ACSESS proceedings
2010

Simulating the Nonlinear Schrödinger Equation using the Computational Capability of NVIDIA Graphics Cards

Ronald M. Caplan and Ricardo Carretero

AP10-04

CSRC an interdisciplinary gateway to the future
SAN DIEGO STATE UNIVERSITY

Simulating the Nonlinear Schrödinger Equation using the Computational Capability of NVIDIA Graphics Cards

Ronald M. Caplan[1] *, *Ricardo Carretero*[1]
[1] *Computational Science Research Center San Diego State University*
* *sumseq@gmail.com*

Abstract: Much research in systems governed by the nonlinear Schrödinger equation (NLSE) continues to this day. Most work is performed using numerical simulations, but in order to make the research more efficient (and in some cases, even, plausible), it is desirable to have new numerical methods and technology which speed up the computation time of the simulations. A recent development in parallel computation is the use of video graphics cards for scientific numerical problems. We describe our efforts to use NVIDIA graphics cards to speed up computations of the one-dimensional NLSE. This is done by modifying our original code using a C-language extension API called CUDA. We also describe our numerical methods which include a two-step high-order compact finite difference scheme as well as a simple to implement constant modulus-squared background boundary condition. We find that by using a single inexpensive graphics card, our simulations can run up to 70 times faster. We hope to extend the results to two and three dimensions, where such speedup will have the most benefit.

Keywords: CUDA, GPU, nonlinear Schrödinger equation

1 Introduction

There has and continues to be much research in systems governed by the Nonlinear Schrödinger Equation (NLSE) [2]. Examples of such systems include Bose-Einstein condensates [3] and nonlinear optics [1]. Since the dynamics of the NLSE are nonlinear, most research is performed by simulating the NLSE using numerical methods. This is especially true in two and three dimensions, where such simulations can take a very long time to run. In order to make the research more efficient (and in some cases, even, plausible), it is desirable to have new numerical methods which speed up the computation time of the simulations.

Graphical processing units (GPUs) are a new tool in massively parallel computing. The GPUs are structured to have many moderate speed ALUs arranged in a very efficient hardware orientation. For some problems, the GPUs have the potential to speed up computations by a factor of over one hundred.

In this paper, we describe our implementation and speedup results for simulating the one dimensional NLSE using NVIDIA GPUs with their custom API called CUDA. We show that we can achieve very good results, and therefore plan to further develop the code into multidimensional domains.

2 Nonlinear Schrödinger Equation and Trial Solution

The one-dimensional general NLSE takes the form:

$$i\Psi_t + a\Psi_{xx} + (V(x) + s\,|\Psi|^2)\Psi = 0, \quad (1)$$

where Ψ is the value of the wave function, $V(x)$ is the external potential function, and a and s are constant parameters determined by the physical system being described. The sign of s determines if the nonlinearity is 'focusing/attractive' or 'defocusing/repulsive'.

In order to test our codes we need an example problem to which we know the exact solution. In this regard we use the following one-dimensional dark soliton [3]:

$$\begin{aligned}\Psi(x,t) &= \sqrt{\left|\tfrac{\Omega}{s}\right|}\tanh\left[\sqrt{\tfrac{|\Omega|}{2a}}\,(x - c\,t)\right]\,E(2)\\ E &= \exp\left(i\left[\tfrac{c}{2a}\,x + \left(\Omega - \tfrac{c^2}{4a}\right)t\right]\right)\end{aligned}$$

where c is the velocity of the soliton, x_0 is its initial position, and Ω is the frequency. This particular solution exists and is exact in the NLSE when $V(x) = 0$ and $s < 0$. The soliton describes a localized curve in the modulus squared of Ψ which propagates without dispersion or dissipation amidst a constant background in the modulus squared. A depiction of such a soliton within our simulations is given in Fig. 1.

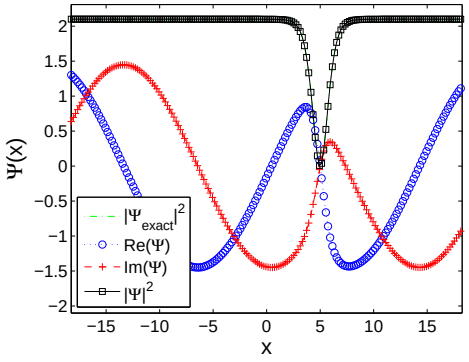

Figure 1: Depiction of a dark soliton within a numerical simulation of the NLSE. Here, $a = 1.1$, $s = -1.1$, and $c = 0.5$.

3 NVIDIA GPU

Over the past few years, the computational power of video cards Graphic Processing Units (GPU) has grown greatly, outgrowing the computational power of CPUs. Various companies produce GPU hardware, the two largest being ATI and NVIDIA. The GPUs are constructed to be massively parallel, with many mid-speed ALUs. For our project, we use NVIDIA's line of graphics card GPUs. Each card contains a number of multi-processor (MP) units, each containing 8 ALUs with a common 16KB shared memory. The cards also contain a large global memory anywhere from 512MB to 4GB as shown in Fig. 2.

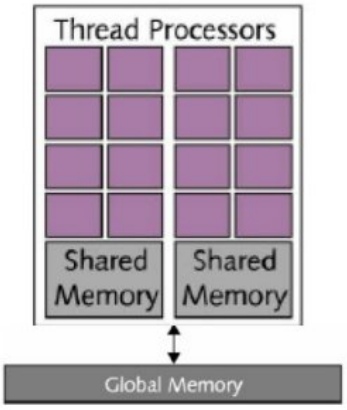

Figure 2: Schematic of NVIDIA GPU architecture. Each multiprocessor contains 8 APUs with a shared memory, and all multiprocessors have access to a large global memory [4].

This allows for the GPUs to be very powerful in many parallelizable scientific computational problems.

4 CUDA API

NVIDIA allows programmers to utilize its GPUs through an API called CUDA. CUDA is a code extension to C/C++ which gives low-level access to the GPUs memory and processing abilities. Currently, a new API called OpenCL is being developed which can function across multiple GPU vendors, as well as other CPU architectures; however, for best performance, the OpenCL code needs to be optimized for the specific architecture being used. Since NVIDIA already has a well developed optimized API tailored to their GPUs, we decided to use CUDA over OpenCL (if in the future it is desired to run the code on other GPUs, converting CUDA code into OpenCL is said to be very straight forward).

In a typical CUDA code, data is transferred from the CPU to the GPU, and then the GPU performs computations on the data, which, when completed, is transferred back to the CPU. The execution of the computations is done through a logic structure hierarchy of grids, block, and threads.

Each compute grid contains a number of blocks. Each block has a number of compute threads. The threads are what perform the computations. Each block of threads is executed on a multiprocessor, where the threads

are pipelined on the ALUs in a highly optimized manner. Each block has a shared (fast) memory which is accessible to the threads in that block. All threads in the grid have access to the (slow) global memory of the GPU as shown in Fig. 3 [4].

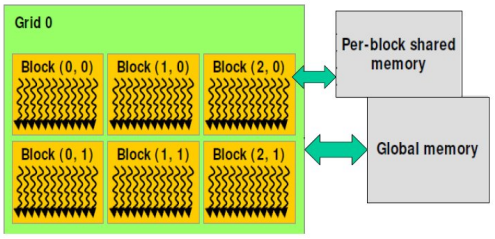

Figure 3: Schematic of NVIDIA CUDA API structure. Each compute grid contains a number of blocks. Each block has a number of threads which can all access the same shared memory, but not the shared memory of any other block. All threads in the grid can access the global memory. [4]

As in all parallel environments, synchronization, memory usage, and dependencies have to be dealt with, but the CUDA API makes these issues easy to handle.

5 Numerical Method for NLSE Simulations

Here we describe our numerical methods for simulating the NLSE and their implementation using the CUDA API.

5.1 High Order Two-Step Compact Scheme

For our numerical method, we use a fourth-order Runga-Kutta (RK4) scheme in time. We write the NLSE as

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

so the RK4 scheme is defined as

$$\Psi^{n+1} = \Psi^n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad (3)$$

where

$$\begin{aligned} k_1 &= F(\Psi^n), \\ k_2 &= F(\Psi^n + \tfrac{1}{2}\Delta t k_1), \\ k_3 &= F(\Psi^n + \tfrac{1}{2}\Delta t k_2), \\ k_4 &= F(\Psi^n + \Delta t k_3), \end{aligned}$$

and $\Psi^n = \Psi_j^n \, \forall j$ where n is the current time step. In order to allow for the use of courser grids, we also use a fourth-order spatial scheme for the Laplacian in $F(\Psi)$. Usually, such schemes had wide stencils, which are not suitable for parallel applications due to extra communication and/or memory latency. Also, the grid points near the boundary are difficult to deal with. To avoid this, we use a 2-step high-order compact scheme (2SHOC) defined as

$$D_j = \frac{\Psi_{j+1} - 2\Psi_j + \Psi_{j-1}}{h^2}, \quad (4)$$

$$\left.\frac{\partial^2 \Psi}{\partial x^2}\right|_j \approx \frac{7}{6}D_j - \frac{D_{j+1} + D_{j-1}}{12}, \quad (5)$$

where h is the spatial grid spacing. The scheme is derived from using a central difference of the precomputed central difference to give a second order approximation of the fourth spatial derivative, which when inserted into the first error term of the standard central difference results in a fourth-order approximation of the second derivative.

In our implementation of the 2SHOC scheme, each step in computed separately on the GPU, and so the computations are compact in each step. The drawbacks of the scheme are the extra memory requirement for storing D, and an increase in the number of floating point operations per grid point when compared to the standard fourth-order 5-point stencil. However, for our GPU implementation, the advantages of using the 2SHOC scheme outweigh the disadvantages.

5.2 Modulus-Squared Dirichlet Boundary Condition

For our example problem, we would like to have a simple boundary condition. For many problems, if the domain is large enough, we can assume a constant value at the boundaries and use a Dirichlet condition. However in our problem this does not work because we assume a constant background value and the boundary in $|\Psi|^2$, not in the value of the Ψ. We therefore formulate a modulus-squared-Dirichlet (MSD) boundary condition. We start with our desired boundary condition

$$|\Psi_0|^2 = B,$$

where B is a constant. This implies

$$\frac{\partial}{\partial t}|\Psi_0|^2 = 0.$$

Separating into real and imaginary parts and using the chain rule yields

$$\Psi_R \frac{\partial \Psi_R}{\partial t} = -\Psi_I \frac{\partial \Psi_I}{\partial t}$$

A solution that satisfies this equation is

$$\frac{\partial \Psi_R}{\partial t} = C\Psi_I \qquad \text{and} \qquad \frac{\partial \Psi_I}{\partial t} = -C\Psi_R,$$

Making some substitutions, taking one-sided spatial differencing on both sides of the resulting equation, and rearranging, the constant C can be determined as a function of Ψ, in which case we formulate the MSD boundary condition as

$$\frac{\partial \Psi_0}{\partial t} \approx \left(\frac{\Psi_0}{\Psi_1}\right) \frac{\partial \Psi_1}{\partial t}, \tag{6}$$

where the time derivative of the interior point is precomputed using the interior numerical scheme. This boundary condition can be used for any time-dependent complex PDE, and is easy to implement. However, for the first step in our 2SHOC scheme, we need accurate boundary conditions for the Laplacian in the computation of D. To do this, we use the NLSE to rearrange the MSD to give use a boundary condition for the Laplacian:

$$\nabla^2 \Psi_0 \approx \left[\frac{\nabla^2 \Psi_1}{\Psi_1} + L\right] \Psi_0, \tag{7}$$
$$L = \frac{V_1 - V_0}{a} + \frac{s}{a}(|\Psi_1|^2 - |\Psi_0|^2).$$

Using both forms of the MSD boundary condition in the 2SHOC scheme keeps the spatial accuracy fourth-order.

5.3 CUDA Implementation

The full details of the CUDA implementation of our numerical method is beyond the scope of this paper. We instead give a brief outline of the code.

Each CUDA block must be able to run independently of all others, and therefore each step in our numerical method must be executed in a separate CUDA kernel. The kernel describes what each thread should compute. In our code, each thread computes on one grid cell. The input vector for each kernel is stored in global memory. Since access to this memory is very slow, we utilize the per-block shared memory. Each thread copies the needed value from global memory into shared memory, and after synchronizing the threads in the block, each thread has access to a block sized chunk of the input vector. Thus, each thread's access to the input vector is now in fast shared memory. A problem exists at the block boundaries, since the boundary thread does not have access to the input vector in shared memory in the adjacent block. Therefore, for the boundaries of the block, the thread must access one cell of global memory for the compact computations. For steps that simply add input vectors together, shared memory is not used because they only need to access each cell once, and so storing the value in shared memory would not decrease the number of global memory accesses. The code runs a defined number of time steps and then transfers the current vector of Ψ to the host CPU for plotting and analysis. The larger the number of time steps the kernels compute, the better the speedup since memory transfers from the GPU to CPU and vice-versa are very slow.

6 Speedup Results

For our results, we use an NVIDIA GeForce GTX 260 GPU which retails for around $200. This model of GPU has 896MB of global RAM and 24 MPs which equate to 192 ALUs. The PC on which we run the code has a Quad-core Intel Xeon 2GHz processor with 2GB of RAM, is using the GPU to run the display, and is running 64-bit Linux. The speedup results are shown in Fig. where N is the number of grid points. We also show a few of the highest speedup examples in Table 1.

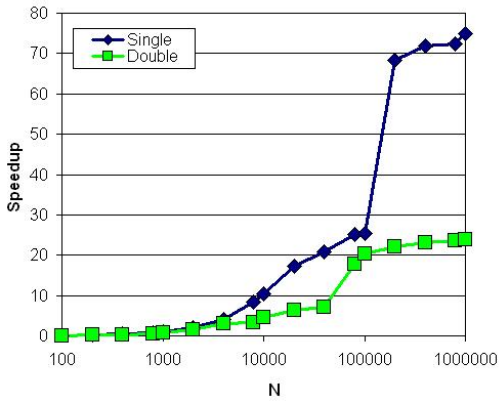

Figure 4: Speedup results for simulating the NLSE 1D dark soliton in both single and double precision for 10,000 time steps using our 2SHOC scheme with the fourth order Runga-Kutta scheme in time. N is the grid size. The simulations have a grid spacing of $h = 0.2$, a time step of $k = 0.001$, and end time of $t = 10$. The soliton has the same parameters as in Fig. 1.

N	**CPU**	**GPU**	**Speedup**
10000	17.18	1.64	**10.45**
100000	171.26	6.72	**25.48**
1000000	4062.08	54.23	**74.90**

Table 1: Sample speedup results for NLSE simulations. The CPU and GPU timings are in seconds. All simulation parameters are as in Fig. 4. The timings shown here are for single precision.

We see very good speedup, especially when we consider that this is using 1 PC with a simple $200 add-on card. A simulation that took over an hour on the CPU was computed by the GPU in less than a minute. To get equivalent speedup using a multi-PC parallel grid would require much more resources and money. We do notice that using double precision results in much less speedup. However, double precision support on the GPUs is very new, and the newest NVIDIA GPU architecture called Fermi is said to handle double precision more efficiently.

Currently, one can purchase custom made PCs which contain multiple dedicated GPUs with the combined compute power of over a Tera-flop. These desktop super computers have the potential to increase speedup by an order of magnitude at a fraction of the cost of an equivalent PC CPU cluster.

7 Conclusion and Outlook

From our speedup results, we conclude that using CUDA enabled GPUs to reduce the computation time of our NLSE simulations is well worth the added code development time. In single precision, we have obtained speedups of over 70 when using a single modestly priced GPU.

We predict that the GPU codes will be most effective in two and three dimensional settings, since such multidimensional settings are where the grid size will naturally be very large.

Two dimensional codes are currently in development, with the final goal to produce a full three-dimensional code for use in research projects.

We acknowledge support from NSF grant NSF-DMS- 0505663 and NSF-DMS-0806762.

References

[1] R. M. Caplan, R. Carretero-González, P.G. Kevrekidis, and Q. E. Hoq, *Azimuthal modulational instability of vortices in the nonlinear Schrodinger equation*, Optics Communications **282** (2009), 1399–1405.

[2] L. Debnath, *Nonlinear partial differential equations for scientists and engineers*, 2 ed., Birkhauser Boston, New York, New York, 2005.

[3] P. G. Kevrekidis, D. J. Frantzeskakis, and R. Carretero-González, *Emergent nonlinear phenomena in Bose-Einstein condensates: Theory and experiment*, 1 ed., Springer, New York, New York, 2007.

[4] NVIDIA, *Cuda 2.3 documentation*, 2009.