta)$ describes the distribution of atoms with respect to Δ. It is typically a symmetric function peaked around 0, e.g. a Gaussian or a Lorentzian. Introducing the notation $\underline{\psi}(\Delta; z, t) = \{a_m(\Delta; z, t), b_n(\Delta; z, t)\}$ for the ground and excited state amplitudes separately, we can write 3 (after some convenient phase transformations) in the form known as Optical Bloch-Equations (OBE):

$$\begin{aligned}\partial_t a_m(\Delta; z, t) &= i(m(\kappa_g - \delta) + \Delta)a_m(\Delta; z, t) \\ &\quad + i\frac{d_{eg}}{2\hbar}\sum_l \varepsilon_l^*(z, t)\mathcal{F}_{m,m+l}b_{m+l}(\Delta; z, t) \\ \partial_t b_n(\Delta; z, t) &= in(\kappa_e - \delta)b_n(\Delta; z, t) \\ &\quad + i\frac{d_{eg}}{2\hbar}\sum_l \varepsilon_l(z, t)\mathcal{F}_{n-l,n}a_{n-l}(\Delta; z, t)\end{aligned} \tag{4}$$

Clearly, the Hamiltonian matrix is of the form $\hat{H}(\Delta; z, t) = \hat{H}_A(\Delta) + \hat{H}_I(z, t)$ where $\hat{H}_A$ is a diagonal matrix independent of z, t. The interaction part $\hat{H}_I$ is independent of Δ, being composed of nondiagonal blocks that depend on z and t through the fields and connect the ground and excited state amplitudes:

$$\hat{H}_I(z, t) = \begin{pmatrix} 0 & \hat{\Omega}^\dagger(z, t) \\ \hat{\Omega}(z, t) & 0 \end{pmatrix} \tag{5}$$

The Bloch part 4 of the equations is connected to the Maxwell part 2 through the macroscopic polarization, which is calculated from the coherences between the ground and excited state sublevels and averaged over the inhomogeneous distribution ($\mathcal{N}$ is the spatial density of atoms):

$$\begin{aligned}&\mathcal{P}_l(z, t) = 2\mathcal{N}d_{eg} \\ &\sum_m \mathcal{F}_{m,m+l}\int a_m^*(\Delta; z, t)b_{m+l}(\Delta; z, t)g(\Delta)d\Delta\end{aligned} \tag{6}$$

The problem is thus to find the complex functions $\varepsilon_l(z, t)$ by solving Eqs. 2 for a set of N_F fields. The typical boundary condition used corresponds to one or two laser pulses entering the medium at $z = 0$, that is, $\varepsilon_l(0, t) = f_l(t - t_0)$, where f_l is a real function peaked around 0 for, say, $l \in \{0, 1\}$ and $\varepsilon_l(0, t) = 0$ for any other l. It is of course insufficient to consider only the two modes that enter the medium, as the nonlinear medium can generate new modes (Raman-sideband generation). Typical initial conditions for the atoms would be $a_0(\Delta; z, 0) = 1$, $a_m(\Delta; z, 0) = 0$ for $m \neq 0$ and $b_n(\Delta; z, 0) = 0$ corresponding to atoms in their lowest vibrational ground states initially. This form of the equations is more general than that usually treated in several respects. First, we allow a large number of ground and excited states (we have worked with up to $N_v = 32$). Second, we consider a relatively large number of field modes and allow each field to interact with multiple atomic transitions.

3. Computation scheme

To study propagation phenomena, we need to solve the Maxwell-Bloch equations 2 and 4 on the domain $z \in [0, z_{max}]$, $t \in [0, t_{max}]$ with resolutions h_z and h_t respectively, i.e. on the two-dimensional space-time grid $z_j = j \cdot h_z$, $j \in \{0, ..., N_z\}$, $t_k = k \cdot h_t$, $k \in \{0, ..., N_t\}$, $N_z = z_{max}/h_z + 1$, $N_t = t_{max}/h_t + 1$. To calculate the corresponding values of the slowly varying complex field envelopes $\varepsilon_{j,k}^l$ for the N_F modes of the set $l \in \{l_{min}, \ldots, l_{max}\}$, we need the macroscopic polarization of the medium for each mode $\mathcal{P}_{j,k}^l$. This in turn must be obtained from the corresponding values of the state vectors $\underline{\psi}_{j,k}^D$ for all the atoms $D \in \{0, \ldots, N_\Delta - 1\}$ in the ensemble. Adding the boundary conditions for the fields $\varepsilon_{0,k}^l = f_k^l$ and the initial condition for the atomic states $\underline{\psi}_{j,0}^D = \underline{A}$ completes the formulation of the problem.

A simple and robust method of solution is to use finite difference schemes as sketched by the pseudocode for the general structure of the program (algorithm 1). For each step in space we first iterate over the points in the time domain (lines 2 to 7) to advance all atomic states using a simple fourth-order Runge-Kutta scheme (line 3 in algorithm 1, function further detailed in algorithm 2). Then we calculate the polarization for each mode by averaging over all atoms in the ensemble (lines 4 to 6, here $\beta = \mathcal{N}d_{eg}k_0/\epsilon_0\epsilon$). The fields are advanced using a second-order Adams-Bashforth scheme to the next spatial location (lines 8 to 12). With the fields known at $j + 1$, the matrices $\hat{H}_{j+1,k}$ can be constructed to be used in the next iteration. The constraint of probability conservation $|\underline{\psi}_{j,k}^D|^2 = 1$

can be used conveniently to monitor the solution's accuracy.

Algorithm 1 Global calculation scheme.

1: **for** $j = 1$ **to** $N_z - 1$ **do**
2: **for** $k = 1$ **to** N_t **do**
3: $\underline{\psi}^D_{j,k} = RK4(\underline{\psi}^D_{j,k-1}, \hat{H}^D_{j,k}, \hat{H}^D_{j,k-1})$
4: **for** $l = l_{min}$ **to** l_{max} **do**
5: $\mathcal{P}^l_{j,k} = \sum_{D,m} \beta g^D \mathcal{F}_{m,m+l} a^D_{m,k} b^D_{m+l,k}$
6: **end for**
7: **end for**
8: **for** $k = 1$ **to** N_t **do**
9: **for** $l = l_{min}$ **to** l_{max} **do**
10: $\varepsilon^l_{j+1,k} = \varepsilon^l_{j,k} + \frac{3}{2} i h_z \mathcal{P}^l_{j,k} - \frac{1}{2} i h_z \mathcal{P}^l_{j-1,k}$
11: **end for**
12: **end for**
13: **end for**

The pseudo-code shown here is just the bulk of the program, miscellaneous tasks like starting up, saving results or calculating state vector norms for monitoring numerical accuracy have been omitted. Note that we use mathematical notation for brevity in the algorithms wherever this does not hamper comprehension. The concise listings may thus "hide" additional nested loops, for example in the Runge-Kutta part (algorithm 2), where $\underline{\psi}^D_k$, $\underline{k}^D_p$ are vectors and $\hat{H}^D_k$ are matrices with complex elements. Also not detailed here is the fact that because of the structure of the Hamiltonian (Eq. 5), the matrix-vector multiplications are in fact performed as multiplications by the diagonal elements and the nondiagonal blocks $\hat{\Omega}$ separately to avoid multiplying with a large number of zero elements. As the Adams-Bashforth scheme requires the use of two spatial points to calculate the fields at the next one, the program is started up using a first-order Euler method with a smaller spatial step.

Algorithm 2 The RK4 function. The index j of the variables has been suppressed for clarity.

Require: $\underline{\psi}^D_{k-1}$,$\hat{H}^D_k$,$\hat{H}^D_{k-1}$
1: **for** $D = 0$ **to** $N_\Delta - 1$ **do**
2: $\underline{k}^D_1 = -i h_t \hat{H}^D_{k-1} \cdot \underline{\psi}^D_{k-1}$
3: $\underline{k}^D_2 = -i h_t \frac{1}{2} (\hat{H}^D_{k-1} + \hat{H}^D_k) \cdot (\underline{\psi}^D_{k-1} + \frac{1}{2} \underline{k}^D_1)$
4: $\underline{k}^D_3 = -i h_t \frac{1}{2} (\hat{H}^D_{k-1} + \hat{H}^D_k) \cdot (\underline{\psi}^D_{k-1} + \frac{1}{2} \underline{k}^D_2)$
5: $\underline{k}^D_4 = -i h_t \hat{H}^D_k \cdot (\underline{\psi}^D_{k-1} + \underline{k}^D_3)$
6: $\underline{\psi}^D_k = \underline{\psi}^D_{k-1} + \frac{1}{6} (\underline{k}^D_1 + 2\underline{k}^D_2 + 2\underline{k}^D_3 + \underline{k}^D_4)$
7: **end for**

The above two pseudocodes show that the main parts of the calculation can be readily parallelized. The iterations over the atoms of the ensemble at line 3 (detailed in algorithm 2) are completely independent. The iteration to calculate polarization modes (lines 4 to 6) can also be calculated independently for each mode, just as the iteration to advance the fields to the next point in space (lines 8 to 12). While all possibilities to parallelize calculations for speeding them up were exploited in both OpenMP and CUDA implementations, in this paper we will detail only the various implementations of the Runge-Kutta function 2. This can be justified by the fact that for real production runs (i.e. a lot of atomic sublevels, field modes and many atoms) it is this part that takes up by far the most time. Profiling the GPU code using Nvidia's Compute Visual Profiler shows that depending on input parameters, the Runge-Kutta function consumes between 60-75% of the overall execution time of the program, so it is the coding of this function that has by far the greatest impact on the overall performance. The next most laborious part is the computation of the polarization (lines 4-6 in algorithm 1) which consumes about half of the remaining execution time. Note that the computation times shown and compared in section 5 are the overall execution times for the whole program.

In the present approach, we advance all the atoms one timestep, calculate the polarizations, and then proceed to the next timestep as we can then reuse the memory allocated to the atomic variables at every step. One can also exchange the iterations to calculate the whole time evolution of a single atom before calculating another one. However, if the full time evolution of all atomic variables is calculated and stored before computing the polarizations, orders of magnitude more memory is needed. Alternatively, one can also calculate the full time evolution of a single atom, increment the polarizations with its contribution and then proceed to the next atom, again reusing the memory. But when atoms are calculated in parallel in this approach, possible race conditions must be dealt with.

The size of the most time consuming part of the computation, calculating the atomic evolution and the polarization is clearly proportional to N_Δ (algorithm 2 and line 5 in algorithm 1). The dependence of the computational complexity on the other two important parameters, N_v and N_F is not so trivial. Theoretically the size of the nondiagonal blocks $\hat{\Omega}$ of the Hamiltonian matrix appearing in the RK4 function is proportional to N_v^2, but if N_F

is low, the blocks will have a lot of zero elements. Thus increasing N_v on its own does not mean that the computational complexity will grow quadratically. On the other hand, increasing N_F, while it itself appears only linearly in the equations, also increases the problem size by populating $\hat{\Omega}$ and $\hat{\Omega}^\dagger$ with nonzero elements.

4. Program implementation

Algorithm 1 was implemented in three different programs. The simplest one was a sequential C++ code, compiled to execute on a single CPU thread. The second program was very similar, but used the OpenMP language extension to run on multiple CPU threads simultaneously. The third program used the C++ language extension of Nvidia's CUDA™ technology to perform the calculations on one of the test system's GPUs. In all three cases the -O3 optimization level was used by the compiler and the execution of the test problems was timed using the ftime() function call.

For each program, execution time has been investigated extensively as a function of input parameters. To this end, three test problems were used, which differed only in the number of fields whose evolution we followed. The first problem, $N_F = 2$ was a relatively simple task, often investigated although with much simpler atomic systems. The second and third problems, $N_F = 9$ and $N_F = 19$ were the problems considered in [12]. Each of the problems were investigated with the same several values of N_v and the same (large) range of N_Δ. Because the number of fields determines just how many nonzero elements the offdiagonal blocks $\hat{\Omega}, \hat{\Omega}^\dagger$ of the Hamiltonian have, it affects the work to be done a great deal. Obviously, the more modes we consider, the more distant coupling we obtain in terms of vibrational levels. This means that for more fields to calculate, we should use more vibrational levels as well, and vice versa. Therefore when using exactly the same set of possible N_v values for the problems, some cases were in fact "unphysical", but were retained all the same for a good comparison of the performance. The speedup, defined as the ratio of execution times for identical input parameters $S = T_{single}/T_{OpenMP}$ or $S = T_{OpenMP}/T_{GPU}$ has been established in all cases. The conclusions presented in section 5 were drawn from a large number comparisons, even though only a fairly small, representative set of results have been plotted to illustrate the results.

4.1. The test system

The same, fairly current tabletop PC was used to execute the programs in each case, with the most important elements of the hardware configuration being as follows:

- CPU: Intel®Core™i7 870, 2.93GHz (Nehalem micro architecture, 45 nm)
- Main memory: 8 GB of DDR3, 1333 MHz
- GPU1: Geforce® GTX580 with 1536 MB GDDR5, compute capability 2.0 (Fermi) and 1.54 GHz GPU Clock speed, 16 SM x 32 CUDA cores
- GPU2: Geforce® GTX460 with 1024 MB GDDR5, compute capability 2.1 (Fermi) and 1.45 GHz GPU Clock speed, 7 SM x 48 CUDA cores

The OS of the test system was openSUSE 11.2, 64 bit Linux with the most important software components as follows: kernel version: 2.6.31.5, C++ compiler: gcc (SUSE Linux) 4.4.1, glibc version: glibc-2.10.1, MPI implementation: openmpi-1.2.8, CUDA runtime environment 4.0. The test system was dedicated to the purpose of the calculations, it had no graphical login environment running (booted to runlevel 3), and had no user load apart from a single distant login shell used to start the calculations. This way a reliable benchmark could be established, with the execution times of the programs differing much less than one percent between repeated runs in all but a very few cases.

4.2. Single-thread execution

Our program does not use any implicitly multithreaded library routines (such as NAG for linear algebra), so the simple C++ code executes on a singe CPU core at all times.

4.3. Parallel execution using OpenMP

OpenMP is an application program interface for writing programs that employ multi-threaded, shared memory parallelism. Compiler directives, inserted in the C++ code, direct the forking and rejoining of threads explicitly. In our case, the number of threads that the program forks for parallel regions was defined at compile time to be either four (the number of physical CPU cores) or eight (the thread number that the OS could schedule simultaneously on the CPU due to Hyper-threading)

and the dynamic alteration of the thread number by the runtime was disabled. We have also found it to be advantageous to define the thread affinity by hand (via an environment variable), i.e. to bind threads to specific cores explicitly. This prevents thread migration operated by the OS kernel. Allowing the kernel to schedule threads dynamically on the cores results in a considerable fluctuation of the execution times and, in general, greater execution times, probably due to a high number of L2 cache misses. This is similar to the behavior found in [19]. However, this strategy is certain to be advantageous only for systems that are dedicated to run the computation.

The OpenMP implementation of the RK4 routine is shown by the pseudocode in algorithm 3. The threads are forked at line 1 and the subsequent work sharing constructs **#pragma omp for** cause the instructions within the loops to be executed by the threads as a team. Each thread performs the multiplication for different atoms (different value of the loop variable D). Clauses following the directive define how the threads are handed blocks of the iteration to complete. Because of the simplicity of this iteration, (a single, fixed-size matrix-vector multiplication) a static scheduling was found to be fastest, when one fourth(eighth) of the work is handed in a contiguous block to each of the four(eight) threads. In more complicated cases (when the instruction count of iterations varies for example), the work can be scheduled dynamically.

This function can also be implemented with a single **for** loop over D that calculates the updated state of an atom all at once (see algorithm 2), but this method was found to be somewhat slower. The reason is, that there is an implied barrier at the end of each **for** loop where the threads synchronize. With the listing shown here this synchronization causes all the threads to work with the same one of the four variables, $\{\underline{k}_1, \ldots, \underline{k}_4\}$ at any one time, decreasing the chance of cache misses.

Algorithm 3 RK4 function with OpenMP

Require: $\underline{\psi}^D_{k-1}$,$\hat{H}^D_k$,$\hat{H}^D_{k-1}$

1: **#pragma omp parallel**. . . //fork team of threads
2: {
3: **#pragma omp for** // share work within team
4: **for** $D = 0$ **to** $N_\Delta - 1$ **do**
5: $\underline{k}^D_1 = -ih_t \hat{H}^D_{k-1} \underline{\psi}^D_{k-1}$
6: **#pragma omp for**
7: **for** $D = 0$ **to** $N_\Delta - 1$ **do**
8: $\underline{k}^D_2 = -ih_t \frac{1}{2}(\hat{H}^D_{k-1} + \hat{H}^D_k)(\underline{\psi}^D_{k-1} + \frac{1}{2}\underline{k}^D_1)$
9: **#pragma omp for**
10: **for** $D = 0$ **to** $N_\Delta - 1$ **do**
11: $\underline{k}^D_3 = -ih_t \frac{1}{2}(\hat{H}^D_{k-1} + \hat{H}^D_k)(\underline{\psi}^D_{k-1} + \frac{1}{2}\underline{k}^D_2)$
12: **#pragma omp for**
13: **for** $D = 0$ **to** $N_\Delta - 1$ **do**
14: $\underline{k}^D_4 = -ih_t \hat{H}^D_k(\underline{\psi}^D_{k-1} + \underline{k}^D_3)$
15: **#pragma omp for**
16: **for** $D = 0$ **to** $N_\Delta - 1$ **do**
17: $\underline{\psi}^D_k = \underline{\psi}^D_{k-1} + \frac{1}{6}(\underline{k}^D_1 + 2\underline{k}^D_2 + 2\underline{k}^D_3 + \underline{k}^D_4)$
18: }

4.4. Parallel execution using CUDA - general considerations

GPUs are powerful parallel computing machines that support the SIMD (single instruction multiple data) paradigm and can execute certain computations much faster than traditional CPUs. Numerous programing tools have appeared recently that facilitate GPU program implementation for scientific computations, while the architecture also evolved to implement floating point operations, both with single and double precision. One convenient tool is the CUDA™ technology, which is a general purpose parallel computing architecture from Nvidia. It provides a set of extensions to the C++ programming language for computations to be executed on Nvidia graphics cards. The package includes tools to compile, debug, and profile programs [20, 21, 22]. Complex numbers essential in quantum theory can be (and in our case were) implemented in the GPU code using a templated complex number type based on the native CUDA types float2 and double2 [23].

The basic CUDA program consists of a single CPU thread (host thread) that initiates a series of "kernel launches", i.e. issues special function calls to create a multitude of parallel threads executing on the GPU (the "device" in CUDA terminology). The GPU can only operate on data in device memory, so the necessary data structures must first be created and initialized on the device.

The GPU is built up of a number of "streaming multiprocessors" (SMs), each of which is composed of individual "CUDA cores" (stream processors). This division in the structure is very important, as cores of a single SM share a fast cache (shared memory). A kernel launch must define the kernel's geometry corresponding to this hardware structure - the multitude of threads are ordered into three-dimensional blocks, and the blocks into a three-

dimensional grid. Threads of a single block always execute on the same SM. They can cooperate via the fast shared memory of the SM and their execution can be synchronized. A single SM will always execute threads of a single block at any one time, grouped into "warps" of 32 threads. Different thread blocks must be independent, their order of execution must be irrelevant. They can not be synchronized and can cooperate only via much slower global memory. The three-dimensional indexing of threads and blocks is convenient when invoking vector and matrix calculations.

Choosing the correct kernel geometry is the most important factor in optimal task distribution over the GPU. The number of threads in a block should preferably be a multiple of a SM's number of cores and threads that require synchronization must be on the same block. The number of blocks should be at least as many as the GPU's SM number, but usually many more. Up to eight blocks may be active on a SM with 48 warps altogether if SM resources permit. The term SM "occupancy" refers to the ratio of active warps to the maximum number of warps supported per SM of the device. High occupancy is important because it helps hide memory latencies in global memory transactions. The GPU can switch threads very easily, so if lots of warps are active on a SM, warps with threads waiting for memory transactions can be dumped and executed later when the required data is cached. Kernel occupancy can be calculated and kernel geometry optimized using information returned by the compiler. However, while low occupancy usually means low performance, higher occupancy does not necessarily mean greater execution speed.

4.5. Parallel execution using CUDA - the RK4 function

The general algorithm looks very similar when written in CUDA C (algorithm 4) except lines 3, 4-6 and 8-12 in algorithm 1 are substituted by kernels (lines 3, 5 and 8 in algorithm 4 respectively) that launch threads on the GPU to do the computations. The general format as illustrated on line 3 is similar to a function call with the geometry defined in two sets of three integers, *grid1* and *block1*. Different kernels are generally launched with different geometries ($grid2$, $block2$ and $grid3$, $block3$). Additionaly, results may be transfered back to the host at line 7 to be saved if necessary - this part is typically executed at every 20th step, which also implies that most of the data will exist only on the device and not in the host memory. The only time data is transfered from the host to the device is at startup when the boundary and initial conditions are initialized on the device.

Algorithm 4 general algorithm,CUDA implementation

1: **for** $j = 1$ **to** N_z **do**
2: **for** $k = 1$ **to** N_t **do**
3: $\underline{\psi}_k^D = RK4 \lll grid1; block1 \ggg (\ldots)$
4:
5: $\mathcal{P}_{j,k}^l = CalcPol \lll grid2; block2 \ggg (\ldots)$
6: **end for**
7: // transfer $\underline{\psi}_k^D$ and $\varepsilon_{j,k}^l$ to host if necessary
8: $\varepsilon_{j+1,k}^l = Adams \lll grid3; block3 \ggg (\ldots)$
9: **end for**

For the RK4 routine, it is most natural to choose the threads calculating the evolution of a single atom to be in one block (each block calculates a different atom). The geometry of the kernel launch was chosen to be $grid1 = (N_\Delta, 1, 1)$, $block1 = (N_v/M, N_v, 1)$, the block index enumerating the atoms. In each block, N_v^2/M threads cooperate as detailed in algorithm 5. The multiplication of the state vector by $\hat{\Omega}^\dagger$ and $\hat{\Omega}$ is accomplished in chunks of N_v rows and N_v/M columns at a time (lines 3 to 9), with each thread performing $2M$ multiplications. The results are accumulated in a $2N_v \times N_v/M$ array C that is stored in the SM's cache. Then a reduction is calculated at lines 10 to 15 with less and less threads taking part in each step. The synchronization of the threads after the multiplications (line 9) and after each step of the reduction (line 15) are essential, that is why this calculation cannot be spread over more blocks. Storing the results of each threads' part of the work in shared memory is also essential, as during the reduction some threads access the results calculated by others. The element with the diagonal part of the Hamiltonian is added only at the end (line 17) when $\underline{k}_1$ is stored. All the $\underline{k}_i$ are calculated in a similar manner and are then summed to obtain the new state vector.

The parameter M makes it possible to tune the size of a block, its optimal choice is an important question. For small N_v, $M = 1$ is clearly best, as a block with less threads than CUDA cores per SM cannot make full use of the SM. On the other hand, if N_v^2 grows too large, M can be used to decrease the thread number and shared memory usage to be within the SM's physical limits. (Choosing at

Algorithm 5 RK4 kernel, CUDA implementation

Require: D // position within grid, $D \in \{0, N_\Delta - 1\}$
Require: n, m // position within block, $n \in \{0 \dots N_v/M - 1\}$, $m \in \{0 \dots N_v - 1\}$
Require: $\hat{\Omega}, \hat{\Omega}'$ //the nondiagonal blocks of the Hamiltonian for two consecutive time points
Require: $\hat{E}^D$ //the vector of the diagonal elements of the Hamiltonian
Require: $\underline{\psi}^D$ // the current state vector
1: declare vectors $\underline{k}_1, \underline{k}_2, \underline{k}_3, \underline{k}_4$ in shared memory
2: declare C, a $2N_v$ by N_v/M matrix in shared memory
3: $C_{m,n} = \hat{\Omega}^\dagger_{m,n} \cdot \psi^D_{n+N_v}$
4: **for** $j = 1$ **to** $M - 1$ **do**
5: $\quad C_{m,n} = C_{m,n} + \hat{\Omega}^\dagger_{m,n+jN_v/M} \cdot \psi^D_{n+jN_v/M+N_v}$
6: $C_{m+N_v,n} = \hat{\Omega}_{m,n} \cdot \psi^D_n$
7: **for** $j = 1$ **to** $M - 1$ **do**
8: $\quad C_{m+N_v,n} = C_{m,n} + \hat{\Omega}_{m,n+jN_v/M} \cdot \psi^D_{n+jN_v/M}$
9: syncthreads()
10: $q = N_v/2M$
11: **while** $q! = 0$
12: $\quad$ **if** $n < q$
13: $\quad\quad C_{m,n} = C_{m,n} + C_{m,n+q}$
14: $\quad q = q/2$
15: $\quad$ syncthreads()
16: **if** $n = 0$ **then**
17: $\quad k_{1,m} = ih_t(E^D_m \psi^D_m - C_{m,0})$ // $\underline{k}_1$ has been calculated
// next to calculate $\underline{k}_2$, $\underline{k}_3$, $\underline{k}_4$ in a similar manner
18: $C_{m,n} = 1/2 \cdot (\hat{\Omega}^\dagger_{m,n} + \hat{\Omega}^{\dagger\prime}_{m,n}) \cdot (\psi^D_{n+N_v} + 1/2 \cdot k_{1,n+N_v})$ // etc...
19: $\quad \vdots$
20: **if** $n = 0$ **then**
21: $\quad \psi^{D,new}_m = \psi^D_m + \frac{1}{6}(k_{1,m} + 2k_{2,m} + 2k_{3,m} + k_{4,m})$

most N_v^2 threads to calculate the multiplication by a $2N_v \times 2N_v$ matrix is justified by the fact that only the nondiagonal $N_v \times N_v$ blocks $\hat{\Omega}, \hat{\Omega}^\dagger$ have elements outside the diagonal). The ideal choice of M has been found by experimenting and is always given together with N_v when discussing results in section 5.

In a way, a single thread of the RK4 function's OpenMP implementation corresponds to block of threads of the CUDA implementation, in that each of these units does the calculation for a single atom. While the CPU executes four threads on four cores at any one time, the GPU executes as many blocks as there are SM's on the device (16 for GTX580 and 7 for GTX460).

4.6. Some final remarks

The other kernels of algorithm 4 (at lines 5 and 8) have also been optimized carefully for the even distribution of the computational load on the SMs of the GPU.

There are a number of further points when considering the usage of GPUs for a computation. One is the question of moving data between host and device for example, which may result in a considerable extra overhead over the time required purely for the calculation. In our case however, this overhead is completely negligible, as the data is "created" on the device by iterating the propagation routine, and only a small fraction is transferred to the host to be saved at certain timesteps. Furthermore, the data transfer can be executed asynchronously, so it does not delay the execution of the subsequent kernels.

One important property of GPUs is the great memory bandwidth and specialized memory access patterns that allow accelerated data transfer very important in some calculations. This property is not exploited by the current problem. In fact, profiling the most laborious part, the RK4 routine shows that it is very heavily compute bound for the physically interesting range of parameters.

5. Results and discussion

5.1. Speedup from single-thread to CPU multithread

Analysis of the execution times for the test problems shows that the speedup accomplished by using multiple threads on the CPU compared to single-thread program execution depends considerably on the parameters that define the problem size N_F, N_v and N_Δ. For small N_Δ, single thread execution is faster (not surprisingly), but as N_Δ increased, the speedup grows fast and attains more or less constant values between 1.9-3.5 for four CPU threads and between 2.2-4.0 for eight CPU threads. The speedup gained with eight threads is consistently higher than that gained with four threads, but it usually does not reach 4 even though the program executes on all four physical cores. Memory bandwidth may have something to do with this, as well as the fact that this processor has the ability to dynamically raise the frequency that its cores operate at (Turbo Boost technology) until thermal limits permit. When only a single core is in heavy use, its operating frequency can be substantially higher than when all four of them are fully utilized.

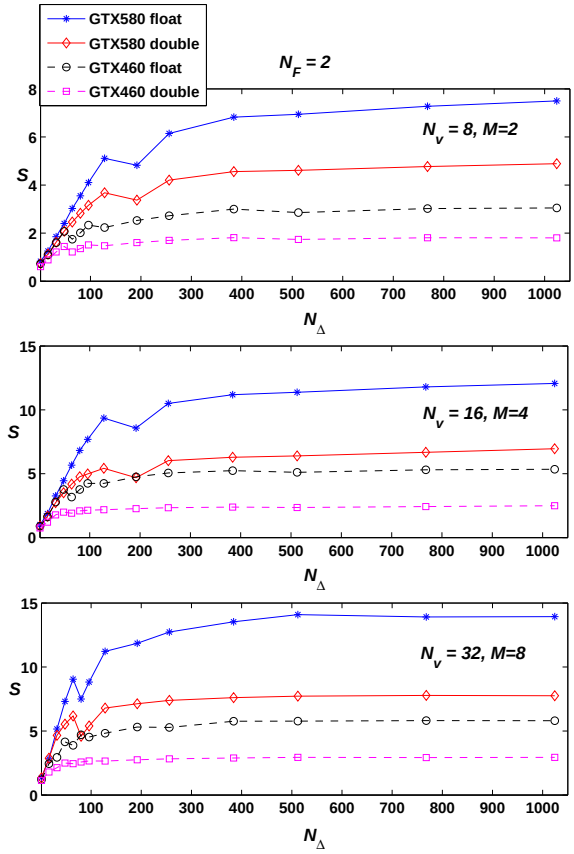

Figure 1: Speedup of the GPU program compared to the OpenMP implementation executing on 8 threads for the first test problem ($N_F = 2$), for various values of N_v as a function of N_Δ.

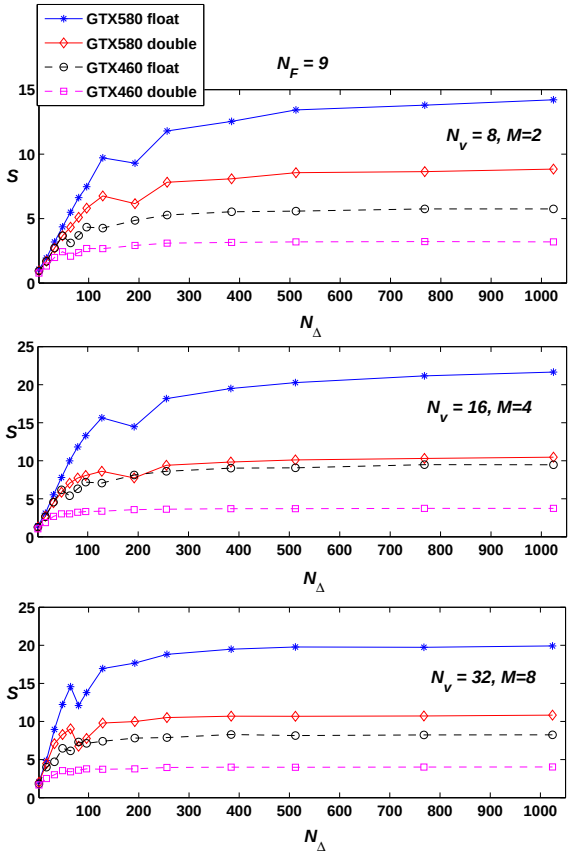

Figure 2: Speedup of the GPU program compared to the OpenMP implementation executing on 8 threads for the second test problem ($N_F = 9$), for various values of N_v as a function of N_Δ.

5.2. Speedup from CPU multithread to GPU

To assess the performance gain achieved by using GPU programs, we calculated and plotted the speedup for a wide range of parameters for all three test problems. Because the eight-thread execution on the CPU was always somewhat faster than four-thread one, in what follows, we compare the performance of the GPU codes exclusively against that case i.e. from now on speedup means $S = T_{OpenMP8thread}/T_{GPU}$. In addition, for the CPU implementations we always use double precision variables as using only float precision does not increase the execution speed. For the GPU implementation on the other hand, there is a considerable speed difference between float and double precision, it is thus important to determine whether float precision is sufficient to obtain valid physical results in our case. Unfortunately, there are no analytical solutions that we can compare our numerical results to, so after having experimented with the CPU program to determine stepsize parameters that yield acceptable results, we have compared the output of the GPU program to that of the CPU program. As expected, when using double precision on the GPU, the output of the two programs is identical practically within the precision of the double variable type. When using only float precision, the results of the GPU program still agree to that of the CPU implementation within the precision of the float variable type. This means that the result is still very reliable from a physical viewpoint, as the dominant fields and probability amplitudes calculated by the GPU code are equal to their CPU counterparts within a relative error below 10^{-6}. The relative error for field and probability amplitudes that are much smaller than the dominant ones (high order

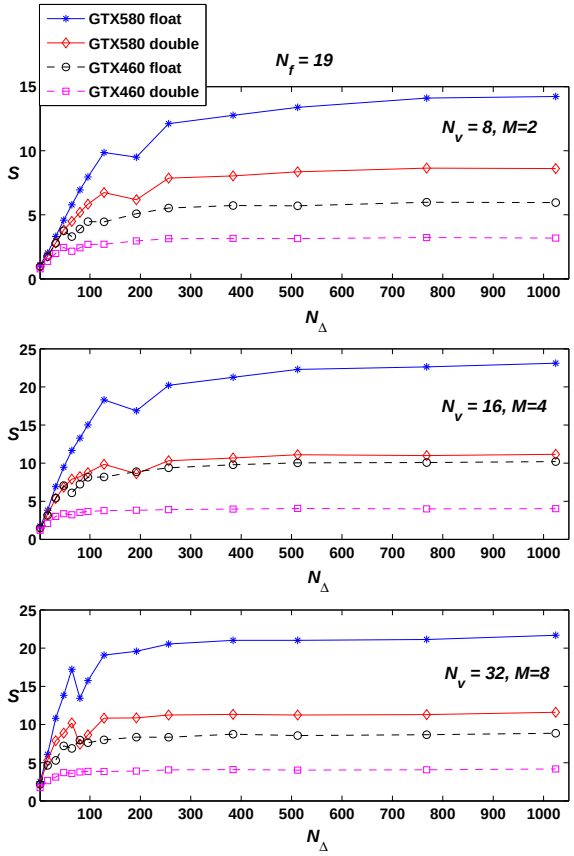

Figure 3: Speedup of the GPU program compared to the OpenMP implementation executing on 8 threads for the third test problem ($N_F = 19$), for various values of N_v as a function of N_Δ.

Raman sideband modes and high order vibrational states of the ions) is substantially higher. This however is acceptable, as physically significant quantities such as atomic level probabilities and field mode energies are proportional to the square of probability and field amplitudes, so the effects of these levels and field modes are negligible. We can thus conclude that maintaining float precision is usually quite sufficient in the calculations to obtain valid physical results in our case. Nevertheless, for the benefit of possible applications that require double precision for valid results, we compare both the float and the double precision performance of the GPU to that of the CPU. We compare the performance of the two GPUs in the system against that of the OpenMP program separately.

The primary results of our investigations are shown in Figs 1,2 and 3 which depict the speedup for both GPUs, for float and double precision calculations as a function of N_Δ. The times used to calculate the speedup are the times required for the entire time program in every case. The curves are plotted for each of the three test problems and for several values of N_v. In each case the execution time with the optimal value of the parameter M has been shown. From the figures it is clear that the speedup always increases to attain a constant value as N_Δ is increased, and for both GPUs using only float precision is roughly twice as fast as using double precision. The speedup is practically constant for $N_\Delta \gtrsim 500$. As this is still somewhat small for a statistical ensemble from a physical point of view, for practical purposes one may safely assume that the speedup achieved by using GPUs for the current problem is the maximum value that is attained. From the figures one can see that the speedup achievable for the physically interesting cases (many field modes, many atomic levels and many atoms) can exceed 20 for the faster of the two GPUs and float precision, and exceeds 10 for the same GPU but double precision. Note that when accuracy is of no great concern, a further speedup of some programs may be achieved using faster, but less precise arithmetic functions such as division or square root. We have not exploited this possibility in our work.

There is an interesting and well defined dip in the speedup on all of the figures for the GTX580. Less visible, but there are also slight dips on the curves for the GTX460 as well. The cause for the dips relates to kernel occupancy and the explanation illustrates how important is the ability of the GPU to schedule a great number of warps for execution simultaneously. As previously mentioned, the most time consuming part of the computation, the calculation of the atomic time evolution (RK4 kernel) is coded such that a single atom is calculated by a single block i.e. it is scheduled on a single SM. Thus it is obvious that choosing the number of atoms to be an integer multiple of the SM number of the GPU (16 for GTX580, 7 for GTX460) is advantageous for performance, because all SMs can be handed an equal share of the task. But that is not all - analyzing the occupancy of the RK4 kernel shows that more than one block can be scheduled on a SM simultaneously, e.g. for $N_v = 32$ and $M = 8$ four blocks, or for $N_v = 16$ and $M = 4$ eight blocks. (For the RK4 kernel, the occupancy depends only on N_v and M, so it is the same for all three test problems.) This means that e.g. for $N_v = 32$ and $M = 8$, the

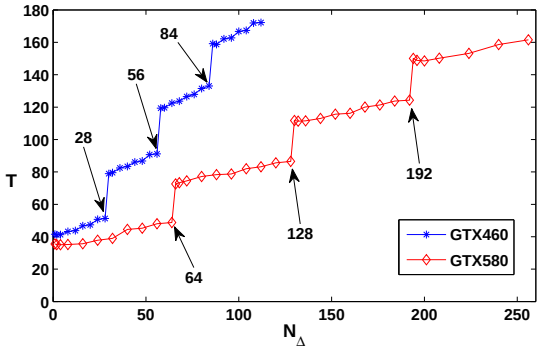

Figure 4: Overall program execution time for $N_F = 19$, $N_v = 32$, $M = 8$ as a function of N_Δ in seconds. The execution time shows step-like jumps each time the number of atoms crosses the multiples of 64 and 28, the number of atoms that can be scheduled for execution on the two GPUs simultaneously.

GTX580 board handles $4 \times 16 = 64$ atoms simultaneously, while the GTX460 board handles $4 \times 7 = 28$ atoms. Performance is thus expected to be optimal when N_Δ is chosen an integer times these values.

Figure 4 shows the program execution time as a function of N_Δ with high resolution. For both GPUs, the same pattern can be observed: the execution time follows a step-like curve. It increases slowly as N_Δ increases until it reaches $n \times 64$ or $n \times 28$ and then suddenly jumps when a single "surplus" atom is added. This jump between e.g. $N_\Delta = n \times 64$ and $N_\Delta = n \times 64 + 1$ is considerably greater than the overall increase in execution time between $N_\Delta = n \times 64 + 1$ and $N_\Delta = (n + 1) \times 64$. (Clearly, the overall execution time still increases during this latter period as well because other kernels also executed during the calculation do not follow this occupancy pattern.) Because the computation time of the program executing on the CPU increases much more smoothly, the speedup S will be greatest when N_Δ is chosen to be $n \times 64$ (or $n \times 28$). In figures 1, 2 and 3 depicting the speedup, the lowest graph shows that the it peaks at $N_\Delta = 64$ for the GTX580 board, then decreases slightly for the next two data points ($N_\Delta = 80$ and $N_\Delta = 96$) and is optimal again for $N_\Delta = 128$. As the rest of the data points are all integer multiples of 64, no further dips are observed.

A similar analysis could be done for kernels performing other steps in the program. It is most likely that the parameters which yield optimal speedup compared to the CPU program are not identical for these. Therefore, in the case of different applications, where the calculation does not have the computational load concentrated in a single step to this extent, the performance gain will not be as susceptible to the ideal choice of parameters.

The slower of the two GPUs still proved to be up to an order of magnitude faster than the CPU itself in performing the calculations. This is interesting from an economical viewpoint - it was a relatively low-end graphics board, costing less than the CPU itself. Furthermore, several graphics boards may be inserted in a desktop computer and used simultaneously to perform the calculations, multiplying the available computing power.

6. Summary and Outlook

We have investigated the possibility of using GPUs in a desktop computer for the numerical solution of the Maxwell-Bloch equations of resonant nonlinear optics. When multimode light propagation has to be investigated in a medium of complicated (many level) atomic systems, whose resonance frequencies are inhomogeneously broadened, the problem becomes numerically very demanding. However, the capability of GPUs for massively parallel calculations turn out to be very useful. We have compared the execution time of three problems for a wide range of parameters to the execution time of single- and multithread programs executing on the CPU. We have found that for realistic applications, using a high-end GPU can easily yield a factor 20 performance gain over the use of current multicore CPUs, and even for a low-end GPU the gain can be a factor of 10. Investigating the parameter dependence of the speedup, we have shown how important it is to divide the computational load over the serial multiprocessors of the GPU in an optimal way.

While we investigated pulse propagation phenomena, the most laborious part of our calculation was in fact a relatively common problem encountered in atomic physics. The solution of the Schrödinger equation for statistical ensembles is, for example, at the heart of the Quantum Monte-Carlo scheme [24, 25] that is used widely. Whether the ensemble is composed of simple, few-level quantum systems, or complicated, many-level ones is not important, variables can always be grouped so that all cores of the SMs are fully utilized. Therefore the methods presented here and the conclusions drawn about the applicability and efficiency of GPUs to solve the problem are relevant to a much wider,

more general field than pulse propagation in resonant nonlinear optics. Performance gains make the effort of programming computations to execute on GPUs worth while, especially because programming tools aimed at providing easier access to the computational powers of GPUs are being developed at a fast pace and becoming available from high level languages such as Matlab.

Acknowledgment

The financial support of the Janos Bolyai Research Fellowship of the Hungarian Academy of Sciences is gratefully acknowledged. The work has been funded by the Research Fund of the Hungarian Academy of Sciences (OTKA) under Contract No. F67922.

[1] S. L. McCall and E. L. Hahn, Phys. Rev. **183**, (1969) 457.

[2] L. Allen and J. H. Eberly, *Optical Resonance and Two-Level Atoms* (Dover Publications, New York 1978)

[3] S. E. Harris, J. E. Field, and A. Imamoglu Phys. Rev. Lett **66**, (1990) 1107.

[4] Michael Fleischhauer, Atac Imamoglu, Jonathan P. Marangos, Rev. Mod. Phys. **77**, (2005) 633.

[5] S. E. Harris, Phys. Rev. Lett. **70**, (1993) 552.

[6] F. T. Hioe and R. Grobe, Phys. Rev. Lett. **73**, (1994) 2559-2562.

[7] Q-Han Park and H. J. Shin, Phys. Rev. A **57**, (1998) 4643-4653.

[8] Ashiqur Rahman and J. H. Eberly, Phys. Rev. A **58**, (1998) R805-808.

[9] N. A. Kurnit, I. D. Abella and S. R. Hartmann, Phys. Rev. Lett **13**, (1965) 567.

[10] I. D. Abella, N. A. Kurnit and S. R. Hartmann, Phys. Rev. **141**, (1966) 391.

[11] W. Tittel et al., Laser & Photon. Rev. **4**, (2010) 244-267.

[12] G. Demeter, Z. Kis and U. Hohenester, Phys. Rev. A **85**, (2012) 033819.

[13] Guokui Liu and Bernard Jacquier eds., *Spectroscopic Properties of Rare Earths in Optical Materials*, (Springer Series in Materials Science, Springer, 2005)

[14] Brian Henderson and Ralph H. Bartram, *Crystal-Field Engineering of Solid-State Laser Materials*, (Cambridge University Press, 2000)

[15] A. Frezzotti, G. P. Ghiroldi, L. Gibelli, Comput. Phys. Commun. **182**, (2011) 2445-2453.

[16] Henrik Schulz, Géza Ódor, Gergely Ódor, Máté Ferenc Nagy, Comput. Phys. Commun. **182**, (2011) 1467-1476.

[17] J. M. Alcaraz-Pelegrina, P. Rodríguez-García, Comput. Phys. Commun. **182**, (2011) 1414-1420.

[18] Heiko Bauke, Christoph H. Keitel, Comput. Phys. Commun. **182**, (2011) 2454-2463.

[19] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati and Denis Barthou, *Analysing the Variability of OpenMP Programs Performance on Multicore Architectures* In Proceedings of the Fourth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2011), Heraklion, Greece. January 23, 2011, available from http://www.prism.uvsq.fr/∼amaz/Mazouz-MULTIPROG2011.pdf

[20] Jason Sanders, Edward Kandrot, *CUDA By Example - An Introduction to General-Purpose GPU Programming* (Addison-Wesley 2011)

[21] NVIDIA CUDA Programming Guide, CUDA C Best Practices Guide, available at http://developer.nvidia.com/nvidia-gpu-computing-documentation

[22] CUDA-MEMCHECK User Manual, CUDA-GDB Nvidia Cuda Debugger - 4.0 Release User Manual, Compute Visual Profiler User Guide, available at http://developer.nvidia.com/nvidia-gpu-computing-documentation

[23] Header file by Christian Buchner available at http://forums.nvidia.com/index.php?showtopic=73978, Copyright (c) 2008-2009 Christian Buchner

[24] J. Dalibard, Y. Castin and K. Molmer, Phys. Rev. Lett. **68**, (1992) 580.

[25] R. Dum, P. Zoller and H. Ritsch, Phys. Rev. A **45**, (1992) 4879.