Computer Physics Communications 184 (2013) 1250–1271

Contents lists available at SciVerse ScienceDirect

Computer Physics Communications

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

NLSEmagic: Nonlinear Schrödinger equation multi-dimensional Matlab-based GPU-accelerated integrators using compact high-order schemes☆

R.M. Caplan *

Nonlinear Dynamical System Group, Computational Science Research Center, and Department of Mathematics and Statistics, San Diego State University, San Diego, CA 92182-7720, USA

ARTICLE INFO

Article history:
Received 20 August 2012
Received in revised form
2 December 2012
Accepted 5 December 2012
Available online 19 December 2012

Keywords:
Nonlinear Schrödinger equation
Bose–Einstein condensates
GPU
GPGPU
Explicit finite difference schemes

ABSTRACT

We present a simple to use, yet powerful code package called NLSEmagic to numerically integrate the nonlinear Schrödinger equation in one, two, and three dimensions. NLSEmagic is a high-order finite-difference code package which utilizes graphic processing unit (GPU) parallel architectures. The codes running on the GPU are many times faster than their serial counterparts, and are much cheaper to run than on standard parallel clusters. The codes are developed with usability and portability in mind, and therefore are written to interface with MATLAB utilizing custom GPU-enabled C codes with the MEX-compiler interface. The packages are freely distributed, including user manuals and set-up files.

Program summary

Program title: NLSEmagic
Catalogue identifier: AEOJ_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AEOJ_v1_0.html
Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland
Licensing provisions: Standard CPC licence, http://cpc.cs.qub.ac.uk/licence/licence.html
No. of lines in distributed program, including test data, etc.: 124453
No. of bytes in distributed program, including test data, etc.: 4728604
Distribution format: tar.gz
Programming language: C, CUDA, MATLAB.
Computer: PC, MAC.
Operating system: Windows, MacOS, Linux.
Has the code been vectorized or parallelized?: Yes.
Number of processors used: Single CPU, number of GPU processors dependent on chosen GPU card (max is currently 3072 cores on GeForce GTX 690).
Supplementary material: Setup guide, Installation guide.
RAM: Highly dependent on dimensionality and grid size. For typical medium–large problem size in three dimensions, 4GB is sufficient.
Keywords: Nonlinear Schröodinger Equation, GPU, high-order finite difference, Bose-Einstien condensates.
Classification: 4.3, 7.7.
Nature of problem:
Integrate solutions of the time-dependent one-, two-, and three-dimensional cubic nonlinear Schrödinger equation.
Solution method:
The integrators utilize a fully-explicit fourth-order Runge–Kutta scheme in time and both second- and fourth-order differencing in space. The integrators are written to run on NVIDIA GPUs and are interfaced with MATLAB including built-in visualization and analysis tools.

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

* Correspondence to: Predictive Science Inc. 9990 Mesa Rim Rd, Suite 170, San Diego, CA 92121, USA. Tel.: +1 858 225 2314.
E-mail addresses: sumseq@gmail.com, caplanr@predsci.com.
URL: http://nlds.sdsu.edu.

0010-4655/$ – see front matter
doi:10.1016/j.cpc.2012.12.010

Restrictions:
The main restriction for the GPU integrators is the amount of RAM on the GPU as the code is currently only designed for running on a single GPU.
Unusual features:
Ability to visualize real-time simulations through the interaction of MATLAB and the compiled GPU integrators.
Additional comments:
Setup guide and Installation guide provided. Program has a dedicated web site at www.nlsemagic.com.
Running time:
A three-dimensional run with a grid dimension of $87 \times 87 \times 203$ for 3360 time steps (100 non-dimensional time units) takes about one and a half minutes on a GeForce GTX 580 GPU card.

1. Introduction

The nonlinear Schrödinger equation (NLSE) is a universal model describing the evolution and propagation of complex field envelopes in nonlinear dispersive media. As such, it is used to describe many physical systems including Bose–Einstein condensates [1], nonlinear optics [2], the evolution of water waves, thermodynamic pulses, nonlinear waves in fluid dynamics, and waves in semiconductors [3]. The general form of the NLSE can be written as

$$i\frac{\partial \Psi}{\partial t} + a\nabla^2\Psi - V(\mathbf{r})\Psi + s|\Psi|^2\Psi = 0, \tag{1}$$

where $\Psi(\mathbf{r}, t) \in \mathbb{C}$ is the value of the wavefunction, $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$ is the Laplacian operator, and where $a > 0$ and s are parameters defined by the system being modeled. $V(\mathbf{r})$ is an external potential term, which when included, makes Eq. (1) known as the Gross–Pitaevskii equation [1].

Graphical processing units were first developed in order to allow graphics cards to parallelize their massive computations to speed up video games and image processing. After realizing that such hardware could be adapted to run scientific codes as well, companies such as NVIDIA have developed APIs (such as the compute unified device architecture (CUDA) and OpenCL) to allow GPUs to be used for general computing.

For many problems, GPU computing is seen as a large improvement over previous parallel techniques since access to a large cluster can be expensive or not available. A typical GPU (as of this writing) can have up to 3072 processing cores and up to 12GB of RAM, with a throughput of over a Tera-FLOP on a single card. The price of the GPUs is another major factor, as one (as of this writing) can purchase an off-the-shelve GPU with 1536 cores, and 4GB of RAM for under $450. This allows for super-computing capabilities to be realized on a single desktop PC for medium-sized problems.

GPUs have been used for various finite-difference PDE integrator codes with good results [4–9]. In Ref. [4], the authors wrote code to simulate the two-dimensional Maxwell equations yielding a speedup of up to ten times when compared to the CPUs of the day (2004). They did this before CUDA was developed using assembly code directly. Ref. [5] also did not utilize CUDA for their simulations of the three-dimensional Maxwell equations, yielding speedups of over 100 times. The two-dimensional Maxwell equations were simulated in Ref. [6] using CUDA code where speedups of 50 were reported. In Ref. [7], the authors simulate the three-dimensional wave equation with an eighth-order finite-difference scheme using CUDA on multiple GPUs. A similar multi-GPU code was shown in Ref. [8] for the three-dimensional wave equation using a fourth-order finite-difference scheme where speedups of 60 were reported. In Ref. [9], the authors use CUDA to simulate the multi-dimensional complex Ginzburg–Landau equation (a generalization of the NLSE), but speedups versus CPU codes were not reported. All these studies indicate that a GPU treatment of the NLSE would be beneficial.

In this paper we describe our implementation of a code package called NLSEmagic (Nonlinear Schrödinger Equation Multi-dimensional Matlab-based GPU-accelerated Integrators using Compact High-order Schemes) which integrates the NLSE in one, two, and three dimensions. Previous codes have been published which integrate the NLSE in multi-dimensions [10] including a recent C-based code which implements OpenMP parallelism for multi-core CPU systems [11]. The code presented here differs from the previously published codes in two major ways. One, the NLSEmagic codes utilize NVIDIA GPUs for integrating the NLSE which can yield large speedups compared to a serial-CPU code, as well as decent speedups compared to typical multi-core CPUs (see Section 7). Second, the codes presented here are MATLAB-based allowing for user-friendly setup of problems, ease of use, built-in MATLAB analysis and optimization functions, and on-the-fly visualizations drastically reducing the need for post-processing results.

In order to allow the MATLAB-based codes to run at speeds equal to typical compiled codes, we write the main NLSE integrators as custom C codes which connect to MATLAB through the MEX interface compiler. The compiled integrators are called from a MATLAB script code and are up to 10 times faster than writing the integrators as native MATLAB script codes (as shown in Section 5.3). The C codes written for the MEX interface have an additional advantage as they can be written to only have custom MATLAB-based code for the input and output of data from MATLAB allowing them to be portable to non-MATLAB codes if desired.

Since the native language for GPU-compatible CUDA codes is C, adding GPU functionality to MATLAB is possible using the MEX compiler interface. The newest versions of MATLAB have GPU-compatibility built-in [12], and ways to compile CUDA C code segments into callable MATLAB functions. However, because many researchers do not have the newest versions of MATLAB, and in the interest of portability, it is preferable not to use these built-in functions, but rather write portable C integrator codes which contain CUDA code directly, and compile them with the CUDA-capable MEX compiler called NVMEX [13].

Both serial-CPU and GPU-accelerated MEX routines are included in NLSEmagic and are called from MATLAB equivalently. They integrate the NLSE over a specified number of time-steps (the chunk-size) before returning the current solution Ψ to MATLAB. They utilize fully explicit finite-difference schemes, meaning that the solution at the next time-step only relies on values of the solution at the previous time-step. This eliminates the need for solving linear systems and nonlinear iterations at each time-step (which most implicit schemes do).

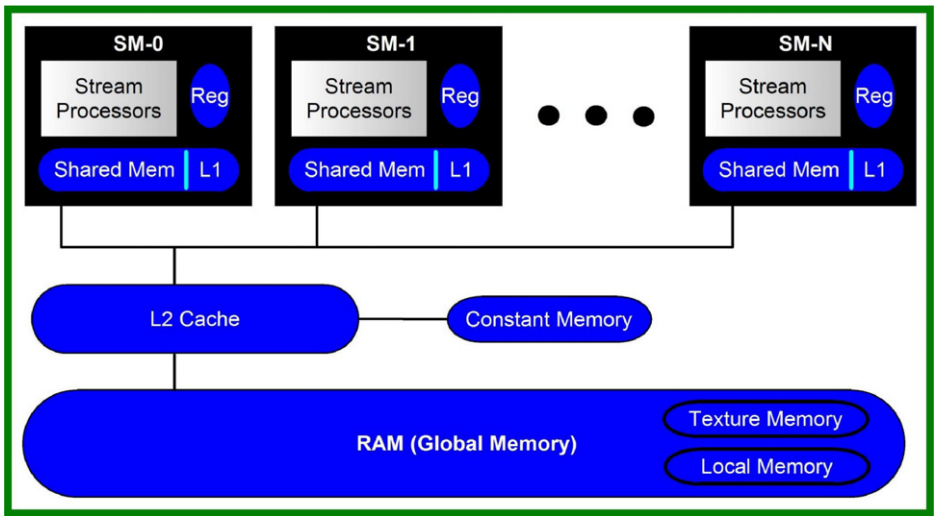

Fig. 1. (Color online) Simplified schematic of an NVIDIA GPU with Fermi architecture (a Tesla architecture looks the same but without caches).

The drawback is that a conditional stability criteria exists that limits the time-step size based on the spatial grid-spacing and the scheme used to approximate the spatial derivatives. This relationship is discussed in Section 3.4, and is implemented into the driver scripts of NLSEmagic. The spatial schemes used are a standard second-order central difference and a two-step high-order compact scheme (2SHOC) [14] which is fourth-order accurate.

This paper is organized as follows: In Section 2, we describe general purpose GPUs, specifically those produced by NVIDIA. We discuss the physical structure of the GPUs as well as the CUDA API logical structure. Compatibility and portability issues are also discussed. In Section 3, we discuss the numerical algorithms used in NLSEmagic including boundary conditions and stability. The example solutions to the NLSE that are used throughout the simulations in this paper are described in Section 4. The serial code implementation of the integrators is described in Section 5, including the script integrators and a detailed description of the MEX code integrators. Speedup results between the MEX codes and the equivalent C codes are shown. Section 6 describes in detail our implementation of the NLSEmagic integrators in CUDA MEX codes. The speedup results of the CUDA MEX codes compared to the equivalent serial MEX codes are shown in Section 7. An overview of the NLSEmagic code package is described in Section 8 including distribution details.

2. General purpose GPU computing with NVIDIA GPUs

Although there are other companies that produce GPU graphics cards (such as ATI), NVIDIA is by far the most established when it comes to general-purpose GPUs (GPGPU). Their compute-only Tesla GPGPU cards as well as their native compute unified device architecture (CUDA) API and enormous support and development infrastructure make NVIDIA the most logical choice for developing and running GPU codes (compatibility and portability concerns will be discussed in Section 2.4). In this section we focus on the physical structure of the GPUs as well as the logical structure defined by the CUDA API.

2.1. NVIDIA GPU physical structure

Since the programming model for NVIDIA GPUs is very closely related to their physical design, it is important to have an overview of their physical structure. A simplified schematic of a Fermi architecture NVIDIA GPU is shown in Fig. 1. The GPU contains a number of stream multi-processors (SM) which each contain a number of stream processors (SP) which are the compute cores of the GPU. Each SM performs computations independently and simultaneously from the other SMs and has no connectivity or communication with other SMs. Each SM has a small, fast memory which is configured to allocate some space for a level 1 (L1) cache and the rest of the space for shared memory. The shared memory is shared between all SPs, so each core can access any part of shared memory. Each SM also has fast registers for the SPs to use for computations and storage of local variables.

The main memory of the GPU is a large amount of DRAM called the *global memory space* which, depending on the GPU, can have a capacity from 512 MB up to 12 GB. Accesses to the global memory from the SPs are much slower than accesses to shared memory [15]. Parts of the global memory space are used for local variables (when they cannot fit into registers or cache) and texture memory. Between the DRAM and the SMs is a level 2 (L2) cache which improves memory performance when SMs access global memory. Additional hardware details are beyond the required scope of this paper.

A typical GPU code transfers data from the host computer's RAM to the GPU RAM, performs computations using the data on the SMs of the GPU, and then when completed, transfers the resulting data back to the host computer's RAM. The memory transfer has large latency associated with it, and so it is ideal to compute as much as possible on the GPU before transferring the data back. However, since the host computer cannot 'see' the data until it has been transferred back from the GPU, a trade-off between maximum performance and usability is often encountered for time-stepping problems (see Section 7.1).

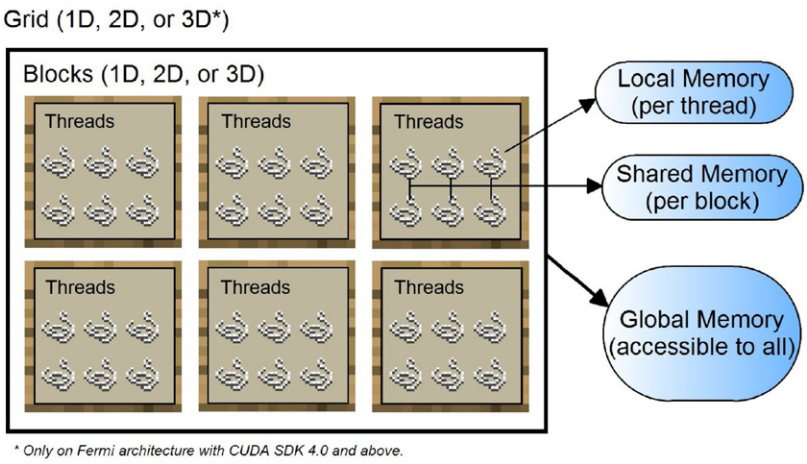

Fig. 2. (Color online) Schematic of the CUDA API logic structure.

2.2. CUDA API and logical structure

NVIDIA allows programmers to utilize its GPUs through an API called the compute unified device architecture (CUDA) API. CUDA is a code extension to C/C++ which gives low-level access to the GPUs memory and processing abilities. The CUDA codes are compiled by a free compiler provided by NVIDIA called `nvcc` (a FORTRAN CUDA compiler is also available through PGI).

There are two sections of a CUDA C code (usually written in a file with a `.cu` extension). One section of code is the host code which is executed on the CPU. This section contains setup commands for the GPU, data transfer commands to send data from the CPU to the GPU (and vice-versa), and code to launch computations on the GPU. The second part of a CUDA code contains what are known as CUDA *kernel* routines. These are routines which get compiled into binaries which are able to be executed on the GPU by calls invoked by the host code.

The CUDA programming model is based on a logical hierarchical structure of *grids*, *blocks*, and *threads*. This hierarchy is shown in Fig. 2. When a kernel is launched, it is launched on a user-defined compute-grid which can be one-, two-, or three-dimensional. This grid contains a number of blocks where the computations to be performed are distributed. Each block must be able to be computed independently as no synchronization is possible between blocks. Each block contains a number of threads, which are laid out in a configuration that is one-, two-, or three-dimensional. The threads perform the computations and multiple threads are computed simultaneously on the SM. Synchronization of threads within a block is possible, but for best performance, should be kept to a minimum. Each thread has its own local memory and all threads in a block have access to a block-wide shared memory. All threads in all blocks on the grid have access to the global memory.

Each thread has access to intrinsic variables (stored in registers) which are used to identify its position in the compute-grid and block. The grid dimensions are stored in variables called `gridDim.x`, `gridDim.y`, and `gridDim.z` and the block dimensions are stored in `blockDim.x`, `blockDim.y`, and `blockDim.z`. The block that a thread is contained in is given in coordinates by the variables `blockIdx.x`, `blockIdx.y`, and `blockIdx.z`. The thread's position in the block is given by the variables `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`. All these variables are used to determine what part of the problem the specific thread should compute.

The logical structure relates directly to how the hardware of the GPU executes a kernel code. Each block is executed by an SM, where each thread is executed by an SP. This is why the threads within a block have access to a shared memory, but blocks cannot access each others shared memory. The local memory of each thread is stored in the registers of the SM, unless the register space is used up in which case the local variables get stored in cache, and, if the cache's are full, to global memory (reducing performance).

2.3. Compatibility

Since new and updated GPU cards are being developed and released continuously (such as the recently released Kepler-based GPUs), compatibility can become a major concern. To help with this issue, NVIDIA has maintained a compatibility scheme known as the major/minor compute capability of a GPU. This is a number in the form of *y*.*x* where *y* and *x* are the major and minor compute capabilities respectively. Whether code written and compiled for one GPU will work on a different model GPU depends on the compute capabilities of the two cards and on how the code was compiled (see Ref. [15] for details).

CUDA codes can be compiled into a binary executable for a specific GPU known as 'cubin' objects, and/or be compiled into parallel thread execution (PTX) assembly code which is compiled into binary at runtime for the GPU being used (which can have a slight performance impact).

The compiler options for the NLSEmagic code package described in this paper have been set to compile PTX code for 1.0, 1.3, and 2.0, as well as cubin objects for compute capability 2.0 (The codes have also been tested to compiled on the new compute capability 3.x architectures, which requires modification to the compile scripts).

2.4. Portability

A major reason many scientific programmers are hesitant to write their programs for GPUs and using CUDA to do so, is the issue of portability. As a solution to these concerns, a new API called OpenCL [16] has been developed by the Khronos Group [16] which can function across multiple GPU vendors, as well as other CPU architectures. Although code written in OpenCL is guaranteed to work on all multi-core

architectures, specific optimization is necessary for the codes to work efficiently. A number of studies have been done comparing the performance of OpenCL versus CUDA [17–19]. In general, the CUDA codes outperform the OpenCL equivalent codes (sometimes by a significant percentage). However, with further OpenCL-specific performance tunings, it is possible to get the OpenCL codes to run nearly as fast as the CUDA codes. OpenCL and CUDA are very similar in terms of logical structure and code design. Therefore, codes written in CUDA (such as NLSEmagic) can easily be changed into OpenCL codes (in fact, an *automatic* CUDA-to-OpenCL converter called `cu2cl` has recently been developed [20]).

Many other portability options exist for CUDA codes and GPU codes in general including the Portland Group's x86 CUDA compiler [21], the MCUDA compiler created by the University of Illinois [22], and the Ocelot project from Georgia Tech [23] (see references for details).

3. Finite-difference schemes

There are many methods to numerically integrate the NLSE. The scheme we use in NLSEmagic is the classic fourth-order Runge–Kutta (RK4) in time and a choice of either a standard second-order central-difference (CD) in space for the Laplacian operator, or a fourth order two-step high order compact scheme (2SHOC) [14]. These schemes are some of the simplest explicit finite-difference schemes for the NLSE that are conditionally stable. Once the principles of the CUDA implementation are understood, more advanced schemes could be added to the code package (such as Mimetic operators [24]).

The basic methods used in the code presented here are different than those used in Refs. [10,11]. There, the authors use a semi-implicit, second-order in space and time split-step Crank–Nicolson (SSCN) method. The major difference between the methods are that the RK4 used here is conditionally stable (see Section 3.4), while the SSCN scheme is unconditionally stable, which in some cases can be more efficient depending on the accuracy and resolution requirements of the problem. However, since the schemes used in NLSEmagic are up to fourth-order accurate in both space and time, the equivalent accuracy of a simulation can often be realized on a coarser grid, in which case the problem of bounds on the time-step can be greatly reduced.

In three dimensions, the computational grid consists of $N \times M \times L$ grid points ($N \times M$ in two dimensions, N in one dimension). The wave function of the NLSE is discretized as

$$\Psi(x, y, z, t) \equiv \Psi^n_{i,j,k}, \tag{2}$$

where n is the current time step and i, j, k are the spatial positions on the grid. The time-step is denoted as $\mathbf{k}$ and the spatial step-size is the same in each direction and defined by h. Therefore, $t = n\mathbf{k}$, $x = x_0 + ih$, $y = y_0 + jh$, and $z = z_0 + k\,h$. We define the end-time of the simulation as $t_{\text{end}} = \mathbf{k}\, t_{\text{res}}$ where t_{res} is the number of time-steps taken.

3.1. Fourth-order Runge–Kutta

In order to avoid (especially in three-dimensional problems) the need to solve a linear system and run nonlinear iteration routines at each time step, we use a fully explicit time-difference scheme. The classic four-stage, fourth-order Runge–Kutta (RK4) scheme [25] is the simplest single-stage explicit time-stepping that can be used for the NLSE directly (i.e, not using a real-imaginary staggered step as in Ref. [26]) and is (conditionally) stable [27]. The RK4 scheme can be written in algorithmic form as

$$\begin{aligned}
&(1)\ K_{\text{tot}} = F(\Psi^n) && (6)\ K_{\text{tmp}} = F(\Psi_{\text{tmp}})\\
&(2)\ \Psi_{\text{tmp}} = \Psi^n + \frac{\mathbf{k}}{2} K_{\text{tot}} && (7)\ K_{\text{tot}} = K_{\text{tot}} + 2\,K_{\text{tmp}}\\
&(3)\ K_{\text{tmp}} = F(\Psi_{\text{tmp}}) && (8)\ \Psi_{\text{tmp}} = \Psi^n + \mathbf{k}\, K_{\text{tmp}}\\
&(4)\ K_{\text{tot}} = K_{\text{tot}} + 2\,K_{\text{tmp}} && (9)\ K_{\text{tmp}} = F(\Psi_{\text{tmp}})\\
&(5)\ \Psi_{\text{tmp}} = \Psi^n + \frac{\mathbf{k}}{2} K_{\text{tmp}} && (10)\ \Psi^{n+1} = \Psi^n + \frac{\mathbf{k}}{6}(K_{\text{tot}} + K_{\text{tmp}}),
\end{aligned} \tag{3}$$

where

$$F(\Psi) = \frac{\partial \Psi}{\partial t} = i\left[a\nabla^2\Psi + \left(s\,|\Psi|^2 - V(\mathbf{r})\right)\Psi\right].$$

In NLSEmagic, the Laplacian of Ψ in $F(\Psi)$ is evaluated using either the CD or 2SHOC scheme. We denote the resulting combined scheme as RK4 + CD and RK4 + 2SHOC respectively.

The RK4 has an error which is fourth-order in the time step ($O(\mathbf{k}^4)$). In NLSEmagic, this error will always be much less than the error in computing the Laplacian of Ψ because, as we will see in Section 3.4, in order for the RK4 scheme to be stable we must have $\mathbf{k} \propto h^2$, the proportionality constant depending on the dimensionality of the problem and spatial scheme. Thus, an error of $O(\mathbf{k}^4)$ in our case is proportional to an error of $O(h^8)$, and since our spatial schemes are accurate only to $O(h^2)$ or $O(h^4)$, the time-step error resulting from the RK4 is negligible.

3.2. Difference schemes for the Laplacian operator

NLSEmagic has two options for computing the Laplacian. The first is the standard second-order central difference scheme (CD), and the other is a two-step high-order compact scheme (2SHOC) which is fourth-order accurate [14].

The 2SHOC scheme allows for the use of courser grids with equivalent accuracy to the CD scheme, or for use with simulations that need high accuracy. Usually, high-order schemes have a wide stencil, which are not ideal for parallel applications due to extra communication and/or memory latency. Also, the grid points near the boundary are difficult to deal with. The 2SHOC schemes avoid these problems because in each step, the computation is compact (relying only on adjacent grid points) making it easier to code. The drawbacks of the scheme are that they require the storage of an extra temporary variable and an increase in the number of floating point operations per

grid point when compared to the standard fourth-order wide stencils. However, for our GPU implementation, we feel the advantages of using the 2SHOC scheme outweigh the disadvantages.

In one-dimension, the two steps of the 2SHOC scheme are defined as

$$(1)\ D_i = \frac{1}{h^2}\left(\Psi_{i+1} - 2\Psi_i + \Psi_{i-1}\right), \tag{4}$$

$$(2)\ \nabla^2\Psi_i \approx \frac{7}{6}D_i - \frac{1}{12}\left(D_{i+1} + D_{i-1}\right). \tag{5}$$

In two dimensions, the 2SHOC scheme is given by

$$(1)\ D_{i,j} = \frac{1}{h^2}\begin{array}{|c|c|c|}\hline & 1 & \\ \hline 1 & -4 & 1 \\ \hline & 1 & \\ \hline\end{array}\Psi_{i,j} \tag{6}$$

$$(2)\ \nabla^2\Psi_{i,j} \approx -\frac{1}{12}\begin{array}{|c|c|c|}\hline & 1 & \\ \hline 1 & -12 & 1 \\ \hline & 1 & \\ \hline\end{array}D_{i,j} + \frac{1}{6\,h^2}\begin{array}{|c|c|c|}\hline 1 & & 1 \\ \hline & -4 & \\ \hline 1 & & 1 \\ \hline\end{array}\Psi_{i,j}, \tag{7}$$

and in three dimensions,

$$(1)\ D_{i,j,k} = \frac{1}{h^2}\left(\begin{array}{|c|c|c|}\hline & & \\ \hline & 1 & \\ \hline & & \\ \hline\end{array}\Psi_{i,j+1,k} + \begin{array}{|c|c|c|}\hline & 1 & \\ \hline 1 & -6 & 1 \\ \hline & 1 & \\ \hline\end{array}\Psi_{i,j,k} + \begin{array}{|c|c|c|}\hline & & \\ \hline & 1 & \\ \hline & & \\ \hline\end{array}\Psi_{i,j-1,k}\right), \tag{8}$$

$$\begin{aligned}(2)\ \nabla^2\Psi_{i,j,k} \approx{}& -\frac{1}{12}\left(\begin{array}{|c|c|c|}\hline & & \\ \hline & 1 & \\ \hline & & \\ \hline\end{array}D_{i,j+1,k} + \begin{array}{|c|c|c|}\hline & 1 & \\ \hline 1 & -10 & 1 \\ \hline & 1 & \\ \hline\end{array}D_{i,j,k} + \begin{array}{|c|c|c|}\hline & & \\ \hline & 1 & \\ \hline & & \\ \hline\end{array}D_{i,j-1,k}\right) \\ &+ \frac{1}{6\,h^2}\left(\begin{array}{|c|c|c|}\hline & 1 & \\ \hline 1 & & 1 \\ \hline & 1 & \\ \hline\end{array}\Psi_{i,j+1,k} + \begin{array}{|c|c|c|}\hline 1 & & 1 \\ \hline & -12 & \\ \hline 1 & & 1 \\ \hline\end{array}\Psi_{i,j,k} + \begin{array}{|c|c|c|}\hline & 1 & \\ \hline 1 & & 1 \\ \hline & 1 & \\ \hline\end{array}\Psi_{i,j-1,k}\right).\end{aligned} \tag{9}$$

The standard second-order central differencing in each dimension is simply given by step one of the 2SHOC schemes shown above.

3.3. Boundary conditions

NLSEmagic includes three boundary condition options: (1) Dirichlet (D), (2) Modulus-Squared Dirichlet (MSD), and (3) Laplacian-zero (L0). For one-dimensional simulations, NLSEmagic1D also contains code for a one-sided (1S) boundary condition.

For each boundary condition, it is necessary to define the time-derivative at a boundary point so that it can be implemented in each step of the RK4 scheme. In order to use the boundary conditions with the 2SHOC scheme, they additionally need to be expressed in terms of the Laplacian in order to compute proper boundaries in the first step of the 2SHOC.

Dirichlet boundary conditions are defined as when the value of the function at the boundaries is fixed to be a constant value, i.e.

$$\Psi_b = B, \tag{10}$$

where the subscript b represents a boundary point, and B is a real constant. The time-derivative formulation of this condition is simply

$$\frac{\partial\Psi_b}{\partial t} = 0. \tag{11}$$

For the NLSE, the Dirichlet condition in terms of the Laplacian of the wavefunction is

$$\nabla^2\Psi_b = -\frac{1}{a}(s|\Psi_b|^2 - V_b)\Psi_b. \tag{12}$$

The Modulus-squared Dirichlet boundary condition is defined to be where the modulus-squared of the wavefunction at the boundary is set to a fixed constant, i.e.

$$|\Psi_b|^2 = B, \tag{13}$$

where B is a real constant. The MSD boundary condition is useful for many problems, especially those with a constant density background (such as the examples described in Section 4). The time-derivative formulation of the MSD boundary condition can be approximated as [28]

$$\frac{\partial\Psi_b}{\partial t} \approx i\,\mathrm{Im}\left[\frac{1}{\Psi_{b-1}}\frac{\partial\Psi_{b-1}}{\partial t}\right]\Psi_b, \tag{14}$$

where $\frac{\partial\Psi_{b-1}}{\partial t}$ is pre-computed using the internal finite-difference scheme. This reliance on the previously computed value of the time derivative of the interior point requires special treatment in the CUDA codes of NLSEmagic (see Section 6 for details). The Laplacian form of the MSD for the NLSE is

$$\nabla^2\Psi_b \approx \left[\mathrm{Im}\left(i\,\frac{\nabla^2\Psi_{b-1}}{\Psi_{b-1}}\right) + \frac{1}{a}\left(N_{b-1} - N_b\right)\right]\Psi_b, \tag{15}$$

where

$$N_b = s\,|\Psi_b|^2 - V_b, \qquad N_{b-1} = s\,|\Psi_{b-1}|^2 - V_{b-1}.$$

The Laplacian-zero boundary condition is defined as setting the Laplacian of the wavefunction at the boundary to zero. The time-derivative formulation of the L0 boundary condition for the NLSE is given by

$$\frac{\partial \Psi_b}{\partial t} = i\,(s|\Psi_b|^2 - V_b)\,\Psi_b, \tag{16}$$

while the Laplacian form is, by definition,

$$\nabla^2 \Psi_b = 0. \tag{17}$$

The boundary conditions included in NLSEmagic are chosen due to their relatively easy implementation and broad applicability. Additional boundary conditions could be added to the NLSEmagic codes if needed.

3.4. Stability

The finite-difference schemes used in NLSEmagic are fully explicit and, like most explicit schemes, are conditionally stable. This means that for a given spatial step size h, there exists a maximum value of the time-step $\mathbf{k}$ that can be used without the scheme becoming unstable (i.e, blow-up). In Ref. [27], we performed a full linearized stability analysis for the schemes in NLSEmagic. The results are coded into the scripts of NLSEmagic where, given the spatial step size h, the largest stable time-step $\mathbf{k}$ is automatically set. However, from experience it is observed that the purely linear bounds found in Ref. [27] are typically very close to the linearized bounds, and since the bounds must be artificially lowered (due to nonlinear effects), using the purely linear bounds is often all that is needed (an exception to this would occur in the presence of large external potential values). Therefore, the basic driver scripts of NLSEmagic only compute the linear bounds.

The linear stability bounds of the RK4 + CD scheme for the NLSE with $V(\mathbf{r}) = 0$ and $s = 0$ using Dirichlet or periodic boundary conditions are

$$\mathbf{k} < \frac{h^2}{d\sqrt{2}\,a}, \tag{18}$$

where d is the dimensionality of the problem (1, 2, or 3). The equivalent linear bound for the RK4 + 2SHOC scheme are given by

$$\mathbf{k} < \left(\frac{3}{4}\right)\frac{h^2}{d\sqrt{2}\,a}. \tag{19}$$

In the NLSEmagic driver scripts, these bounds are multiplied by 0.8 to avoid instability due to nonlinearities, boundaries, and/or the external potential.

4. Example test problems

Before describing the implementation of the NLSEmagic codes, we show here the example test problems that we will make use of in testing the speedup of the codes.

In one-dimension, we use the following exact co-moving dark soliton solution to the NLSE with $V(x) = 0$ and $s < 0$ [29]:

$$\Psi(x,t) = \sqrt{\left|\frac{\Omega}{s}\right|}\,\tanh\left[\sqrt{\frac{|\Omega|}{2a}}\,(x - c\,t)\right]\exp\left(i\left[\frac{c}{2a}\,x + \left(\Omega - \frac{c^2}{4a}\right)t\right]\right), \tag{20}$$

where c is the velocity of the soliton and Ω is the frequency. The soliton describes a localized rarefaction curve in the modulus squared of Ψ which propagates without dispersion or dissipation amidst a constant density background. For our simulations we use $s = -1$, $a = 1$, $c = 0.5$, and $\Omega = -1$ and use a spatial grid-size of $h = 0.01$. The computed linear and linearized stability bounds for the RK4 + CD scheme yield a maximum available time-step of $\mathbf{k} = 0.0070711$ and $\mathbf{k} = 0.0070534$ respectively. For the RK4 + 2SHOC scheme, the stability bounds are $\mathbf{k} = 0.0053033$ and $\mathbf{k} = 0.0052934$ respectively. Therefore, we use a time-step of $\mathbf{k} = 0.005$ for all simulations to ensure stability. A depiction of such a soliton within our simulations is shown in Fig. 3. The solution has a constant modulus-square value at the boundaries (far from the soliton) and we therefore use the MSD boundary condition (14).

In two dimensions, we use an approximation to a steady-state dark vortex solution to the NLSE with $V(x) = 0$ and $s < 0$ given by [30]

$$\Psi(r,\theta,t) = f(r)\,\exp[i\,(m\,\theta + \Omega\,t)], \tag{21}$$

where m is the topological charge of the vortex (in our case, we use $m = 1$) and where $f(r)$ is a real-valued radial profile which can only be found exactly through numerical optimization routines. Since we will be simulating the vortex solution of on large grids (up to nearly 2000×2000), inserting the interpolated exact numerical solution of $f(r)$ onto the two-dimensional grid can take excessive time to formulate. Due to this, and since we are not interested in the exact solutions, we do not use the numerically exact profile but rather an approximation to it given by the initial condition of the one-dimensional dark soliton of Eq. (20) for $x > 0$ with $c = 0$.

We use a spatial step-size of $h = 0.25$. Although this allows for a maximum time step of approximately $\mathbf{k} = 0.016$, in order to allow for comparison between simulations of different dimensionality, we set the time-step to that of the one-dimensional examples, $\mathbf{k} = 0.005$. A depiction of the dark vortex within our simulations is shown in Fig. 3.

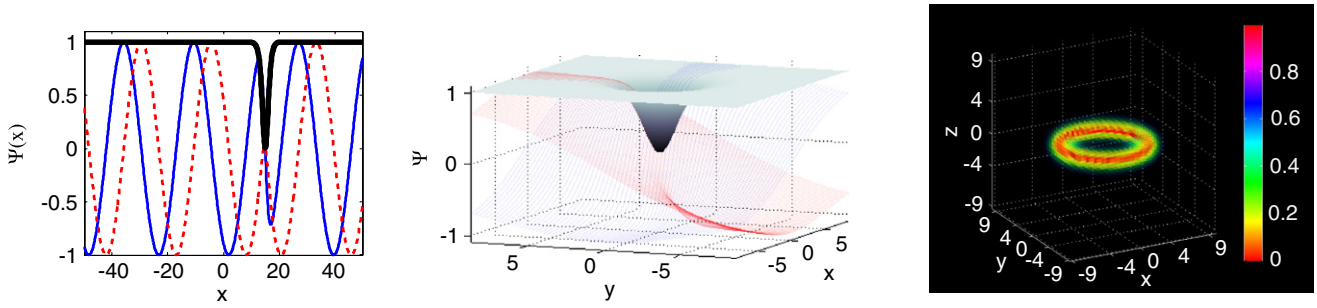

Fig. 3. (Color online) Left: Example of the one-dimensional dark soliton of Eq. (20) at time $t = 30$. The thin solid (blue) line and the thin dashed (red) line are the real and imaginary parts of Ψ, respectively. The thick solid (black) line is the modulus-squared of the wave-function $|\Psi|^2$. The grid size here is $N = 1000$. Middle: Example of the approximate two-dimensional dark vortex of Eq. (21) at time $t = 0$. The (blue) and (red) mesh lines are the real and imaginary parts of Ψ, respectively. The solid (gray) surface is the modulus-squared of the wave-function $|\Psi|^2$. The grid-size is $N = M = 70$. Right: Example of the approximate three-dimensional dark vortex ring of Eq. (22) at time $t = 0$. The image shows an inverted-volumetric rendering of the modulus-squared of Ψ. The grid size is $N = M = L = 29$.

In three dimensions, we use an approximation to a dark vortex ring solution to the NLSE amidst a co-moving back-flow which causes the vortex ring to be a steady-state solution (see Ref. [31] for details). The form of the initial condition in cylindrical coordinates is

$$\Psi(r, z, \theta, 0) = g(r, z)\ \exp\left[i\left(\frac{c}{2a}z\right)\right], \tag{22}$$

where $g(r, z)$ is taken to be a numerically-exact two-dimensional $m = 1$ dark vortex (which, since we only are using two-dimensional interpolations of resolutions up to 144×144, is able to be formulated efficiently in this case) at position $r = d$ in the r–z half-plane, where d is the radius of the vortex ring, and c is its intrinsic transverse velocity (given to asymptotic approximation in Ref. [32]). In our example, we choose $d = 5$ and use a spatial step of $h = 3/2$. Once again, for comparison purposes, we use a time-step of $\mathbf{k} = 0.005$ even though the scheme would be stable with a larger time-step (as large as 0.045). A depiction of the vortex ring within our simulations is shown in Fig. 3.

5. Serial code implementations

The serial codes in NLSEmagic are included for two reasons. One, they allow calculations of the speedup of the CUDA codes and validate their output. Second, the MEX serial codes are quite usable on their own, as they are very simple to install/compile and run, and, as shown in Section 5.3, are much faster than their equivalent MATLAB script codes. Script integrators are also included as they are useful as a platform for testing new numerical schemes and boundary conditions.

All NLSE integrators in NLSEmagic are set-up to simulate a 'chunk' of time-steps of the NLSE and return the resulting solution Ψ. The 'chunk-size' is set based on the desired number of 'frames' (i.e. the number of times Ψ is accessed for plotting and analysis). For the serial integrators, this chunk-based approach is not necessary, but as we will see in Section 7.1, the chunk-based approach is very important for the CUDA integrators.

5.1. MATLAB script code integrators

The script implementations of the NLSE integrators are used to quickly develop and test new methods, and for comparison of the speedup of the MEX integrators. They are split into two script files. The main driver scripts (`NLSE*D.m`) compute the chunk of time-steps in a `for` loop, while the computations of $F(\Psi)$ with the desired boundary conditions in the RK4 scheme of Eq. (3) are done by calling a separate script file (`NLSE*D_F.m`). All of the scripts utilize MATLAB vectorized operations wherever possible for efficiency (i.e. using `A+B;` instead of `for i=1:N; A(i)+B(i); end;`). The script integrators are very straight-forward implementations of the numerical schemes and have been validated by numerous simulations of known solutions to the NLSE and LSE.

5.2. MATLAB MEX code integrators

When using a MEX file, it is important to handle complex values correctly. Since MEX files are written in C, one must either use a complex-number structure, or split the real and imaginary parts of the variables into separate vectors and write the numerical schemes accordingly. As shown in Ref. [9], for CUDA codes, using complex structures is less efficient than splitting the real and imaginary parts directly. To keep the serial and CUDA codes as similar to each other as possible, the split version of the numerical schemes are used. Writing $F(\Psi)$ of Eq. (3) in this way yields

$$\begin{aligned} F(\Psi)^R_{i,j,k} &= -a\,\nabla^2\Psi^I_{i,j,k} - s\left[\left(\Psi^R_{i,j,k}\right)^2 + \left(\Psi^I_{i,j,k}\right)^2\right]\Psi^I_{i,j,k} + V_{i,j,k}\,\Psi^I_{i,j,k} \\ F(\Psi)^I_{i,j,k} &= \ \ a\,\nabla^2\Psi^R_{i,j,k} + s\left[\left(\Psi^R_{i,j,k}\right)^2 + \left(\Psi^I_{i,j,k}\right)^2\right]\Psi^R_{i,j,k} - V_{i,j,k}\,\Psi^R_{i,j,k}, \end{aligned} \tag{23}$$

where Ψ^R and Ψ^I are the real and imaginary parts of the solution respectively. Additional operations are straight-forward (for the split version of the MSD boundary condition, see Ref. [28]).

As mentioned, a MEX code in C is very similar to a regular C code, but uses MATLAB-specific interfaces for input and output data, and special MATLAB versions of memory allocation functions. Instead of having a `main()` function as in standard C, the primary program is

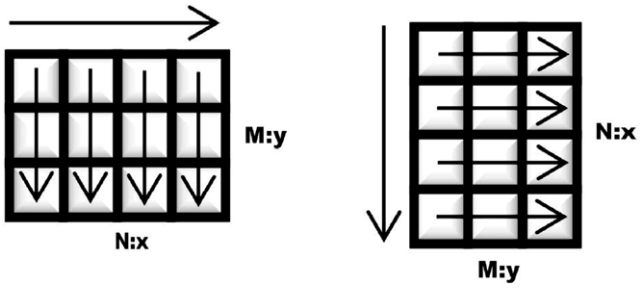

Fig. 4. Representation of a two-dimensional array in MATLAB (left) and in C (MEX) (right). The arrows indicate the linear access pattern. The x and y dimensions shown are the x and y dimensions of the solution Ψ.

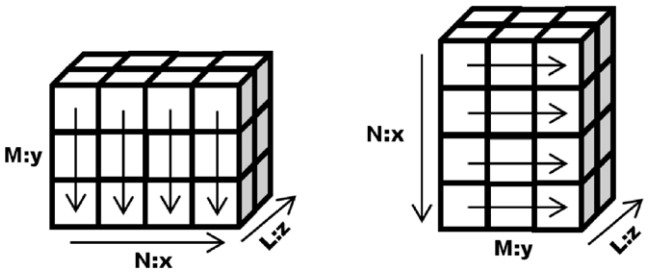

Fig. 5. Representation of a three-dimensional array in MATLAB (left) and in C (MEX) (right). The arrows indicate the linear access pattern. The x, y and z dimensions shown are the x, y and z dimensions of the solution Ψ. Here, the z dimension maintains its orientation in the access pattern, while the x and y dimensions are transposed.

within a function called `mexFunction(nlhs, plhs[],nrhs, prhs[])`. The pointer `prhs[]` (pointer-right-hand-side) gives the C code access to the input data being sent from MATLAB, where `nrhs` is the number of input variables (including arrays and scalars). The pointer `plhs[]` (pointer-left-hand-side) is used in allocating memory for output arrays and scalars, allowing the MEX code to return data to MATLAB.

Special MEX functions (`mxGetPr()`, `mxGetPi()` and `mxGetScalar()`) are used to extract the data from the `prhs[]` array of MATLAB inputs. The function `mxGetScalar()` is used to extract a scalar value while the functions `mxGetPr()` and `mxGetPi()` retrieve the pointer to the real and imaginary part of the selected input array respectively. If the input array is fully real (which can be often the case, especially for initial conditions), then the `mxGetPi()` is null-valued which causes a segmentation fault if accessed. To get around this problem, code has been added to the NLSEmagic integrators that checks to see if there is an imaginary part of the input array of Ψ (using the function `mxIsComplex()`), and if not, allocates an all-zero imaginary array manually (which is then freed at the end of the MEX code).

The MEX functions `mxCreateDoubleMatrix()` (for 1D and 2D arrays) and `mxCreateNumericArray()` (for higher dimensionality arrays) are used to allocate memory space for output arrays which are then pointed to by `plhs[i]`. This memory is then seen as a one-dimensional array by the C code, but in MATLAB will be the desired dimensionality.

In order to use the input and output arrays in C, proper indexing is essential. The MEX functions `mxGetN()` and `mxGetM()` are used to get the dimensions of one-dimensional and two-dimensional arrays (see below for three-dimensional array handling). However, because MATLAB stores arrays column-order, while C stores them row-order, the M value obtained from `mxGetM()` in the C code is actually the number of columns, while the N value obtained from `mxGetN()` is the number of rows (see Fig. 4). Therefore, the input array is seen by the MEX file as transposed from the way it is seen in MATLAB (of course no actual transpose operation is performed in the MATLAB to C passing of arrays, only the order in which the different languages access the elements is different). Thus, while in MATLAB, the column length of the Ψ solution array is the number of grid points in the x-direction of Ψ and the row length is the number in the y-direction, inside the MEX file, this is reversed. (As will be seen in Section 6, this makes the 'x' CUDA grid dimension to be along the y-direction of the solution and vice-versa.) The difference between the row-order access of C and the column-order access of MATLAB must be taken into account to correctly access the elements in an input array (for our discussion here, denoted A). To access array element (i, j) (where $i \in [0, N-1]$ and $j \in [0, M-1]$) of array A in the MEX file, one uses `A[M*i + j]`, where M is the number of columns obtained from `mxGetM()`.

In three dimensions, the MEX functions `mxGetNumberOfDimensions()` and its counterpart `mxGetDimensions()` are used to get the dimensions of the three-dimensional input arrays. The `mxGetDimensions()` returns an array of dimension lengths which, in our codes, are stored as L, M, and N. As in the two-dimensional case, the access pattern differs in MATLAB and C. Once again, the M and N dimensions of the MATLAB array are seen by the MEX file as swapped (see Fig. 5). Thus, in Section 6, the CUDA grid dimensions for x will be along the y-direction and vice-versa but the z grid dimension will correspond to the z-direction of the solution. To access array element (i, j, k) (where $i \in [0, L-1]$, $j \in [0, N-1]$, and $k \in [0, M-1]$) of array A in the MEX file, one uses `A[M*N*i + M*j + k]`.

Within a MEX file, memory allocations and deallocations can be performed using MEX comparable functions to the C functions `malloc()` and `free()` called `mxMalloc()` and `mxFree()`. The MEX function `mxMalloc()` allocates the desired memory in a MATLAB memory-managed heap. The advantage of using `mxMalloc()` instead of `malloc()` is that if a MEX file fails, or is interrupted during its execution, MATLAB can handle and free the memory to better recover from the failure [33].

The compilation of a MEX file is performed within MATLAB using the `mex` command as `mex mymexcode.c`. The MEX compiler uses whatever C compiler on the current machine is selected by MATLAB. By default, MATLAB uses an included LCC compiler, but this can be changed using the `mex -setup` command in MATLAB. For the CUDA MEX codes on Windows, the compiler must be set to Microsoft Visual C++ while on Linux, the standard GCC compiler can be used (for our serial MEX codes, we also use these compiler options). After a MEX file

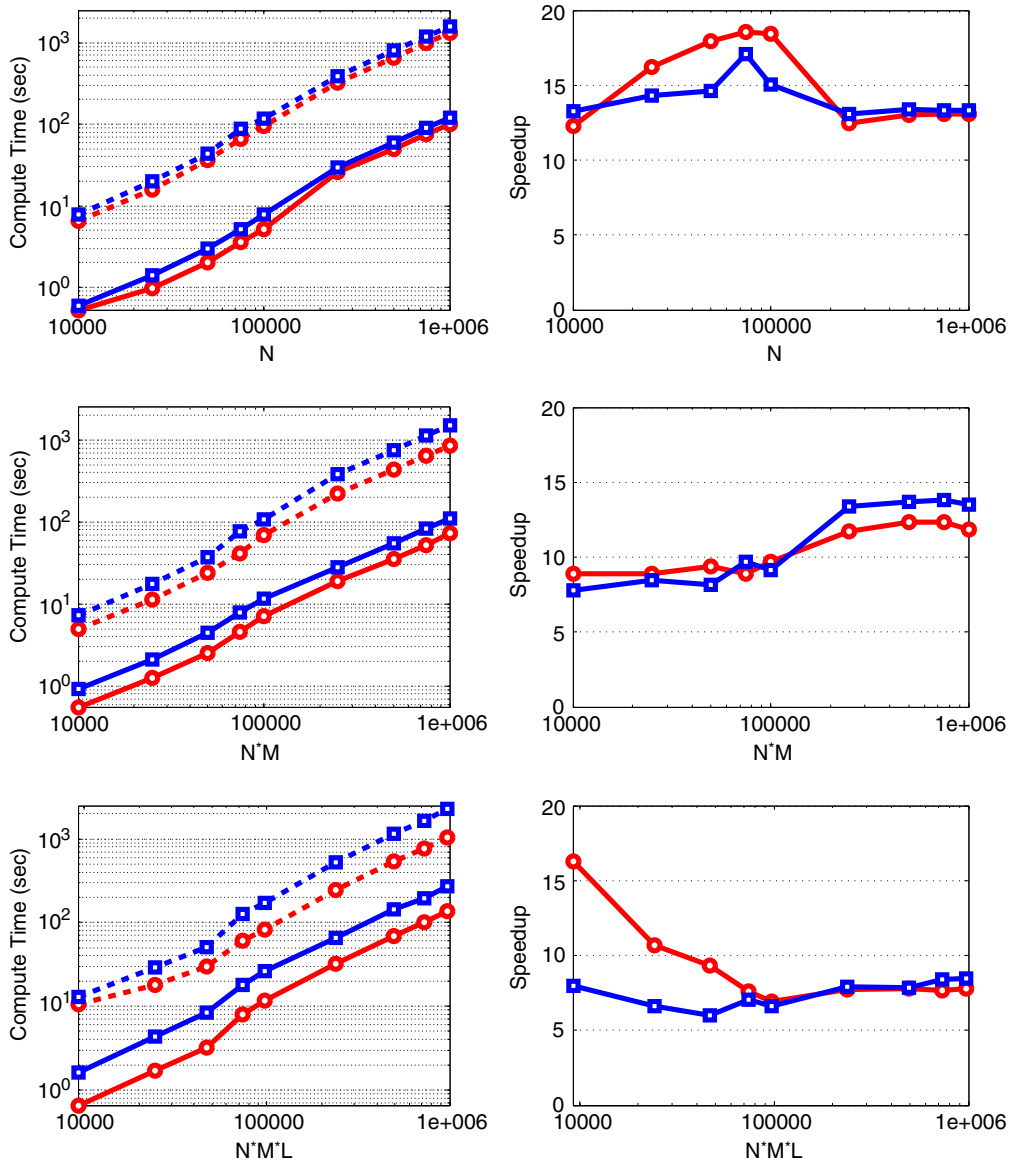

Fig. 6. (Color online) Timing results comparing script (dashed lines) and C MEX (solid lines) NLSE double-precision integrators for both the RK4 + CD (red circles) and RK4 + 2SHOC (blue squares) schemes using the example problems of Section 4 with an end-time of $t = 5$ and a number of frames of 10. The simulations are performed for a total number of grid points of $\mathbf{N} = 10{,}000$ to $\mathbf{N} = 1{,}000{,}000$. The results for the one-, two-, and three-dimensional codes are shown top to bottom. The left column displays the simulation times and the right column shows the corresponding speedup of the MEX codes.

is compiled, the resulting binary file has a special extension which depends on the operating system being used (`.mexw32` and `.mexw64` for 32-bit and 64-bit Windows respectively, and `.mexglx` and `.mexa64` for 32-bit and 64-bit Linux respectively).

Both single and double precision versions of the NLSEmagic serial MEX integrator codes are included in order to properly compare the single-precision versions of the CUDA MEX codes to their serial counterparts (see Section 7). The single precision MEX codes convert the input arrays using the cast `(float)`, and then convert the output arrays back to double precision (MATLAB's native format) after the computation of the desired number of time-steps.

5.3. Speedup of serial MEX codes

To illustrate the advantage of using MEX codes over script codes in MATLAB to integrate the NLSE, we show timing results comparing the performance of the script integrators of NLSEmagic to the equivalent MEX codes using the example problems described in Section 4. As mentioned in Section 5.1, our script codes are sufficiently optimized by utilizing MATLAB's built-in vector operations wherever possible. We set an end-time of $t = 5$, the number of frames to 10, and use double precision for all simulations as it is MATLAB's native format for the script codes. The simulations are performed for a total number of grid points of $\mathbf{N} = 10{,}000$ to $\mathbf{N} = 1{,}000{,}000$. The results are shown in Fig. 6. We see that using MEX codes are, on average, eight to fifteen times faster than the equivalent script codes. Although there is variation in the speedup (more for lower dimensions and variable over problem size), these variations are small and therefore the speedup of the MEX integrators can be considered approximately constant for all dimensions and problem sizes. Therefore, the serial MEX codes included in NLSEmagic are valuable in their own right (especially for researchers who are currently using script codes) since no special setup is needed to compile them and their use is straight-forward.

6. GPU-accelerated CUDA MEX code integrators

In this section we describe in detail the design of the CUDA MEX NLSE integrators in NLSEmagic. Both single and double precision versions of all integrators are produced in order for the codes to be compatible with older GPU cards, as well as to increase performance when double precision accuracy is not required (as will be shown in Section 7, the single precision GPU codes can be much faster than the double precision codes).

As mentioned in Section 2.2, there are two main sections of a CUDA code, the host code and the kernel codes. The host code is run on the CPU and is used to set up the problem, handle memory allocations and transfers to the GPU, and run the GPU kernel code. The kernels are the routines that are run on the GPU.

The basic outline of the integrators is that the input solution Ψ and potential $V(\mathbf{r})$ arrays obtained through the MEX interface are transferred to the GPU's memory. Then, the desired number of time-steps are computed on the GPU using kernel functions, and the resulting solution is transferred back to the CPU memory, where it is outputted to MATLAB through the MEX interface. Since the memory transfers from CPU to GPU and vice-versa are quite slow, the larger the number of time-steps the GPU computes before sending the solution back to the CPU (i.e. the larger the chunk-size), the better the expected speedup. In Section 7.1 we investigate this in detail for our simulations.

6.1. Specific code design for all dimensions

We now discuss the details of the CUDA implementation which are the same for the one-, two-, and three-dimensional integrator codes (in the sections that follow, we will discuss dimension-specific code design). We focus on the CUDA design aspects of the codes, and therefore do not mention any MEX interfaces (as they are all equivalent to the serial MEX codes previously described). Thus, we assume that the solution Ψ and potential $V(\mathbf{r})$ have already been obtained through the MEX interface along with all scalar parameters.

The first step in the code is to allocate arrays on the GPU's global memory to store the solution, potential, and RK4 temporary arrays. This is accomplished by using the `cudaMalloc()` or the `cudaMallocPitch()` (see Section 6.3) function. The next step in the code is to transfer the solution Ψ and potential $V(\mathbf{r})$ arrays from the CPU's memory to the memory locations allocated for them on the GPU's global memory. This is done through the `cudaMemcpy()` or the `cudaMemcpy2D()` (see Section 6.3) function.

Before calling the compute kernel routines to integrate the NLSE, it is first necessary to define the CUDA compute-grid and block sizes for use in the kernel calls. The choice of block size is not trivial, and is dependent on the size of the problem and the GPU card being used. In order for a GPU to be most efficient, it requires that the problem have good *occupancy* on the specific card. This means that as many SMs are running as many SPs during the program simultaneously. In NLSEmagic, the block sizes are chosen to provide efficiency for typical sized problems in each dimension (see the following sections for details). The block size in each dimension is set using the `dimBlock()` function.

The compute-grid size is found by taking the `ceil` of the length of the dimension of the specified problem array divided by the block size of that dimension. For example, the x-direction grid size in two dimensions would be found by `ceil(M/dimBlock.x)` where `dimBlock.x` is the x-direction block-size (it is important to note that, as per the discussion in Section 5.2, the x- and y-direction of the CUDA blocks are actually along the y- and x-direction respectively of the solution Ψ within the C code). Once the values for the grid dimensions are found, the grid is set up using the function `dimGrid()`.

Once the CUDA compute-grid and block sizes are set up, the code computes a chunk-size number of time-steps of the RK4 scheme using kernels which run on the GPU. The more computations a single kernel call performs, the better the performance of the GPU code. Therefore, we try to combine as many steps in the RK4 algorithm of Eq. (3) as possible. Since each compute block must be able to run independently of all the other blocks within a single kernel call, synchronization and race conditions will limit how many steps of the RK4 can be combined. For example, each computation of $F(\Psi)$ in Eq. (3) (after the first one), for each grid point, requires the previously completed computation of Ψ_{tmp} of the neighboring points from the previous step in order to compute the Laplacian of Ψ. Therefore, even with block-wide synchronizations, grid points along the boundary of compute-blocks will have no guarantee that the Ψ_{tmp} values it needs will have been computed yet or not (or possibly, even overwritten). Therefore, multiple separate kernels are required to run the steps in Eq. (3). When using the RK4 + 2SHOC scheme, an additional kernel (called four times) is required due to the same synchronization issues involving neighboring points needed from one step to the other (in this case the values of the Laplacian D in the first step of the 2SHOC scheme).

Although the computations of $F(\Psi)$ and D require separate kernels, the other steps in the RK4 algorithm *can* be combined with the kernels for $F(\Psi)$. Thus, the RK4 can be computed using only four kernel calls (8 when the additional kernel needed in the 2SHOC scheme is included) which contain steps 1–2, 3–5, 6–8, and 9–10 of Eq. (3). The only problem is that steps 3–5 and 6–8 can cause a synchronization issue. This is because step 5 (and 8) overwrites the Ψ_{tmp} values, which are the inputs to step 3 (and 6). Thus, when one block of threads finishes the step 3–5 computation, the adjacent block could still require the old values of Ψ_{tmp} along its neighboring edge to compute the Laplacian. To solve this, we introduce a storage array called Ψ_{out} which allows step 5 to output its values to Ψ_{out} so that the other blocks still have access to the original values of Ψ_{tmp}. Then in step 6, the code evaluates $F(\Psi_{\text{out}})$, while step 8 outputs to Ψ_{tmp}. Adding this vector does not increase the overall global memory storage requirements of the RK4 scheme because since the steps are now combined into four kernels, the K_{tmp} array no longer needs to be stored in global memory and can instead be stored in the per-block shared memory. The RK4 algorithm can therefore be described on the GPU as

$$
\begin{aligned}
&(1)\ k_{\text{tmp}} = F(\Psi) && (7)\ k_{\text{tmp}} = F(\Psi_{\mathbf{out}})\\
&(2)\ K_{\text{tot}} = k_{\text{tmp}} && (8)\ K_{\text{tot}} = K_{\text{tot}} + 2\,k_{\text{tmp}}\\
&(3)\ \Psi_{\text{tmp}} = \Psi + \frac{\mathbf{k}}{2}\,k_{\text{tmp}} && (9)\ \Psi_{\text{tmp}} = \Psi + \mathbf{k}\,k_{\text{tmp}}\\
&(4)\ k_{\text{tmp}} = F(\Psi_{\text{tmp}}) && (10)\ k_{\text{tmp}} = F(\Psi_{\text{tmp}})\\
&(5)\ K_{\text{tot}} = K_{\text{tot}} + 2\,k_{\text{tmp}} && (11)\ \Psi = \Psi + \frac{\mathbf{k}}{6}\,(K_{\text{tot}} + k_{\text{tmp}}),\\
&(6)\ \Psi_{\mathbf{out}} = \Psi + \frac{\mathbf{k}}{2}\,k_{\text{tmp}},
\end{aligned}
\tag{24}
$$

where k_{tmp} is only stored in shared memory, while Ψ, K_{tot}, Ψ_{tmp}, and Ψ_{out} are stored in global memory (although during computation, they are transferred to shared memory as discussed below).

All four kernel calls in the computation of a RK4 step are computed using a single kernel function named `compute_F()`. One of its input parameters tells the kernel which step it is computing, and the corresponding computation is selected in a switch statement. For the 2SHOC scheme, an additional kernel is written named `compute_D()` which computes the D array before each call to `compute_F()`.

As discussed in Section 2.1, thread accesses to the per-block shared memory are much faster than accesses to the global memory of the GPU. Therefore if any array needed in the computations is accessed more than once by a thread in the block, it is worthwhile to copy the block's required values of the array from global memory into shared memory. Then, after computing using the shared memory, the results can be copied back into global memory. Since shared memory is block-based, each thread in the block needs to copy its own value from the global array into the shared memory space (some threads may need to copy more than one value as we will show in Sections 6.2–6.4).

In `compute_F()`, five shared memory arrays are required (seven for the 2SHOC scheme). These consist of the real and imaginary parts of the Ψ input and the $F(\Psi)$ result (called k_{tmp} in the RK4 algorithm of Eq. (24)), as well as the potential array V. In the 2SHOC scheme, the real and imaginary parts of D are also stored in shared memory arrays. In the `compute_D()` kernel for the 2SHOC scheme, only two shared memory arrays are needed (for Ψ).

Each thread in the block copies the global memory values into the shared memory arrays. Since the number of stream processors in the stream multi-processor that the block is being computed on almost always has less processors than the number of threads in the block, threads will typically not have access to all of the shared memory of the block after copying their own values. This is a problem because each thread has to access neighbor elements in the shared memory block for computation of the Laplacian of Ψ. Therefore, a `__syncthreads()` function is called after the copy which synchronizes all the threads in the block, ensuring that the entire shared-memory array is filled before using it.

After the synchronization, the threads on the boundary of the block have to copy additional values into shared memory since they require accessing points which are beyond the block boundary due to the finite-difference stencil (see Sections 6.2–6.4 for details in each dimension). These transfers are not done before the block-synchronization because in CUDA, when a group of threads all need to perform the same memory transfer, they are able to do it in a single memory copy instead of one-by-one. Adding the boundary transfers before the synchronization may break up this pattern and cause the memory-copies not to be aligned for single-instruction copying.

After the transfers to shared memory are completed, the $F(\Psi)$ values are computed for all grid points within the boundaries of the solution Ψ. Threads which happen to be on the boundary of Ψ compute the boundary conditions of $F(\Psi)$. When using the MSD boundary condition of Ref. [28], a block-synchronization is required to be sure that the interior value of $F(\Psi)$ has been pre-computed (since it is necessary for the computation of the MSD condition). A problem exists if in any direction, the block is only one cell wide (since the interior point of $F(\Psi)$ will not be computed by that block). To avoid this problem in the most efficient way (i.e, without adding extra error-checking code), we leave the detection and solution of this issue to the NLSEmagic driver scripts which automatically adjust the solution grid size to ensure that this condition does not occur.

Once the boundary conditions of the $F(\Psi)$ array is completed, the kernel uses the result to compute the remaining sub-steps of the RK4 algorithm. After the desired number of RK4 time-steps (the chunk-size) have been completed, a call to `cudaDeviceSynchronize()` is made which ensures that all kernels are fully completed before continuing in the host code. This is required because although the CUDA GPU-to-CPU memory copies are designed to be implicitly synchronizing, in practice for large problems, we found that this was not completely reliable.

The current Ψ arrays are then transferred to the CPU using the `cudaMemcpy()` (or `cudaMemcpy2D()`) function. The next step is to free all the CUDA global memory spaces used (with `cudaFree()`) and reset the device with `cudaDeviceReset()`). Typically one would not reset the device in a CUDA code, but due to some problems with MATLAB memory management on Linux platforms, this step is necessary and does not have any noticeable impact on performance. Finally, the MEX file then returns the new value of Ψ back to MATLAB.

6.2. One-dimensional specific code design

In the one-dimensional NLSEmagic integrators, the CUDA compute-grid is set-up to be one-dimensional with a block size of 512 threads. Although for small problems, a block size of 512 is not efficient in terms of occupancy (see Section 6), we feel that since one-dimensional problems typically run fast enough without any GPU-acceleration, the only time GPU-acceleration will be needed in one-dimensional problems is when the problem size is very large. In such a case, a block size of 512 will allow for good occupancy in most situations.

One of the key concerns in parallelizing finite-difference codes is the need for the cells on the edge of a compute-block to require values from their neighbor in another block. Since in the CUDA codes all threads in all blocks have equal access to global memory, there seems to be no problem with threads on the block boundaries. However, this is only true if the CUDA code only uses global memory (see Refs. [8,9], where the authors take this approach).

In order to greatly speedup the codes, it is vital to use the per-block shared memory in the computation whenever any global memory space is needed to be accessed by any thread more than once. The typical way to use shared memory is to have each thread in the block copy one value of Ψ (or D) and V from global memory into shared memory arrays which are the same size as the block size. Then the computation is done using the shared memory array, and at the end each thread stores the output back into the global memory array. However, in this scenario, the block boundary cells do not have all their required Ψ (or D) values available in shared memory. There are a few different ways to deal with this. A simple solution is to have the block boundary threads directly access the required neighbor point(s) of Ψ (or D) from global memory when needed (as was done in Ref. [6]). Another solution is to set up the grid to overlap the boundary cells so that all threads copy values into shared memory as before, but only the interior threads of the block perform the computations, which can be performed using only shared memory. Through testing, we have found that this solution is not efficient because a large number of threads (the boundary threads) are idle after the memory copy.

Another solution is that instead of having the shared memory size be equal to the block size, the shared-memory size is allocated to be two cells greater than the block size. In the stage of the kernel where the threads transfer Ψ and V from global memory into shared memory, the boundary threads also transfer the neighboring points into the extra cell space at the boundary of shared memory. Therefore there are two global memory transfers performed by the block boundary cells instead of one. This process is depicted in Fig. 7. Later, in

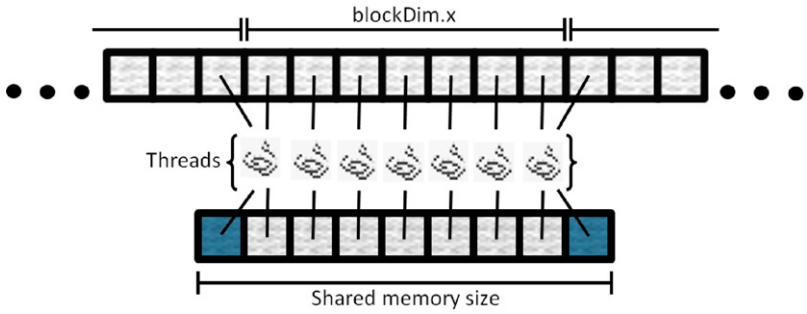

Fig. 7. (Color online) One-dimensional shared memory structure and transfer of global memory to shared memory in NLSEmagic. Note that the threads on the boundary of the block must perform two global memory retrievals into shared memory.

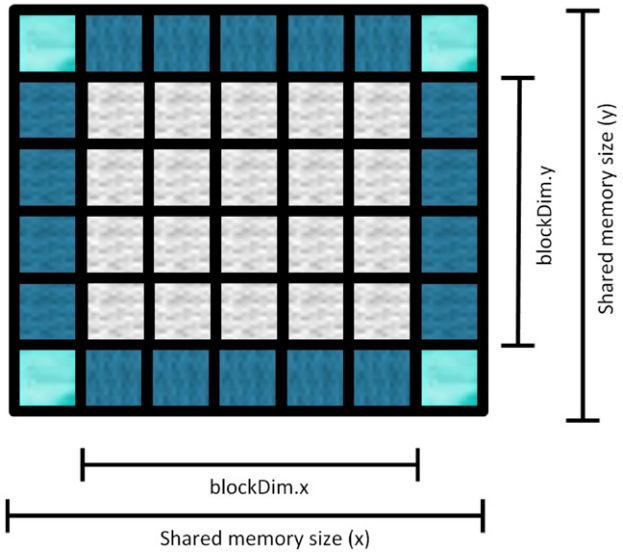

Fig. 8. (Color online) Two-dimensional shared memory structure. The threads on the boundary of the block must perform additional global memory retrievals into shared memory. Those on the corners perform a total of three transfers for the CD scheme, and four transfers for the 2SHOC (due to the required corner cells in the 2SHOC scheme). All other boundary threads perform two total transfers.

the computation part of the kernel, all threads can then compute using shared memory. Although this seems to be equivalent to the first method in terms of the number of global memory accesses (since in the first solution, the boundary cells access global memory twice as well — once to transfer their value to shared memory and once to access their required neighboring point), in practice, due to intricacies of the card hardware and software execution (for example, limiting the number of divergent branches), this method can be more efficient overall and therefore the one adopted in our codes.

6.3. Two-dimensional specific code design

In two-dimensional CUDA codes, the CUDA API provides special global memory allocation and transfer routines called `cudaMallocPitch()` and `cudaMemcpy2D()`. The `cudaMallocPitch()` allocates a global memory space which is expected to be accessed in a two-dimensional pattern. It aligns the data with the card hardware, often padding the rows of the matrix. Therefore, the memory allocation function returns a pitch value which is used to access the element (i, j) as `pitch*i+j` instead of the usual `M*i+j`. The pitch value is also used in the `cudaMemcpy2D()` to transfer data from the CPU to the GPU and vice-versa. The special two-dimensional routines are somewhat optional as one could also just use a standard linear memory transfer of the data and access the elements normally, but NVIDIA strongly recommends against it [15]. Therefore, the two-dimensional integrators of NLSEmagic use the specialized functions.

The CUDA compute-grid is set to be two-dimensional with two-dimensional blocks with a block size of 16×16. Based on performance tests, this is the best overall block size to use even though newer cards would allow a larger block size.

As in the one-dimensional code, the shared memory space is allocated to be two cells larger in width and height of the block size as shown in Fig. 8. The boundary threads once again grab the neighboring points in addition to their designated points from global memory during the shared-memory transfer process. For the CD scheme, the corner neighbor points are not needed, but for the 2SHOC scheme they are. This adds an additional global memory access on the corner threads (in addition the extra accesses from the other two neighboring points), making the corner threads copy up to four global values per array into shared memory.

When dividing the problem into CUDA blocks, the CUDA thread access ordering is an important consideration. The CUDA grid and block indexing is column-major ordered, with the 'x' direction along the columns and the 'y' direction is along the rows, and the threads are scheduled in that order. Thus, when copying arrays from global memory to shared memory, it is important that the access pattern of the memory being copied matches the access pattern of the thread scheduler [34]. Since C arrays are row-major, a shared memory array

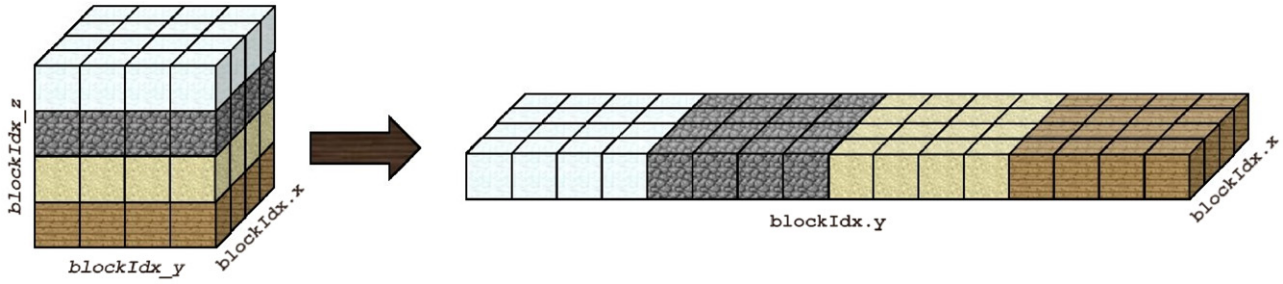

Fig. 9. (Color online) Logical block reordering for a 2D CUDA grid from the desired 3D grid structure.

`A` should be allocated (counter-intuitively) as `A[BLOCK_SIZEY][BLOCK_SIZEX]`. The memory copy from elements in a global array `A_global` is then given as

```
A[threadIdx.y][threadIdx.x] = A_global[pitch*i+j],
```

where the position of the current thread within the solution array is given by

```
j = blockIdx.x*blockDim.x + threadIdx.x,
i = blockIdx.y*blockDim.y + threadIdx.y.
```

Since in MATLAB (when using `meshgrid()` to form the x and y arrays of Ψ) the x direction of Ψ is along the rows of the Ψ array, and the y direction is along the columns, and when in the MEX file, this is transposed, then, as mentioned in Section 5.2, the 'x' direction of the CUDA grid and blocks actually represent the y direction in Ψ and vice-versa. Therefore, in defining the grid size, we use `M/dimBlock.x` for the x dimension of the grid and `N/dimBlock.y` for the y even though `N` is the x-direction of Ψ and `M` is the y-direction. This is very important to remember when checking the MSD boundary condition grid requirement mentioned in Section 6.1. The thread access ordering described is not a trivial matter. In our tests, using the incorrect ordering can slow down the codes by almost a half!

6.4. Three-dimensional specific code design

In three dimensions, the CUDA API provides a hardware-optimized memory allocation and transfer functions similar to the two-dimensional ones mentioned in Section 6.3. However, the implementation of the three-dimensional functions (called `cudaMalloc3D()` and `cudaMemcpy3D()`) are not straight-forward and requires making structures with information about the arrays, and passing the structures to the functions. Through testing of the two-dimensional NLSEmagic codes versus a naive two-dimensional implementation (without the specialized allocation and memory-copy), we did not observe any significant performance boost in using the hardware-optimized routines. Therefore, in order to keep things as simple as possible, the three-dimensional integrators of NLSEmagic use the standard `cudaMalloc()` and `cudaMemcpy()` used in the one-dimensional codes in which case a point (i, j, k) in a global memory array `A_global` is simply given by `A_global[N*M*i + M*j + k]`.

In Refs. [7,8], the three-dimensional finite-difference CUDA codes were set-up to do multiple two-dimensional sweeps and combine the results to form the three-dimensional derivatives. Thus, only two-dimensional grid and block structures were used. This was done because the limit of 16 KB of shared memory of the older GPUs was felt to be too limiting for using three-dimensional blocks. However, in our NLSE integrators, we use three-dimensional blocks which we note is efficient even when using older Tesla GPUs. On the Fermi cards, we have tested our code for various block sizes and have used the current most efficient block sizes for a typical problem size depending on the numerical method and precision being used.

An inherent difficulty with three-dimensional CUDA codes is that originally, even though blocks could be three-dimensional, the CUDA compute-grid could only be two-dimensional. Thus, one could not implement three-dimensional codes in a straight-forward extension from two-dimensional grid structures. This is an additional reason why previous studies (such as Refs. [7,8]) used two-dimensional slice algorithms.

NVIDIA has solved this problem as of the release of the CUDA SDK 4.0. With the new SDK, all Fermi based GPUs and higher can now have a full three-dimensional grid structure. Our NLSE integrators in NLSEmagic were developed before this release, and therefore did not take advantage of this new feature. Instead, a custom logical three-dimensional structure on a two-dimensional CUDA compute-grid was used. In order to allow NLSEmagic to be compatible with older Tesla-based GPUs, the integrators are not currently updated to use the new three-dimensional grids.

The three-dimensional problem grid is viewed as a series of three-dimensional slabs, each being one block high. These slabs are then seen as being placed side-by-side to create a two-dimensional grid of three-dimensional blocks as shown in Fig. 9. We thus have a CUDA compute-grid which is two-dimensional with three-dimensional blocks, and a 'virtual' compute-grid which is a three-dimensional grid with three-dimensional blocks. In order to make the indexing as easy as possible within the actual CUDA compute-grid, we create new indexing variables per thread in order to use the full three-dimensional virtual compute-grid.

We note here that, as in the two-dimensional case, the CUDA block/thread directions x, y, and z do not all correspond to those of Ψ. The CUDA block x-direction is actually along the y-direction of Ψ (M) and the CUDA block y-direction is actually along the x-direction of Ψ (N). The z-direction of both are equivalent. This is important to keep in mind for the computation of grid sizes and in checking the block requirements of the MSD boundary-condition.

The first step in setting up the three-dimensional grid is that a variable

```
gridDim_y = ceil(N/dimBlock.y)
```

is set to represent the virtual y-direction grid size. Therefore, the actual CUDA grid is defined to be

```
dimGrid(ceil(M/dimBlock.x),gridDim_y*ceil(L/dimBlock.z)).
```

The value of `gridDim_y` is passed into the kernels in order for the threads to have access to it. Inside the kernel, a new variable representing the thread's z-direction block position within the virtual three-dimensional grid called `blockIdx_z` is created defined as

```
blockIdx_z = blockIdx.y/gridDim_y,
```

which uses integer division to act as a `floor` function. The y-direction block position in the virtual grid is then found and stored in a new variable as

```
blockIdx_y = blockIdx.y - blockIdx_z*gridDim_y.
```

The `blockIdx_y` variable is therefore replacing the intrinsic `blockIdx.y` variable (which would give the y-direction block within the true CUDA grid). These new block position variables are depicted in Fig. 9.

Once the new indexing variables are created, the computation of the thread's designated position within the solution array is straightforward as if there were a true CUDA three-dimensional grid. The value of the thread's indexes are

```
k = blockIdx.x*blockDim.x + threadIdx.x,
j = blockIdx_y*blockDim.y + threadIdx.y,
i = blockIdx_z*blockDim.z + threadIdx.z,
```

where the thread is associated with the global memory position `A_global[N*M*i + M*j + k]`.

Once the older Tesla GPU cards are no longer in general use, direct three-dimensional CUDA grids can be implemented into the code, which, for our implementation, would be very straight-forward (just setting up the grid based on M, N, and L directly and replacing the new thread indexes with the intrinsic ones).

As was the case in the two-dimensional codes, the dimensions of the shared memory space must be done in accordance with the thread access patterns in order to be efficient. (Note that this discussion is completely independent of the implementation of the three-dimensional compute-grid since, either way, three-dimensional blocks are being used and this issue is a block-wide problem.) Since the threads are accessed in column order, first along x, then y, then z, it is important that the shared memory space be allocated as `A[BLOCK_SIZEZ][BLOCK_SIZEY][BLOCK_SIZEX]` so that the memory copy is done as

```
A[threadIdx.z][threadIdx.y][threadIdx.x] = A_global[N*M*i + M*j + k].
```

As in the two-dimensional case, this correct ordering has a huge effect on performance (upwards of 50%) and is therefore vital.

As in the lower-dimensional integrator codes, the three-dimensional integrators allocate the shared memory space to be two cells larger in each dimension than the block size. Thus, the threads on the block boundaries copy additional values from global memory into shared memory. For each required shared memory array, the interior threads copy one value, the threads along the faces of the block boundaries copy two values, the threads along the edges of the block boundary copy three values (four for Ψ in the 2SHOC scheme), and the threads on the corners of the block boundary have to copy four values (seven for Ψ in the 2SHOC scheme). Since the 2SHOC scheme never relies on the diagonal corner cells of Ψ or D, as it does not use a full 27-points stencil as seen in Eq. (9), no additional copies are necessary. Due to the large number of extra serialized memory copies for the boundary cell threads, the performance of the three-dimensional codes is expected to be less than the two-dimensional codes (this is indeed the case as shown in Section 7.4).

7. Speedup results

Here we show the results of comparing the compute-time of the NLSEmagic CUDA integrators versus the serial MEX integrators. The serial integrators are run on a single core of an Intel Corei3 CPU, and the GPU codes are run on a GeForce GTX 580 card (full specifications of both the CPU and GPU hardware are given in the Appendix). Even though the double precision performance is artificially reduced in the GeForce cards, we show that this only effects our one-dimensional code, whereas our two- and three-dimensional codes have memory bottlenecks which make the reduced double precision performance negligible in practice. (After possible future optimization of the code, this may not be the case and it would be beneficial to run the codes on a Tesla compute-only GPU.) Because the GeForce GPUs are many times cheaper than the Tesla (or Quadro) cards and the performance is not expected to be effected in a major way, we feel justified in running our codes on a GeForce card.

It is important to point out that our goal in the speedup timings is to compare the GPU integrators to a typical serial implementation, and demonstrate their efficiency. As mentioned in the introduction, there are other options available to parallelize a serial code including OpenMP and MPI on desktop multi-core machines, as well as on large high-performance clusters. Our focus is not on how the GPU integrators compare to other parallel implementations (such as the OpenMP code in Ref. [11]), but how they perform compared to a serial implementation. Although not the main focus, to further demonstrate the advantages of the GPU implementation, we give estimates on how the GPU integrators would perform versus the theoretical maximum parallel performance of an OpenMP code run on a quad-core CPU with the same specifications as the single CPU core used in the serial runs.

For all speedup tests, we use the example simulations of Section 4 with an end-time of $t = 50$ and vary the total grid-size from about 1000 to 3,000,000. In each case we record the computation time of the integrators, ignoring the small extra time required for generating the initial condition and outputting results.

Before computing the speedup results, it is first necessary to examine the effect that CPU–GPU memory transfers has on the performance of the integrators.

7.1. Chunk-size limitations

As mentioned in Section 2.1, memory transfers between the CPU and GPU are very slow, and therefore it is best to compute as much as possible on the GPU before transferring the data back. For simulations where analysis results are cumulative, the entire simulation can be performed on the GPU with only two memory transfers (one at the beginning and one at the end of the computation). For most studies of time-dependent problems such as the NLSE, it is desired (or essential) to have access to the computation data at regular intervals in order to save the data, display it for observation and animations, and to run intermediate analysis. However, the more times the data is needed by the CPU, the slower the code will perform. Therefore for NLSEmagic, the more frames the simulation is plotted and analyzed, the slower the overall code will be.

In order to use the codes in the most efficient manner, it is necessary to see how much the simulation is slowed down as a function of the size of the chunk of time-steps performed by the CUDA integrators (which we have designated as the *chunk-size*). Once a chunk-size is found which exhibits acceptable lack of slow-down, the number of frames that the solution is viewable will be determined by the number of time-steps.

To see how the chunk-size affects performance of the NLSEmagic CUDA integrators, we run the examples described in Section 4 for various chunk-sizes and compare the compute-times. Since the performance of the codes due to chunk-size is not affected by the total number of time steps (as long as the number of time steps is larger than several chunk-sizes), we fix the end-time of the simulations to be $t = 5$ and the time-step to be $\mathbf{k} = 0.005$ yielding 1000 time-steps. We run the examples in each dimension with single and double precision for both the RK4 + CD and RK4 + 2SHOC schemes for total grid sizes ($\mathbf{N}$) of around 1000, 10,000, 100,000, and 1,000,000 (the two- and three-dimensional codes have slightly different resolutions due to taking the floor of the square- or cubed-root of $\mathbf{N}$ as the length in each dimension respectively). To eliminate dependency of the results on the specific compute times of the given problem (which would make comparison of different grid sizes difficult), for each chunk-size, we compare the compute times of the simulation to the fastest time of all the simulations of that problem. As expected, this fastest time nearly always occurs when the chunk-size is equal to the number of time-steps. We thus compute a 'slowdown' factor as the time of the fastest run divided by the time of the other runs. This makes the results applicable to almost all simulations run with NLSEmagic. The results are shown in Fig. 10. We see that in each case, the chunk-size has to be somewhat large for the codes to perform with top efficiency. Simulations with lower chunk-sizes can be done with acceptable slowdown based on the figures. In the higher dimensional cases, the larger the total grid size, the less the slowdown factor for a given chunk-size (in the one-dimensional case, the results are varied, with the lowest resolution tested outperforming the highest). Also, in general, it is seen that the higher-dimensional codes have less slowdown for a given chunk-size than lower-dimensional codes. Likewise, using the 2SHOC versus the CD scheme and double versus single precision, lower the slowdown for a given chunk-size as well. This is understandable since the more work the CUDA kernels are doing, the higher percentage of the total time is spent on computations compared to memory transfers to the CPU and back. In either case, when running NLSEmagic, it is best to first look up on Fig. 10 where the problem being simulated falls, and then choose an appropriate chunk-size to get the best performance out of the codes (within the simulation requirements).

7.2. One-dimensional speedup results

For the one-dimensional results, we use the dark soliton solution in Section 4 with an end-time of $t = 50$. The solution is plotted five times during the simulation yielding a chunk-size of 5000, which is well over the efficiency requirements as discussed in Section 7.1. The soliton is simulated with a grid size which varies from $N = 1000$ to $N = 3{,}000{,}000$. Validation of the codes is done by comparing the solution to the known exact solution. The simulation compute-times and speedups compared to the serial MEX integrators are shown in Fig. 11 and Table 1. The best results are those of using the CD + RK4 scheme in single precision where we observed speedups around 90 for the larger grid sizes. Since, as was shown in Fig. 6, the serial integrators are about 12 times faster than the MATLAB script codes for those resolutions, the NLSEmagic CUDA MEX codes for the one-dimensional CD + RK4 scheme in single precision for large resolutions are about *1000 times faster* than the equivalent MATLAB script code. In terms of actual time for the simulation tested, this would equate to taking roughly 30 s using the GPU code, 40 min using the serial MEX code, and 8 *hours* 20 min using the MATLAB script code. Assuming a perfect OpenMP implementation of a quad-core CPU of equivalent specifications, the GPU integrators still achieve a maximum speedup of over 20.

Since the block size for the integrators was chosen to be 512, it is understandable that the compute time for the CUDA MEX codes stays quite low until resolutions of 10,000 or so. This is because the GPU being used has 16 MPs, and therefore resolutions up to 8000 can be computed in one sweep of the GPU MPs, while higher resolutions require multiple blocks to be computed on the same MP.

It is noticeable that the double-precision performance is almost half that of the single precision. This is partly due to what was mentioned in Section 2.1 that the GeForce cards have their double precision FLOP count artificially reduced by three-quarters, making the FLOP count one-eighth that of the single precision performance. In addition, memory transfer is considered to be a large factor in code performance, and since double-precision variables take twice the memory space as single precision, a reduction in performance is understandable.

It is also apparent that the speedup when using the 2SHOC scheme is lower than the CD scheme. This is also understandable because in the 2SHOC scheme, kernels which only compute the D array (the standard second-order Laplacian) are needed, and they have a smaller amount of floating-point operations than the kernels which compute the full NLSE and RK4 step. Since the amount of computations in the D kernel is so small but the amount of required memory transfers is comparable to those of $F(\Psi)$, the speedup is smaller. As will be shown in Sections 7.3 and 7.4, in higher dimensions, this issue is somewhat minimized due to the added number of points in the second-order Laplacian stencil (the speedup reduction is lowered from 27% in one dimension to 16% in three dimensions in single precision, and from 45% to 0% in double precision).

7.3. Two-dimensional speedup results

For the two-dimensional tests, we use the unoptimized steady-state dark vortex approximate solution of Section 4. Since there is no analytical solution to the dark vortex, and moreover, since we are using only an approximation of the true solution, we cannot record the

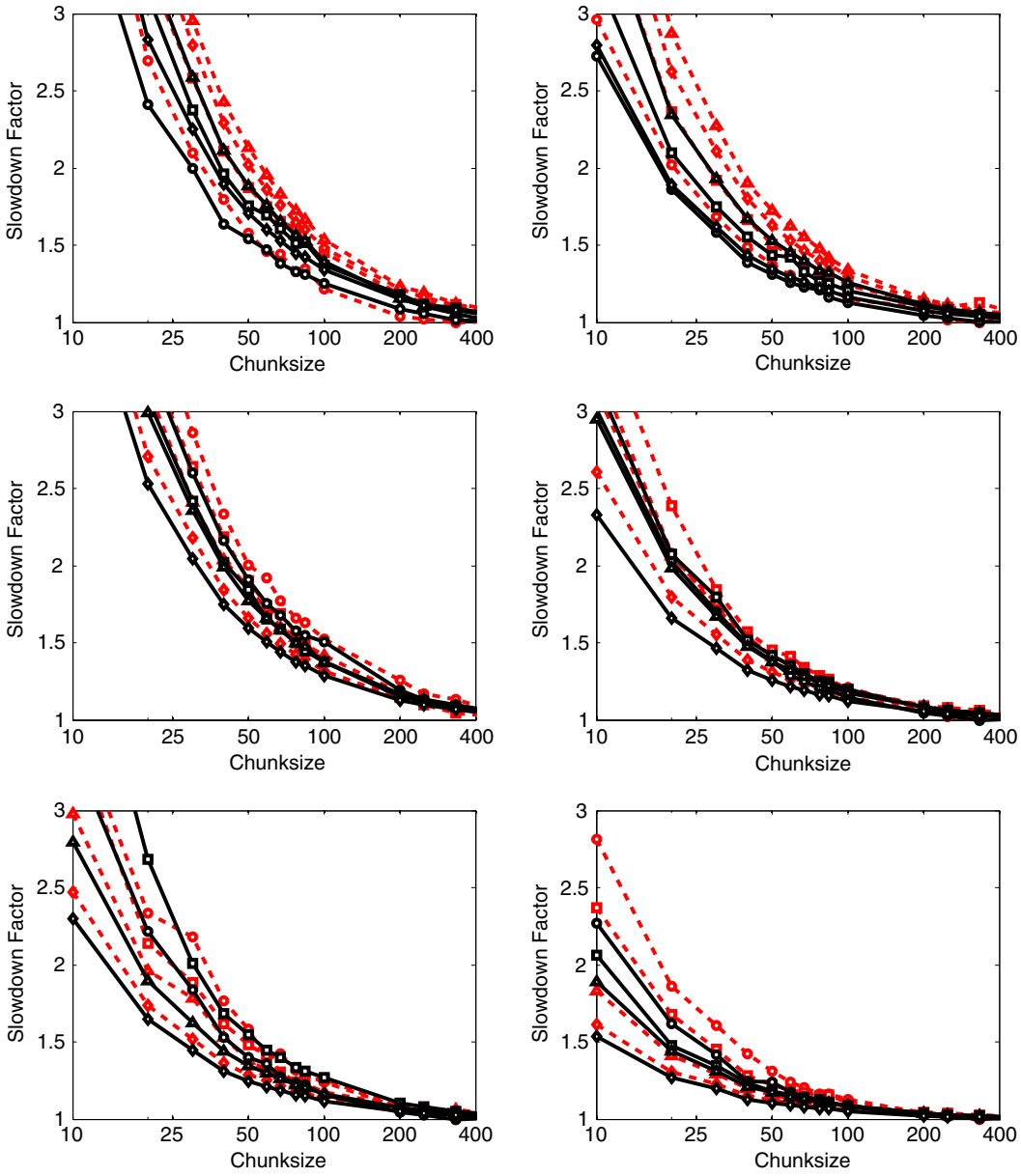

Fig. 10. (Color online) The slowdown factor as a function of chunk-size of the CUDA NLSEmagic integrators computed using the examples of Section 4 with an end-time of $t = 5$ with 1000 time-steps. The slowdown is defined as how much slower the simulation is compared to the fastest possible simulation (where the chunk-size equals the number of time-steps – not shown). The results are shown for the RK4 + CD and RK4 + 2SHOC (left and right respectively), for one-, two-, and three-dimensional simulations (top to bottom respectively), and for single and double precisions (dashed and solid lines, respectively). The circle, square, triangle, and diamond lines represent total grid-sizes **N** of 1000, 10,000, 100,000, and 1,000,000, respectively.

error of the runs. Therefore, in order to validate the simulations, we displayed near-center cell data points for each frame and compared them with those from the output of the non-CUDA MEX codes, and checked that they were equivalent.

Like the one-dimensional tests, the solution is plotted five times during the simulation yielding a chunk-size of 5000, which is well over the efficiency requirements discussed in Section 7.1. The vortex solution is simulated with a total grid size which varies from about $\mathbf{N} = N * M \approx 1000$ to $\mathbf{N} \approx 3{,}000{,}000$. The N and M dimensions are determined by taking the floor of the square-root of **N** and are sometimes slightly altered due to the block requirements of using the MSD boundary condition (see Section 6.1).

The simulation compute-times and speedups compared to the serial MEX integrators are shown in Fig. 12 and Table 2. There is noticeably less speedup in the two-dimensional codes when compared to the speedup of the one-dimensional codes. A possible explanation for the cause of the reduced speedup is that in two dimensions there are many more boundary cells versus interior cells of the CUDA blocks (in one dimension there are $O(1)$ while in a two-dimensional square block, there are $O(N)$). Since, as described in detail in Section 6.3, block cell boundaries require additional global memory accesses (especially the corners), a decrease in performance was excepted. Also, the number of total boundary grid points is much higher in two dimensions, in which case there are more threads computing grid boundary conditions than in one dimension. This can cause less speedup in the codes since the MSD boundary condition requires an extra block-synchronization not needed in the internal scheme. Slightly better speedup results could be obtained by choosing a problem that has Dirichlet boundary conditions, but it is important to show results that apply to a more general range of applications.

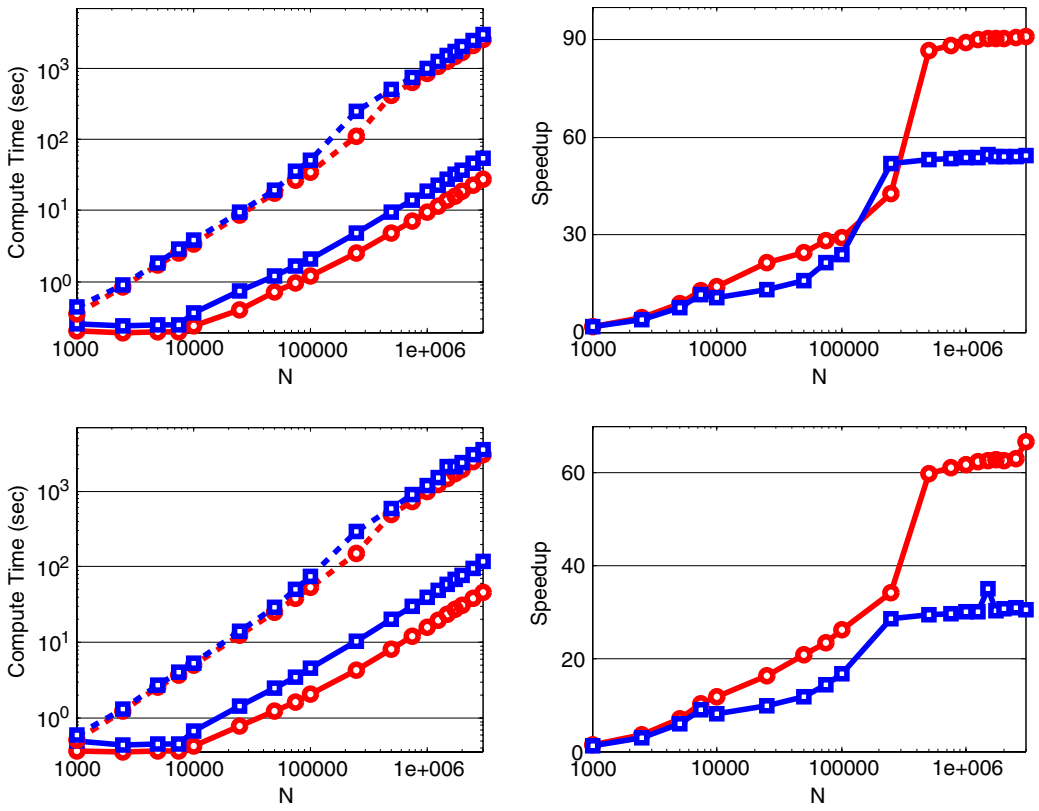

Fig. 11. (Color online) Computation times (left) and speedups (right) of the one-dimensional NLSEmagic CUDA MEX integrators (solid) versus the serial MEX integrators (dashed) for simulating the dark soliton solution described in Section 4 with an end-time of $t = 50$. The results are shown for both the RK4 + CD scheme (top) and the RK4 + 2SHOC scheme (bottom) using both single (red circles) and double (blue squares) precision.

Table 1
Subset of the one-dimensional timing results from Fig. 11. The results are shown for the RK4 + CD scheme in single (s) precision (best results) and for the RK4 + 2SHOC in double (d) precision (weakest results).

N	GPU Time (s)		CPU Time (s)		SPEEDUP	
	CD (s)	2SHOC (d)	CD (s)	2SHOC (d)	CD (s)	2SHOC (d)
1 000	0.21	0.49	0.35	0.61	1.69	1.23
10 000	0.24	0.67	3.39	5.38	14.07	7.99
100 000	1.21	4.52	35.15	75.31	29.12	16.65
1 000 000	9.41	39.72	838.21	1192.27	89.08	30.01
3 000 000	27.65	117.32	2515.56	3581.40	90.98	30.53

Despite the reduction in speedup of the two-dimensional codes, the speedup observed is still quite high, especially considering the cost of the GPU card. In fact, in the RK4 + CD test of $\mathbf{N} = 1732^2 = 2999824$, the GPU code took about 40 s, while the serial MEX code took over 24 min. Based on the MEX speedups of Section 5.3, the equivalent MATLAB script code would be expected to take almost *5 h* to complete the same simulation. In this case, assuming a perfect OpenMP implementation of a quad-core CPU, the GPU integrators achieve a maximum speedup of around 8. Although lower than the one-dimensional case, this is still a significant improvement.

It is important to note that resolutions for large **N** values (where the best speedups are observed) are more common in two dimensions. Thus, the higher speedup results are expected to be more common in actual applications than those in one dimension.

7.4. Three-dimensional speedup results

For the three-dimensional timings, we use the unoptimized approximate dark vortex ring solution described in Section 4. As was the case in the two-dimensional tests, we cannot record the error of the simulations. Therefore, to validate the simulations, we once again displayed near-center cell data points for each frame and compared them with those from the output of the non-CUDA MEX codes.

Like the one- and two-dimensional tests, the solution is plotted five times during the simulation yielding a chunk-size of 5000, which is once again well over the efficiency requirements discussed in Section 7.1. The vortex ring is simulated with a total grid size which varies from $\mathbf{N} = N * M * L \approx 10{,}000$ to $\mathbf{N} \approx 3{,}000{,}000$. The N, M, and L dimensions are determined by taking the floor of the third-root of **N** and then slightly adjusted as needed to comply with the block requirements of using the MSD boundary condition mentioned in Section 6.1. Although for the one- and two-dimensional tests we started with simulations of size $\mathbf{N} \approx 1000$, in the three-dimensional case this was not possible due to the size of the vortex ring and the chosen spatial step-size (the vortex ring overlaps the grid boundaries when $\mathbf{N} \approx 1000$).

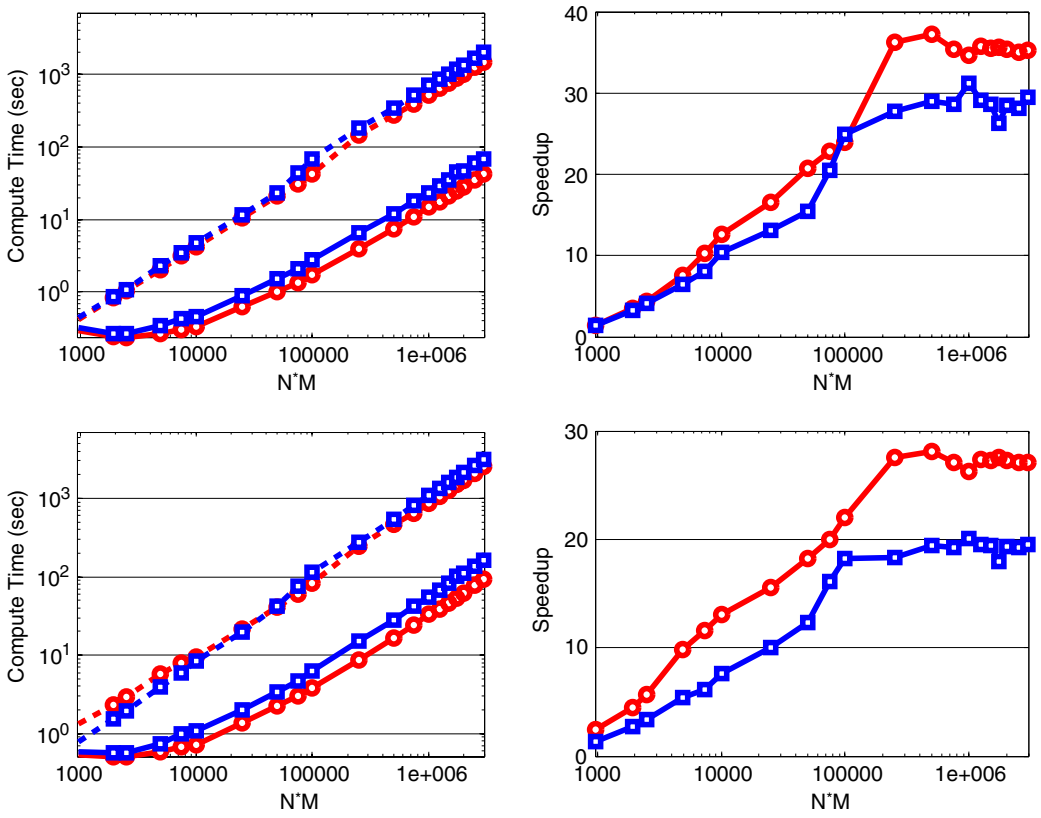

Fig. 12. (Color online) Computation times (left) and speedups (right) of the two-dimensional NLSEmagic CUDA MEX integrators (solid) versus the serial MEX integrators (dashed) for simulating the approximate dark vortex solution described in Section 4 with an end-time of $t = 50$. The results are shown for both the RK4 + CD scheme (top) and the RK4 + 2SHOC scheme (bottom) using both single (red circles) and double (blue squares) precision.

Table 2
Subset of the two-dimensional timing results from Fig. 12. The results are shown for the RK4 + CD scheme in single (s) precision (best results) and for the RK4 + 2SHOC in double (d) precision (weakest results).

$N * M$	GPU Time (s)		CPU Time (s)		SPEEDUP	
	CD (s)	2SHOC (d)	CD (s)	2SHOC (d)	CD (s)	2SHOC (d)
961	0.29	0.59	0.42	0.78	1.43	1.32
10 000	0.34	1.09	4.25	8.35	12.61	7.65
99 856	1.76	6.25	42.20	113.96	23.99	18.23
1 000 000	14.94	54.65	516.65	1094.72	34.59	20.03
2 999 824	42.15	162.79	1485.88	3173.51	35.26	19.49

The simulation compute-times and speedups compared to the serial MEX integrators are shown in Fig. 13 and Table 3. We see that, once again, the higher dimensionality has reduced the speedup performance of the GPU codes. A possible explanation is that the three-dimensional codes have even more boundary cells in the CUDA block ($O(N^2)$) than the two-dimensional case, and those cells require even more global memory accesses (as shown in Section 6.4, the 2SHOC scheme required up to 7 accesses for the corner cells). The additional number of grid boundary cells can also be a factor when using the MSD boundary condition as was explained in Section 7.3. However, in should be noted that the decrease in speedup is less (in percent) than the decrease from the one- to the two-dimensional codes.

Despite the reduction in performance compared to the two-dimensional results, the three-dimensional codes still exhibit very good speedups, especially considering the cost and portability of the GPU card. For example, in the RK4 + CD test of $\mathbf{N} = 149^3 = 2985984$, the GPU code took about 1 min 20 s, while the serial MEX code took over 34 min. Based on the MEX speedups of Section 5.3, the equivalent MATLAB script code would take over *5 h* to complete the same simulation. In this case, the GPU integrators achieve a maximum speedup of around 6 compared to a theoretical perfect quad-core OpenMP implementation. Although smaller than in the lower-dimensional cases, this speedup can be quite significant in large three-dimensional simulations. Also, in such problems, even moderately resolved solutions require very large total grid sizes. Therefore, the larger speedups will be the most common in practical applications.

All the speedup tests presented in this chapter were on grids which were nearly equal-sized in each dimension (i.e. $N \times N$ or $N \times N \times N$). Depending on the sizes, the block structure could be more or less optimized in terms of filling the edge blocks with more or less capacity. This explains why the speedups in two dimensions and (all the more so in three dimensions) are not smooth over changes in $\mathbf{N}$, but rather fluctuate. Therefore, certain specific grid sizes have the potential to be more efficient. As an example, we have simulated a vortex ring solution in a grid with dimensions $87 \times 87 \times 203 \approx 1{,}500{,}000$ for an end-time of $t = 100$ and time-step size of $k = 0.03$ (yielding a chunk-size of only 56 which is on the low end of the efficiency requirements from Section 7.1) with the RK4 + 2SHOC scheme and the MSD

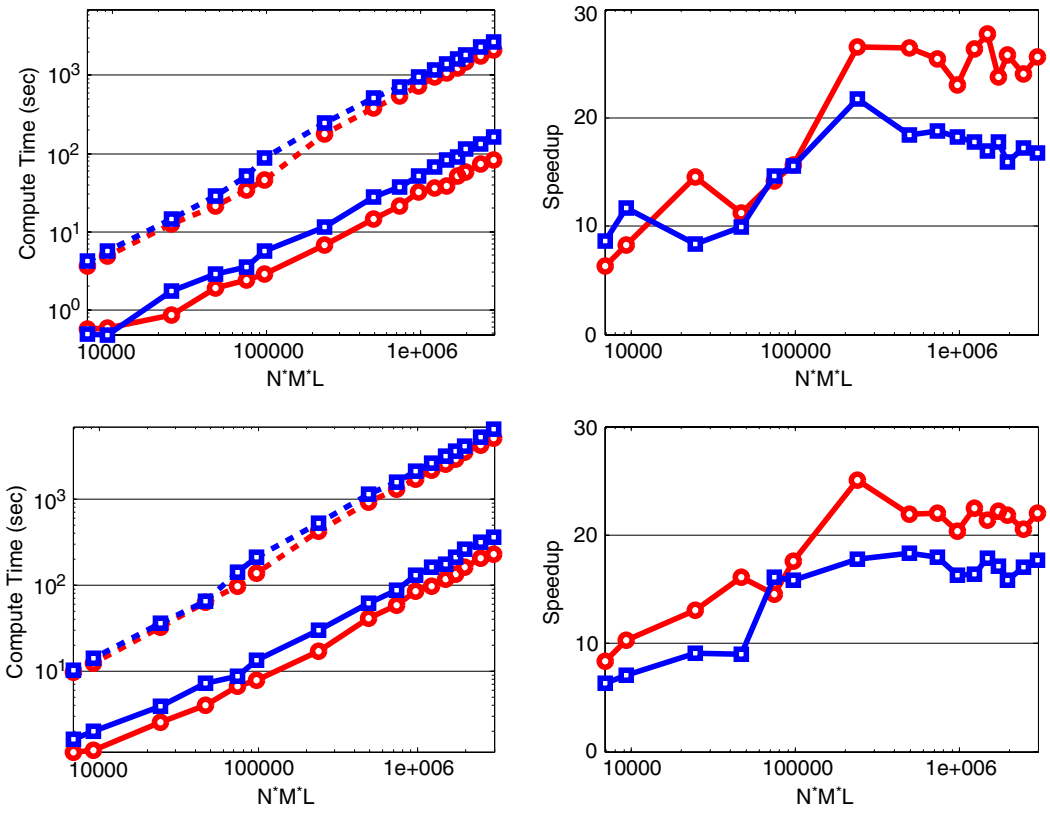

Fig. 13. (Color online) Computation times (left) and speedups (right) of the three-dimensional NLSEmagic CUDA MEX integrators (solid) versus the serial MEX integrators (dashed) for simulating the approximate dark vortex ring solution described in Section 4 with an end-time of $t = 50$. The results are shown for both the RK4 + CD scheme (top) and the RK4 + 2SHOC scheme (bottom) using both single (red circles) and double (blue squares) precision.

Table 3
Subset of the three-dimensional timing results from Fig. 13. The results are shown for the RK4 + CD scheme in single (s) precision (best results) and for the RK4 + 2SHOC in double (d) precision (weakest results).

$N * M * L$	GPU Time (s)		CPU Time (s)		SPEEDUP	
	CD (s)	2SHOC (d)	CD (s)	2SHOC (d)	CD (s)	2SHOC (d)
9 261	0.59	2.00	4.85	14.17	8.29	7.10
97 336	2.90	13.34	45.43	211.23	15.65	15.84
970 299	31.85	131.74	733.46	2146.78	23.03	16.30
2 985 984	82.14	365.47	2106.77	6460.61	25.65	17.68

boundary condition in double precision, and saw a speedup of over 26. This is much higher than the comparable speedup shown in Fig. 13 for the same total grid size and scheme options. Thus, the actual speedup in any given problem may be higher (or admittedly, lower) than those shown in Fig. 13.

We wish to reiterate that the speedup tests performed in this section are to show how an inexpensive GPU upgrade of a researcher's workstation can result in a great speedup over a serial implementation of the NLSE codes. In addition, we have noted that the GPU integrators should still yield a speedup of over 5 for large three-dimensional problems compared to a perfect OpenMP implementation of the integrators on a quad-core CPU. The speedup results of the NLSEmagic integrators will undoubtedly change with the release of new CPU and GPU architectures, but currently, in terms of both cost and performance, the GPU implementations seem to be the most efficient.

8. The NLSEmagic code package

The codes described in this paper are packaged together and collectively named NLSEmagic: Nonlinear Schrödinger Equation Multi-dimensional MATLAB-based GPU-accelerated Integrators using Compact high-order schemes. The basic NLSEmagic code package contains all the serial and CUDA MEX integrators described in this paper as well as MATLAB script driver codes which are used as examples of how to use the integrators. These scripts are called `NLSEmagic1D.m`, `NLSEmagic2D.m`, and `NLSEmagic3D.m`, and include the files `makeNLSEmagic1D.m`, `makeNLSEmagic2D.m`, and `makeNLSEmagic3D.m` which contain the commands to compile the integrators. In order to make setting up the codes as easy as possible, we also make available pre-compiled binaries of the serial and CUDA MEX integrators. All configuration files are included as well as an installation guide and a setup guide for compiling and running CUDA codes through the MATLAB MEX interface.

An important reason for distributing codes is to allow others to reproduce research results. The issue of reproducible research when using numerical simulations is a topic of great concern in the scientific community [35–37]. Because many codes in use are written to run

on specific hardware, and are typically not available to others, it is very difficult to validate numerical results without taking a large amount of time to reproduce an equivalent code to test the simulations. This is highly unfeasible, especially when the codes require parallelization and large computer clusters to run.

As an example of reproducibility, we have included what we call 'full research scripts' to the NLSEmagic package. These driver scripts contain all the code necessary to reproduce the results computed in this paper (as well as those in the author's dissertation [31]). This includes the initial conditions (vortices, vortex rings, etc.), as well as code to visualize, track, analyze, and save images and movies of the simulations. They therefore also contain the script integrators used in the MEX speedup calculations of Section 5.3, which can be used to develop new numerical methods. The full research driver scripts are called `NLSE1D.m`, `NLSE2D.m`, and `NLSE3D.m`, and rely on other script codes which are also included in the download package.

The full-research and basic driver scripts of NLSEmagic make use of some freely available third-party MATLAB codes. One such code is called `vol3D` [38] which is used to produce the volumetric renderings of the three-dimensional NLSE simulations. Another code included is `nsoli` [39] which is a Newton–Krylov nonlinear equations solver used to find numerically exact vortices and vortex rings. Finally, a code called `ezyfit` [40] is used to perform simple least-squares curve fitting to formulate models of the numerical results.

The NLSEmagic code package is distributed on a dedicated website free of charge.[1] A Facebook users group is also maintained for posting updates and discussions.[2] As of this writing, the website has received over 950 unique hits from 50 countries which indicates its interest to many researchers. The code packages and website contain all the documentation necessary to setup, compile, install, and run the various codes.

9. Conclusion

In this paper we have described the implementation of a new code package for simulating the multi-dimensional nonlinear Schrödinger equation utilizing GPU acceleration called NLSEmagic. The codes use an explicit finite-difference scheme using fourth-order Runge–Kutta for the time-stepping and both a second-order central differencing and a two-step compact fourth-order differencing for the Laplacian. The codes are interfaced with MATLAB through MEX-compiled codes written in C and CUDA for ease of use and efficiency. We have shown that the CUDA codes exhibit very good speedup results even when using an inexpensive off-the-shelf GPU card. The NLSEmagic code package is distributed as a free download at www.nlsemagic.com.

All trademarks and trade names including AMD, ATI, EVGA, Mathworks, MATLAB, MEX, Intel, NVIDIA, CUDA, GeForce, Quadro, Tesla, Fermi, Kepler, Microsoft, Windows, Visual C++, Facebook, and PGI are the property of their respective holders.

Acknowledgments

This research was supported by NSF-DMS-0806762 and the Computational Science Research Center (CSRC) at San Diego State University. We gratefully acknowledge Professor Ricardo Carretero for his facilitation of this project and would like to thank Mohammad Abouali for his advise on optimizing the CUDA codes.

Appendix. Specifications of the CPU and GPU used in this paper

Here we show the specifications of the GPU and CPU that were used for the results in this paper. The CPU has the following relevant specifications:

```
Operating System:           MS Windows 7 Enterprise 64-bit
CPU Name:                   Intel Core i3 540
CPU Clock Speed:            3.07 Ghz
Cores:                      2
Threads:                    4
L1 Data cache:              2 x 32 KB
L1 Instruction cache:       2 x 32 KB
L2 cache:                   2 x 256 KB
L3 cache:                   4 MB
DDR3 RAM:                   4.0 GB
Memory Clock rate:          669 Mhz

Performance Information (Computed with QwikMark)
------------------------------------------------
CPU Core Performance:       61 Gflop/s
Memory Bandwidth:           6 GB/s
```

The GPU has the following relevant specifications:

```
GPU Name:                                   GeForce GTX 580 (EVGA)
CUDA Driver Version / Runtime Version:      4.1 / 4.1
CUDA Capability Major/Minor version number: 2.0
```

[1] http://www.nlsemagic.com.

[2] http://www.facebook.com/nlsemagic.

```
Total amount of global memory (GDDR5 RAM):     1536 MB
(16) Multiprocessors x (32) CUDA Cores/MP:     512 CUDA Cores
GPU Clock Speed:                               1.59 Ghz
Memory Clock rate:                             2025.00 Mhz
Memory Bandwidth (GB/sec)                      192.4
L2 Cache Size:                                 786432 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 32768
Maximum number of threads per block:           1024
Concurrent copy and execution:                 Yes with 1 copy engine
Run time limit on kernels:                     Yes
Concurrent kernel execution:                   Yes
Device has ECC support enabled:                No

Performance Information (Computed with CUDA-Z)
----------------------------------------------
Memory Copy
  Host Pinned to Device:          596.208 MB/s
  Host Pageable to Device:        532.289 MB/s
  Device to Host Pinned:          594.033 MB/s
  Device to Host Pageable:        531.315 MB/s
GPU Core Performance
  Single-precision Float:      1622440 Mflop/s
  Double-precision Float:       204026 Mflop/s
  32-bit Integer:               814731 Mflop/s
  24-bit Integer:               813770 Mflop/s
```

References

[1] P. Kevrekidis, D. Frantzeskakis, R. Carretero-González, Emergent Nonlinear Phenomena in Bose–Einstein Condensates: Theory and Experiment, in: Springer Series on Atomic, Optical, and Plasma Physics, vol. 45, 2008.
[2] R.M. Caplan, Q.E. Hoq, R. Carretero-González, P.G. Kevrekidis, Opt. Commun. 282 (2009) 1399.
[3] L. Debnath, Nonlinear Partial Differential Equations for Scientists and Engineers, Birkhauser, second ed., 2005.
[4] S. Krakiwsky, L. Turner, M. Okoniewski, Graphics processor unit GPU acceleration of finite-difference time-domain (FDTD) algorithm, in: Proceedings of the 2004 International Symposium on Circuits and Systems, vol. 5, 2004, pp. 265–268.
[5] S. Adams, J. Payne, R. Boppana, HPCMP User Group Conf., 2007, p. 334.
[6] A. Balevic, et al. Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose GPUs, in: Proceedings of Computational Science and Engineering'08, 2008, pp. 327–334.
[7] P. Micikevicius, 3D finite difference computation on GPUs using CUDA, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009, pp 79–84.
[8] D. Michéa, D. Komatitsch, Geophys. J. Int. 182 (2010) 389.
[9] K.A. Hawick, D.P. Playne, Massey University Tech. Report (2010) CSTN.
[10] P. Muruganandam, S.K. Adhikari, Comput. Phys. Commun. 180 (2009) 1888.
[11] D. Vudragovic, I. Vidanovic, A. Balaz, P. Muruganandam, S.K. Adhikari, Comput. Phys. Commun. 183 (2012) 2021.
[12] Mathworks, http://www.mathworks.com/discovery/matlab-gpu.html, 2011.
[13] NVIDIA, http://developer.download.nvidia.com, 2007.
[14] R.M. Caplan, R. Carretero-González, Appl. Math Comput. (2012) arXiv:1109.1027 (submitted for publication).
[15] NVIDIA, http://developer.nvidia.com/nvidia-gpu-computing-documentation, 2011.
[16] J.E. Stone, D. Gohara, G. Shi, Comput. Sci. Engi. 12 (2010) 66.
[17] K. Komatsu, et al. The Fifth International Workshop on Automatic Performance Tuning, iWAPT2010, 2010.
[18] J. Fang, A.L. Varbanescu, H. Sips, A comprehensive performance comparison of CUDA and OpenCL, in: Proceedings of Parallel Processing'11, 2011, pp. 216–225.
[19] K. Karimi, N.G. Dickson, F. Hamze, ArXiv e-prints, arXiv:1005.2581, 2010.
[20] G. Martinez, W. Feng, M. Gardner, CU2CL: A CUDA-to-OpenCL translator for multi- and many-core architectures, Technical report, Virginia Tech, 2011.
[21] T.P. Group, PGInsider 2 (2010) a1.
[22] J. Stratton, S. Stone, W. Hwu, MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, in: Lecture Notes in Computer Science, vol. 5335, Springer, Berlin, Heidelberg, 2008.
[23] J.S. Vetter, et al., Comput. Sci. Engi. 13 (2011) 90.
[24] J. Castillo, J. Hymanb, M. Shashkov, S. Steinberg, Appl. Numer. Math. 37 (2001) 171187.
[25] J. Butcher, Appl. Numer. Math. 20 (1996) 247.
[26] W. Dai, SIAM J. Numer. Anal. 29 (1992) 174.
[27] R.M. Caplan, R. Carretero-González, App. Num. Math. (2012) arXiv:1107.4810 (submitted for publication).
[28] R.M. Caplan, R. Carretero-González, Appl. Math Comput. (2012) arXiv:1110.0569 (submitted for publication).
[29] Y.S. Kivshar, B. Luther-Davies, Phys. Rep. 298 (1998) 81.
[30] R. Carretero-González, D. F -z -k -k, P. Kevrekidis, Nonlinearity 21 (2008) R139.
[31] R.M. Caplan, Study of Vortex Ring Dynamics in the Nonlinear Schrödinger Equation utilizing GPU-Accelerated High-Order Compact Numerical Integrators, Ph.D. Thesis, Claremont Graduate University and San Diego State University, 2012.
[32] P.H. Roberts, J. Grant, J. Phys. A: Gen. Phys. 4 (1971) 55.
[33] T. Davis, MATLAB Primer, eighth ed., CRC Press, 2010.
[34] B. Jang, D. Schaa, P. Mistry, D. Kaeli, IEEE Trans. Parallel Distrib. Syst. 22 (2011) 105.
[35] D.E. Post, L.G. Votta, Phys. Today 58 (2005) 35.
[36] D.L. Donoho, A. Maleki, I.U. Rahman, M. Shahram, V. Stodden, Comput. Sci. Eng. 11 (2009) 8.
[37] K. Diethelm, Comput. Sci. Eng. 14 (2012) 64.
[38] O. Woodford, J. Conti, http://www.mathworks.com/matlabcentral/fileexchange/22940-vol3d-v2, 2011.
[39] C.T. Kelley, Solving Nonlinear Equations with Newton's Method, in: Fundamentals of Algorithms, SIAM, 2003.
[40] F. Moisy, http://www.fast.u-psud.fr/ezyfit/, 2010.