Simulating the Nonlinear Schrödinger Equation using CUDA with MATLAB Project Report for COMP 670

Ron Caplan

October 27, 2009

Abstract

In this project report, we describe our attempts and results for simulating the Nonlinear Schrödinger Equation using CUDA with MATLAB. Our overall goal is to speed up the simulation's computational time, without requiring too much extra code development time or expensive resources. Our group members are Eunsil Baik, Ron Caplan, and Freddy Mcdonald. This report will focus on the contributions of Ron Caplan. For details on Eunsil and Freddy's contributions, see their respective reports. Our end result is that using CUDA with MATLAB can offer significant speedup, and with proper documentation, can be done fairly easily and without great cost.

1 Introduction

There has and continues to be much research in systems governed by the Nonlinear Schrödinger Equation (NLS) [1]. Examples of such systems include Bose-Einstein condensates [2] and nonlinear optics [3]. 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 desired to have new numerical methods which speed up the computation time of the simulations.

This report will document our team's efforts to speed up computations of the NLSE through the use of Nvidia's CUDA API for GPU graphics and dedicated cards. (What CUDA is, how it works, and how to set it up is discussed in Freddy Mcdonald's project report). In addition to speeding up computation time, our secondary goal is to do so in such a way that an average researcher can implement the CUDA modifications without too much difficulty or time. With this goal in mind, we focused on ways to incorporate CUDA into MATLAB, an easy to use software environment very commonly used by researchers.

We implement CUDA into MATLAB in two ways. First, we develop custom mex files written in C which uses CUDA. This method will be focused upon in this report. Another method was to use pre-written MATLAB libraries called GPUmat, Information about GPUmat and our implementation results can be found in Eunsil Baik's project report.

Our results will show that currently, we feel that writing custom mex files is the only way to get speedup in the computation of the NLSE using our numerical method. Although the implementation of mex files is not user-friendly, the results will show that one can achieve speedups of over 100 versus non-CUDA codes. The complete code for the one-dimensional NLSE simulations is made available online at `http://www.sumseq.com/cuda` to assist others in developing custom mex files for their own use.

2 The Nonlinear Schrödinger Equation

The one-dimensional nondimentionalized NLSE is given by:

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

where Ψ is the value of the wave function, $V(x)$ is the potential function, and a and s are constants determined by the applications. The sign of s determines if the nonlinearity is 'focusing/attractive' or 'defocusing/repulsive'. For this report, our solution to the NLS that we are simulating has $V(x) = 0$, however we leave $V(x)$ in our code to keep it more general (we do however, assume $V(x)$ is real valued).

2.1 Example Problem for 1D NLSE

In order to test our codes we need an example problem which we know the exact solution to. In this regard we use the following one-dimensional sech soliton[4]:

$$\Psi(x,t) = \sqrt{2}\,\mathrm{sech}(\sqrt{2}(x - x_0 - ct))e^{i(cx + \frac{2-c^2}{2}t)}, \tag{2}$$

where c is the velocity of the soliton and x_0 is its initial position. The soliton describes a sech-shaped curve in the modulus squared which propagates without dispersion or dissipation. A depiction of such a soliton within our simulations is given in Fig. 1.

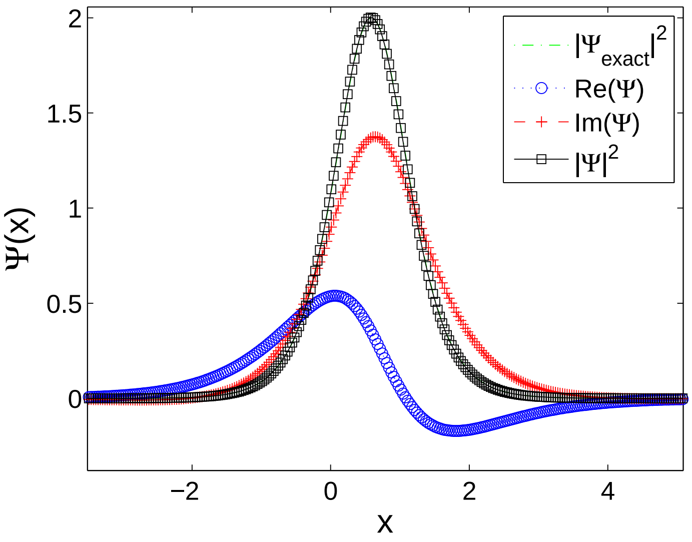

Figure 1: Example of a bright soliton solution to the NLSE in our simulations.

3 Numerical Method and Initial Implementation

There are many ways to numerically integrate the NLSE. The scheme we use is the classic fourth-order Runga-Kutta (RK4) in time and second order central differencing (CD) in space. This scheme is the simplest explicit finite difference scheme for the NLSE that is conditionally stable that we can find. Once the principles of the CUDA implementation are understood, more advanced schemes could be used (such as High-Order Compact Schemes, Mimetic operators, etc).

Our computational grid consists of N grid points. We then discretize the wave function of the NLSE as

$$\Psi(x,t) \equiv \Psi_j^n, \tag{3}$$

where n is the current time step and j is the spatial position on the grid. Therefore we have that $t = nk$ where $k = \Delta t$ and $x = x_0 + jh$ where $h = \Delta x$. We define the end time of the simulation as $t_{\text{end}} = kt_{\text{res}}$ and the spatial end points as $x_{\min} = x_0$ and $x_{\max} = x_0 + (N-1)h$. Therefore $n \in [0, t_{\text{res}}]$ and $j \in [0, N-1]$.

3.1 Fourth-Order Runga-Kutta and Central Difference

To implement the RK4 scheme, we first write the semi-discrete form of Eq. (1) as:

$$\left.\frac{\partial \Psi_j}{\partial t}\right|_{t=nk} = F(\Psi_j^n) = i(a\nabla^2\Psi_j^n + (V_j + s|\Psi_j^n|^2)\Psi_j^n), \tag{4}$$

and then the RK4 is described as [5]:

$$\Psi_j^{n+1} \approx \Psi_j^n + \frac{k}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \tag{5}$$

where

$$\begin{aligned} k_1 &= F(\Psi_j^n), & k_2 &= F(\Psi_j^n + \frac{\Delta t}{2}k_1), \\ k_3 &= F(\Psi_j^n + \frac{\Delta t}{2}k_2), & k_4 &= F(\Psi_j^n + \Delta t k_3). \end{aligned}$$

To evaluate the Laplacian in Eq. (4) we use the CD scheme defined by

$$\nabla^2\Psi_j^n \approx \frac{\Psi_{j+1}^n - 2\Psi_j^n + \Psi_{j-1}^n}{h^2}.$$

From trial and error, we have found that in order to ensure numerical stability, we must set the time step as:

$$k \leq \frac{h^2}{a\sqrt{2}}. \tag{6}$$

Thus, whenever we make the spatial step smaller, the time step must also decrease substantially.

3.2 MATLAB Implementation

As already mentioned, one of our goals is to produce a method to create very fast computations of the NLSE for researchers who may not be advanced programmers. In this light, we chose to develop our odes in MATLAB. One of the drawbacks of MATLAB is computation speed for large problems, which can be fixed by the use of custom mex files. Mex is a C/FORTRAN compiler interface in MATLAB that allows one to use compiled C or FORTRAN code seamlessly in MATLAB. NVIDIA also seeing the advantages of MATLAB have produces a CUDA enabled mex compiler called nvmex. Therefore we will develop our code using C with CUDA and compile the code into a mex file which anyone can use with MATLAB. (It should be pointed out however, that the mex files must be recompiled when using the code on a different platform than it was originally developed).

Our original implementation of the RK4+CD scheme is a non-mex MATLAB code previously written by Ron Caplan consisting of two script files. One file is the main body of the code, while the other is a function which evaluates $F(\Psi)$ of Eq. 4. Thus, for each time step, the script file calls the function script file four times. All the time steps are in a single loop and after a given number of steps, the current wave function is visualized and compared to the exact solution.

In order to have accurate timing results, we need to write a mex version of the code. This is because mex files are generally faster than script files, and since CUDA can only be implemented using mex files, we want to ensure fairness when comparing computation times.

3.3 Mex Implementation

To formulate a mex version of our script code, we simply mimic the script function, but use a mex file to evaluate $F(\Psi)$ instead of the script. It also converts the solution to single precision, evaluates the function, and then converts the result back to double precision. We do this because most affordable CUDA cards can only handle single precision variables, and so to ensure fair timings, we have both codes convert to single precision and then back to double (MATLAB's native format).

It should be noted that when using a mex file, complex variables are split into their real and imaginary parts. Thus, we have to rewrite $F(\Psi)$ into its real and imaginary parts separately. Writing $F(\Psi)$ in this way we have

$$\begin{aligned} F(\Psi)_j^{\text{real}} &= -\frac{a}{h^2}(\Psi_{j+1}^{\text{imag}} - 2\Psi_j^{\text{imag}} + \Psi_{j-1}^{\text{imag}}) - s\Psi_j^{\text{imag}}[(\Psi_j^{\text{real}})^2 + (\Psi_j^{\text{imag}})^2] - V_j\Psi_j^{\text{imag}} \qquad (7) \\ F(\Psi)_j^{\text{imag}} &= \frac{a}{h^2}(\Psi_{j+1}^{\text{real}} - 2\Psi_j^{\text{real}} + \Psi_{j-1}^{\text{real}}) - s\Psi_j^{\text{real}}[(\Psi_j^{\text{real}})^2 + (\Psi_j^{\text{imag}})^2] + V_j\Psi_j^{\text{real}} \end{aligned}$$

A small bug worth mentioning is that if the initial condition is fully real, then the C-pointer in the mex file to the imaginary part of Ψ is null-valued which causes a segmentation fault. To fix this, we check to see if there is an imaginary part of the initial condition, and if not, allocate an imaginary vector of size N manually.

Porting the script file into a mex file did speed up the code as predicted. Two results are shown in Fig. 2. The results are promising because if the CUDA code has significant speedup when compared to the mex implementation, then that speedup is even more impressive when compared to the original script implementation. Now that we have a mex version of the code, we wish to implement CUDA.

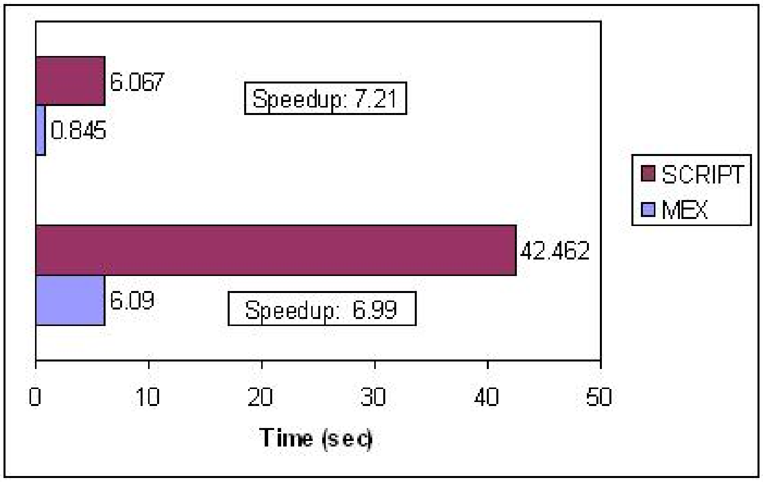

Figure 2: Examples of Speedup for using mex files versus script files in MATLAB. Both runs simulate the NLSE bright soliton with an end time of 50 and velocity of $c = 0.25$. Each timing is the time of computation of all the $F(\Psi)$ calls. Run 1 (top) has parameters: $h = 1/10$, $N = 650$, $tres = 4715$. Run 2 (bottom) has: $h = 1/20$, $N = 1299$, $tres = 18875$.

4 CUDA

CUDA is an API developed by Nvidia to allow one to run computations using C or FORTRAN on specific Nvidia GPU cards. The GPU cards are designed to be massively parallel, and hence have great potential for speedup. For an introduction to CUDA, details on how to set up the CUDA drivers, SDK, and MATLAB

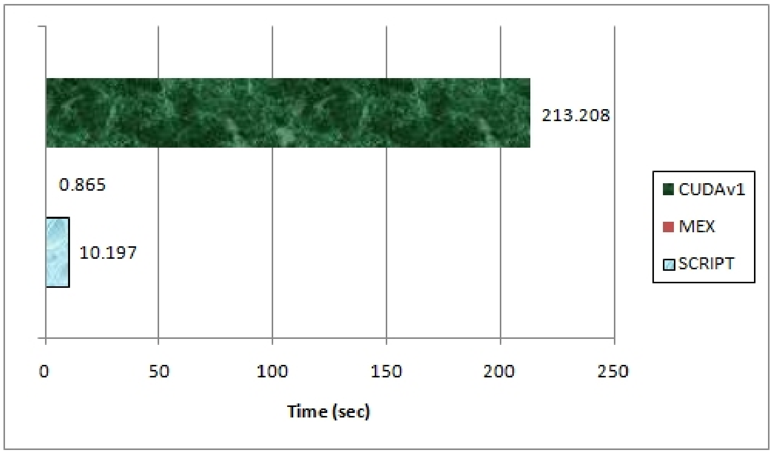

Figure 3: Results for simulating NLS using our script code, mex code, and version 1 of our CUDA mex code. We see that the CUDA code runs much more slowly than the non-CUDA codes. The timings record the time taken for all computations of $F(\Psi)$. The run takes 7543 time steps with a vector size of $N = 1249$. The GPU used is a very low-end CUDA card (8400GS) with 512MB memory with 1 multi-process unit containing 8 processing cores.

connectivity on a Windows or Linux PC, as well as an introduction to debugging and analysis tools, we refer the reader to Freddy Mcdonald's project report.

5 CUDA Mex Implementations of NLSE Simulations and Results

We now describe our CUDA implementations and show the results. We have successfully developed three versions of CUDA mex files, each one improving on the latter. All mex implementations were developed by Ron Caplan with support from the other team members.

5.1 Implementation 1: Single Call

Our first implementation was to directly port our mex file into a CUDA enabled version. However, in order for the GPU to evaluate $F(\Psi)$, the real and imaginary parts of Ψ (as well as $V(x)$) need to be transfered from the CPU to the GPU and then, after computation, must be transfered back to the CPU. This process must be done at every function call. Thus, we have memory transfers of five vectors of size N occurring four times for each time step. Since memory transfers are the slowest part of any CUDA code, we did not expect this code to give us any speed up. As can be seen in Fig. 3, not only was there no speedup, but the CUDA code performs much more slowly then the mex code (or even the original script code). In order to improve the performance, we rewrite the CUDA mex code to limit the number of memory transfers.

5.2 Implementation 2: Multiple Step

In order to minimize memory transfers, we change the code so that instead of just $F(\Psi)$ being computed in the mex file, a chunk of complete time steps are computed. Thus we divide our number of time steps into chunks, so that only five N sized memory transfers are computed per chunk of time steps. Obviously, the smaller the chunk size, the more transfers will be required. This creates a trade-off since typically, one would like to plot and analyze the wave function periodically throughout the simulation. However, the more frames one wants to view, the more memory transfers, and the less performance.

In the CUDA mex file, the time steps are computed using several CUDA kernels. This is necessary because since each evaluation of $F(\Psi)$ requires the previous evaluation's data, there is an issue when using vectors longer than the CUDA block size. This is because, even if one synchronizes the threads in the block, the edges of the block require data from the adjacent block's computation, which may not have yet been computed. Therefore, each step in the RK4+CD algorithm must be computed in its own kernel call. Another issue which requires that we use multiple kernel calls is that when running CUDA on a GPU that is also being used to control the primary display driver of the PC, there is a 2-5 second timeout. That is, if a kernel takes more than 2-5 seconds (depending on the platform) to run, the display driver cuts out and the code crashes. This prohibits us from making the time step loop for the chunk of steps within a kernel. However as previously discussed, this would not work regardless due to the synchronization issues (although perhaps with more thought, that problem could be overcome).

In addition to changing the CUDA mex code, we also rewrote our non-CUDA mex code to use chunks fo time steps as well. We did this in order to ensure fair timing comparisons. Our timing results are shown in Fig. 4. The results show that we do get some speedup using the CUDA mex code, but only for very large N, and even then, the speedup is very small. When we tested even higher values of N, the code crashes the systems display driver. However, this was not due to the memory becoming full, as we are able to run simulations with the same value of N using our third version of the CUDA code.

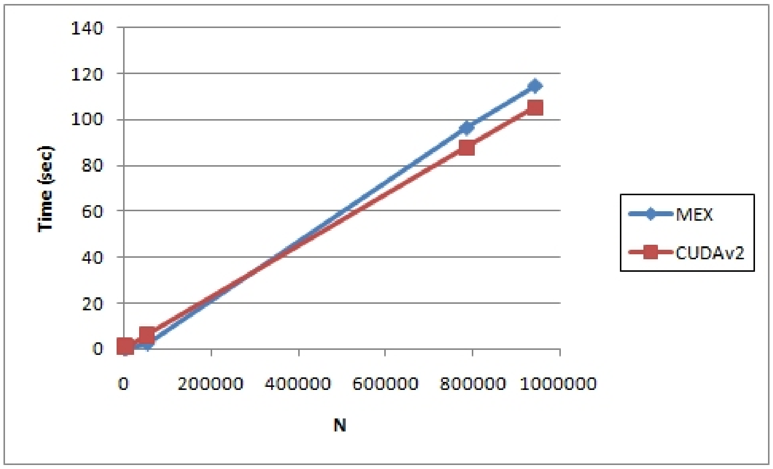

Figure 4: Results for simulating NLS using version 2 of our CUDA mex code. We see that the CUDA code runs faster than the non-CUDA code for very large N. The timings record the time taken for all computations of $F(\Psi)$. The run takes 500 time steps. The GPU used is the saem as in Fig. 3.

5.3 Implementation 3: Using Shared Memory

Since the performance of version 2 of our CUDA mex code was less than desirable, we wish to improve upon it. According to the CUDA documentation [6], accesses to the GPU's global memory have latency of over 100 times that of shared memory. This latency is also about 300 times the number of clock cycles than a multiplication or addition. Therefore, the use of shared memory is also suggested where possible. In our NLSE code, when computing $F(\Psi)$, there are a total of 22 global memory accesses per cell, which means (keeping boundary conditions in mind) $22(N-2)+2 = 22N-42$ total global accesses per $F(\Psi)$.

Our goal is to use shared memory to limit the number of global memory accesses. To do this, we set each computational block to allocate shared memory, and transfer the real and imaginary values of Ψ needed for that block into the shared memory. We then use the Ψ values in shared memory for our computation of Eq. 7. We cannot do this for all cells in the block because the cells at the edges of the block must access values that are contained in the adjacent cells which are outside the block sized vector of shared memory. Thus, for the boundary cells of the block, we use the global memory computation of Eq. 7 from version 2.

Shared memory has a maximum size of 16KB per block. Thus, we need to make sure that we set the block size low enough so that we can fit the shared memory that we need. The maximum size a CUDA block can be is 512 cells, however we determined that theoretically, only a block size of 256 or lower can be used for our problem. Unfortunately, setting the block size to 256 often causes the program to fail, so we instead set the block size to 128. It should be noted that this consideration is independent of N.

After these improvements, we have lowered the number of global memory accesses from $22N - 42$ to $4(N-2-\frac{N}{128})+22\frac{N}{128}+2 \approx 4.14N-6$, a significant decrease (about $17N$ total less global accesses). As this report was being prepared, it was realized that this number can be decreased even more since only the cell calls which cross the boundaries of the block are a problem, and therefore once can do a 'mixed' computation of Eq. 7.

Our timing results for version 3 of our CUDA mex code are given in Fig. 5. We see that for large N we observe decent speedup (about 4) in our simulations. For a GPU card that costs around \$30, we feel this is promising. Although having a value of $N \approx 1000000$ for a 1D problem is almost never done, this is significant for 3D problems, where even a modest grid size of $100 \times 100 \times 100$ yields 1000000 cells. Therefore we feel that the maximum benefit of using CUDA will be in 2D and 3D NLSE codes.

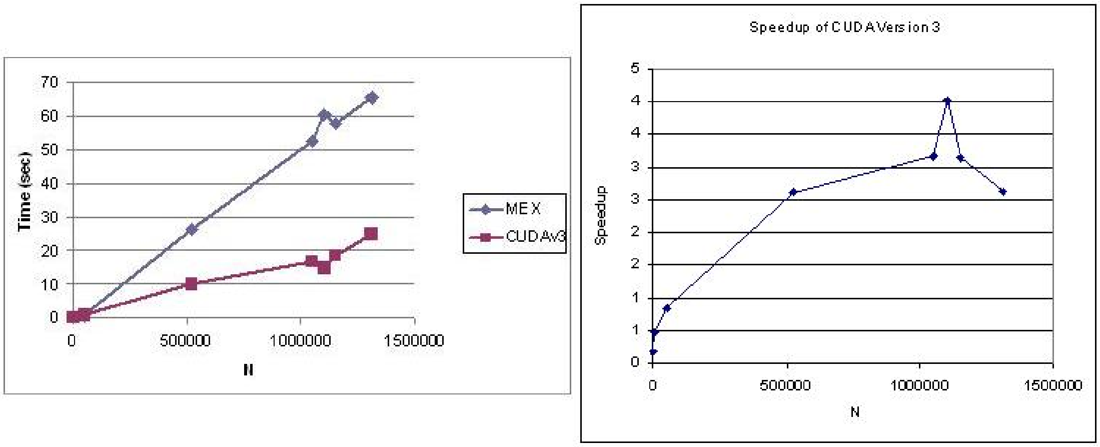

Figure 5: Results for simulating the NLSE using version 3 of our CUDA mex code. Left: The timing results using the time taken for all computations of $F(\Psi)$. Right: Speedup plot of the timings. We see that using shared memory has improved the performance of the code greatly, with a maximum speedup of around 4. The GPU used for the timings is the same as in Fig. 3.

We now report our timings using a better (and more expensive) GPU card. The new card (GTX275 with 896MB memory and 30MPs equaling 240 processor cores) is around \$220 and hence is still quite affordable when compared to the high-end Tesla cards which run into the thousands. Our results are shown in Fig. 6. We see that using the more advanced card greatly increases our speedup. Through other testing we have been able to get this speed up to be of order 100 times faster then the non-CUDA mex code. Since the mex code was shown to be order 10 faster then our original MATLAB script code, we have succeeded in producing a speedup of 1000 from our original script code!

Although we have achieved our objective of speeding up our NLSE simulations using CUDA, the development of the custom mex files was labor intensive, and not straight forward for someone with little C programming experience. Since one of our goals was to have our CUDA code methodology available for researchers without too much trouble, this difficulty is troubling. Although through the use of the documentation that our team has produced this issue is minimized, we nevertheless explored a different MATLAB implementation of CUDA called GPUmat.

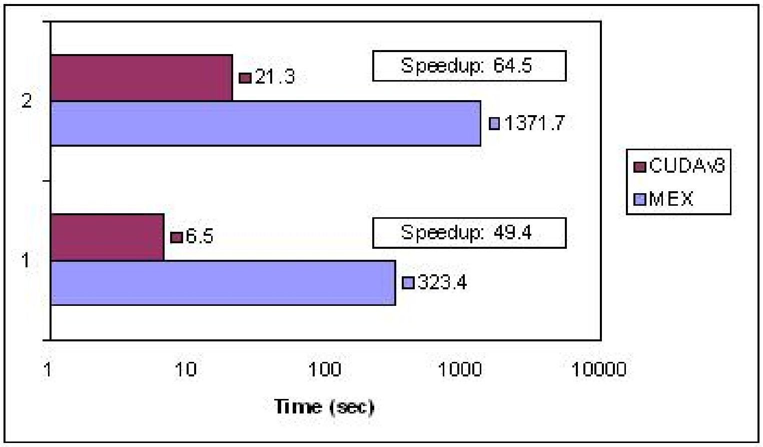

Figure 6: Results for simulating the NLSE using version 3 of our CUDA mex code on a better GPU card. Both simulations were run for 10000 time steps with one frame. Run 1 had $N = 131110$ while run 2 had $N = 524439$. The timings given are 'walltime', not just the computation of $F(\Psi)$. Even so, we see that the speedup is very significant even for modest values of N.

6 GPUmat

GPUmat is a library of custom CUDA-enabled mex files developed and maintained by the GP-you Group (`http://www.gp-you.org`) which allows a MATLAB user to almost seamlessly use CUDA from MATLAB. There is no compiling or C programming required for its use, making it easy for researchers to implement. For more details on GPUmat, its setup, its advantage and disadvantages, as well as our implementation of the NLSE code using it and the results, we refer the reader to Eunsil Baik's project report.

The conclusion reached there was that the GPUmat implementation of the NLSE codes ran much slower than even the original script code, thereby making its use inadvisable. However, the GP-you group as well as other groups are continuing to work on the development, so the final decision on the use of GPUmat and its equivalents is still open-ended.

7 Conclusion

Our primary goal in this project was to use CUDA with MATLAB to speed up computations of the NLSE. We feel this goal has been met. Our secondary goal was to allow the benefits of using CUDA to be easily accessible to researchers who do not have a lot of time on their hands for code development. This goal was not met as well as we would have liked. We now summarize our results and give suggestions for future work.

7.1 Summary of Results

The use of CUDA enabled GPU devices to speed up computation of the NLSE is highly recommended. By writing custom CUDA-enabled mex files for MATLAB, we were able to observe speedups of around 5 for a $30 range GPU, and speedups of over 100 for a $220 range GPU. It is our prediction that using Tesla GPUs would provide even greater speedup, drastically reducing computation time for researchers.

The development of the custom mex files does take some time and effort. It is not straight forward, nor is it easy to set up a CUDA environment. This may discourage some from implementing their codes using CUDA. However, we have provided a user manual for setting up CUDA as well as this, and our other team members reports in order to eliminate this setback. Another approach to using CUDA with MATLAB is to use GPUmat and its equivalents. Although we did not succeed in seeing any speedup using this method, the libraries are under constant development, and so it is possible that this easy to use method will become useful in the future.

7.2 Future Work

For practical use of CUDA for simulating the NLSE, future work needs to be done. A first step is to alter the code to be able to use double precision when using a GPU with the capability to do so. Also, the code must be expanded into two and three dimensional versions, as most research is in those domains. Finally, we feel that implementing other numerical schemes is vital as well. These could include implicit schemes, high-order compact schemes, and mimetic operators. These additions would make the use of CUDA extremely useful to a very large number of researchers.

References

[1] L. Debnath. *Nonlinear Partial Differential Equations for Scientists and Engineers.* Birkhauser Boston, New York, New York, 2 edition, 2005.

[2] P. G. Kevrekidis, D. J. Frantzeskakis and R. Carretero-González. *Emergent Nonlinear Phenomena in Bose-Einstein Condensates: Theory and Experiment.* Springer, New York, New York, 1 edition, 2007.

[3] 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.

[4] V.V. Afanasjev. Rotating ring-shaped bright solitons. *Phys. Rev. E*, **52** (1995) 3153.

[5] G. H. Golub and J. M. Ortega. *Scientific Computing and Differential Equations An Introduction to Numerical Methods.* Academic Press, San Diego, California, 2 edition, 1992.

[6] NVIDIA. *CUDA 2.3 Documentation*, 2009.