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

May 7th 2010

Ronald M. Caplan

San Diego State University

CSRC

Claremont Graduate University

Overview

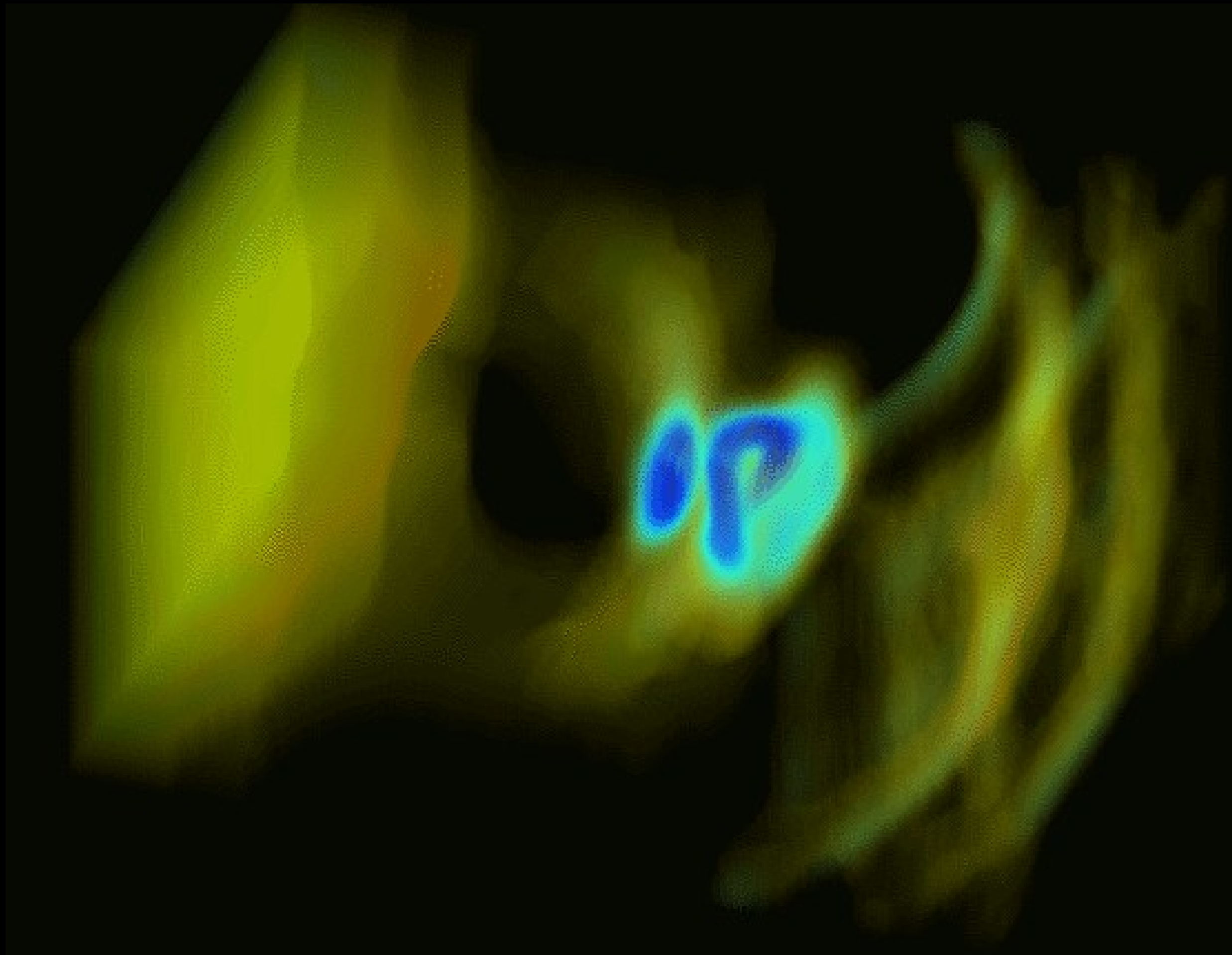
- Background and Purpose
- One-Dimensional Test Example
- GPU Computing
- NVIDIA CUDA API
- High Order Numerical Scheme
- Boundary Conditions
- CUDA Code Implementation
- Speedup Results
- Conclusion

Background and Purpose

$$i\Psi_t + a\nabla^2\Psi + (V(\mathbf{r}) + \mathbf{s}|\mathbf{\Psi}|^2)\mathbf{\Psi} = \mathbf{0}$$

- Nonlinear Schrödinger Equation (NLSE)
  - Bose-Einstein Condensates
  - Nonlinear optics.

- 3D vortex rings
- Need many 3D large simulations
- Want to speed up computations
  - High order schemes
  - Parallel programming
  - Visuals and easy analysis

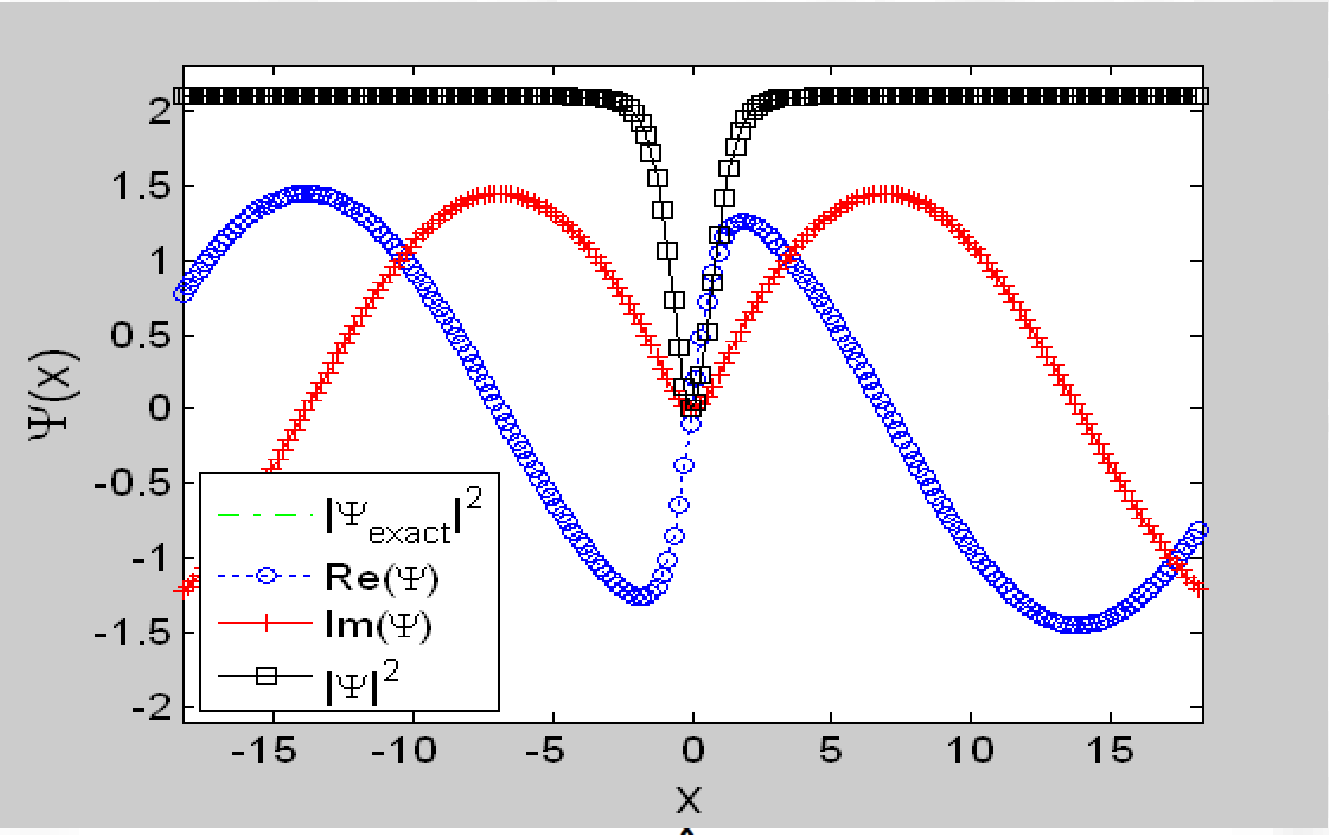


One-Dimensional Test Example

One-Dimensional NLSE $$i\Psi_t + a\frac{\partial^2\Psi}{\partial x^2} + (V(x) + s|\Psi|^2)\Psi = 0$$

Moving dark soliton solution with $V(x) = 0$

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



Constant density background

Parameters:

$a = 1.1$

$s = -1.1$

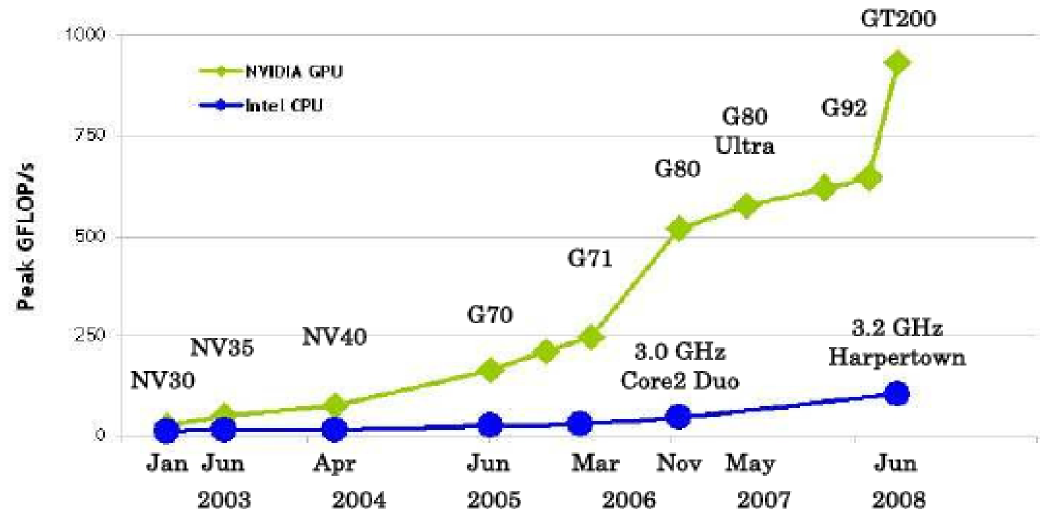
$\Omega = 2.1\,s$

$c = 0.5$

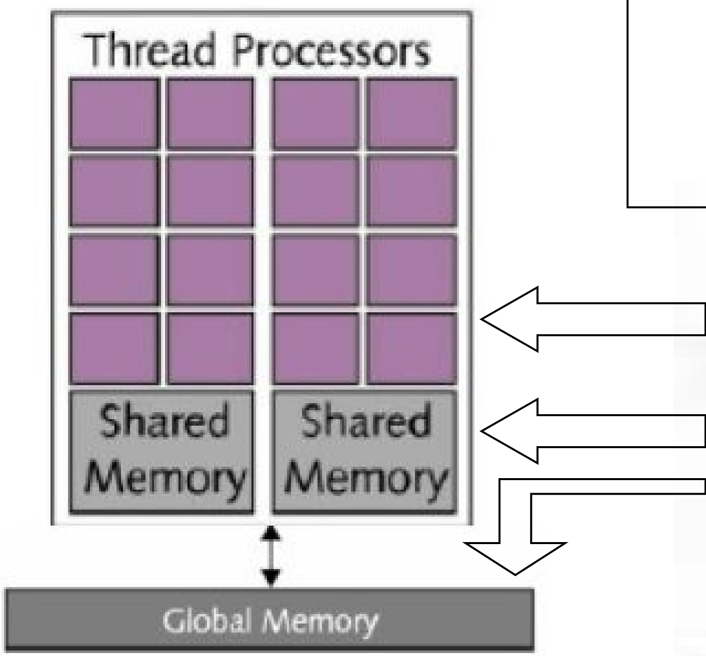
$t_{\text{end}} = 10$

GPU Computing

Graphical Processing Units
- Massively Parallel
- Have surpassed CPU FLOPS
- Very inexpensive compared to CPU clusters.



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	



Collection of Multiprocessors (MP), each with 8 ALUs

Each MP has small fast shared memory

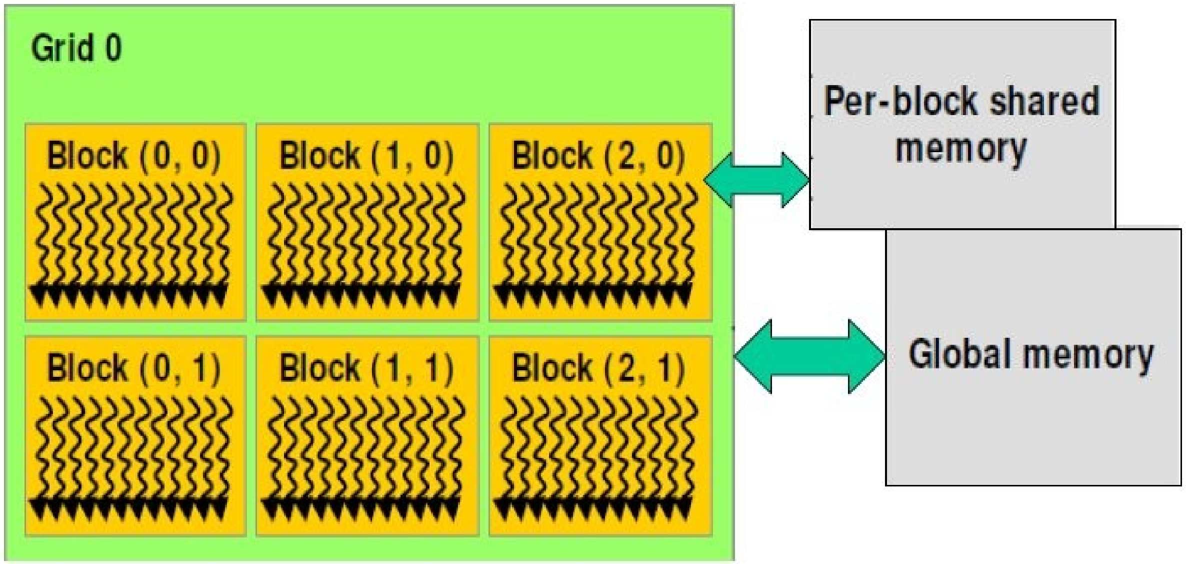
GPU card has large (slow) global memory (RAM)

Other memory (constant, texture, etc.)

NVIDIA CUDA API

NVIDIA C code extension (free!)
Allows low level access to GPU
Logic structure: Grids of Blocks of Threads

CUDA vs. OpenCL
FORTRAN Support



Threads instantiated through calls to a “kernel”
Thread synchronization within blocks
Each thread typically computes one cell of array or matrix
Each thread has access to per-thread local, per-block shared, and global variables.

High Order Finite Difference

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

Second order differencing:

$$\frac{\partial^2 \Psi_j}{\partial x^2} = \frac{\Psi_{j+1} - 2\Psi_j + \Psi_{j-1}}{h^2} - \frac{h^2}{12}\frac{\partial^4 \Psi_j}{\partial x^4} + O(h^4)$$

Take second derivative of differencing

$$\frac{\partial^4 \Psi_j}{\partial x^4} = \frac{\partial^2}{\partial x^2} D_j + O(h^2) = \frac{D_{j+1} - 2D_j + D_{j-1}}{h^2} + O(h^2)$$

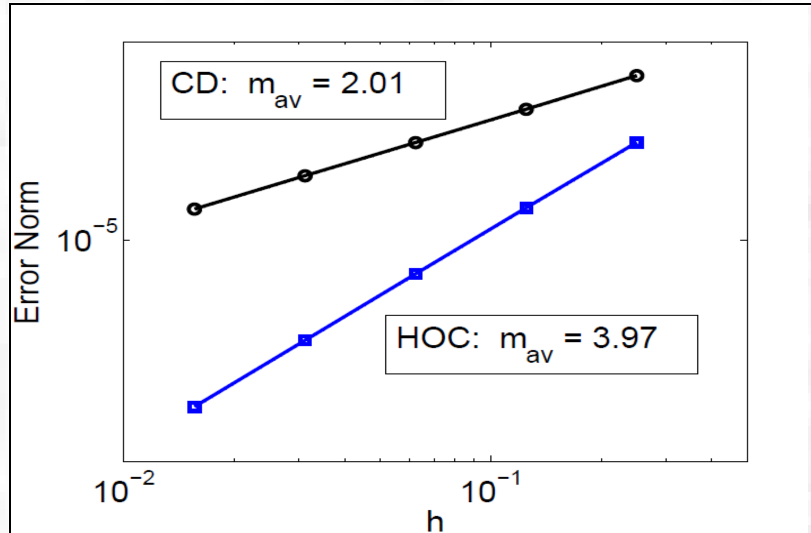
Fourth order approximation:

$$\frac{\partial^2 \Psi_j}{\partial x^2} \approx D_j - \frac{1}{12}(D_{j+1} - 2D_j + D_{j-1})$$

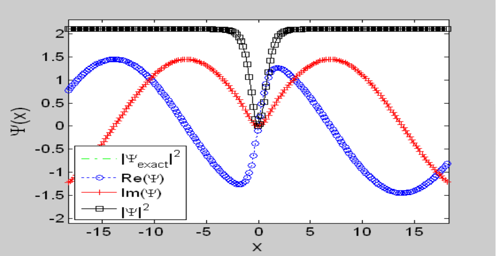
Two-Step High Order Compact Scheme (2SHOC)

$$D_j = \frac{\Psi_{j+1} - 2\Psi_j + \Psi_{j-1}}{h^2}$$

$$\frac{\partial^2 \Psi_j}{\partial x^2} \approx \frac{7}{6} D_j - \frac{D_{j+1} + D_{j-1}}{12}$$



MSD Boundary Conditions



Want simple boundary condition. Dirichlet?

Constant density at boundary: $$|\Psi_0|^2 = B$$

Separate real and imaginary parts: $$\frac{\partial}{\partial t}|\Psi_0|^2 = 0 \Rightarrow \Psi_R \frac{\partial \Psi_R}{\partial t} = -\Psi_I \frac{\partial \Psi_I}{\partial t}$$

Solution to ODEs, but need C: $$\frac{\partial \Psi_R}{\partial t} = C\Psi_I \quad \text{and} \quad \frac{\partial \Psi_I}{\partial t} = -C\Psi_R$$

details: substitutions… one-sided differencing… recombining…

(New?) Modulus-Squared Dirichlet boundary condition:

$$\boxed{\frac{\partial \Psi_0}{\partial t} \approx \left(\frac{\Psi_0}{\Psi_1}\right)\frac{\partial \Psi_1}{\partial t}}$$

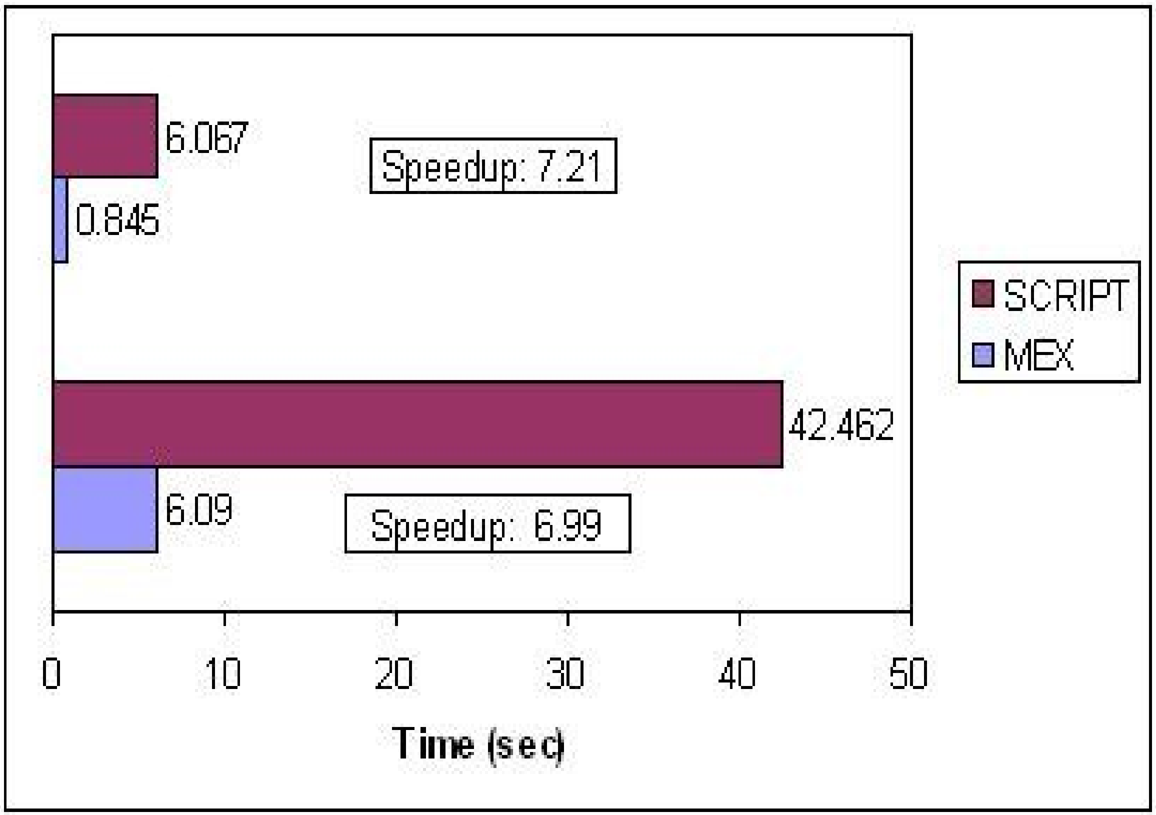
Already computed!

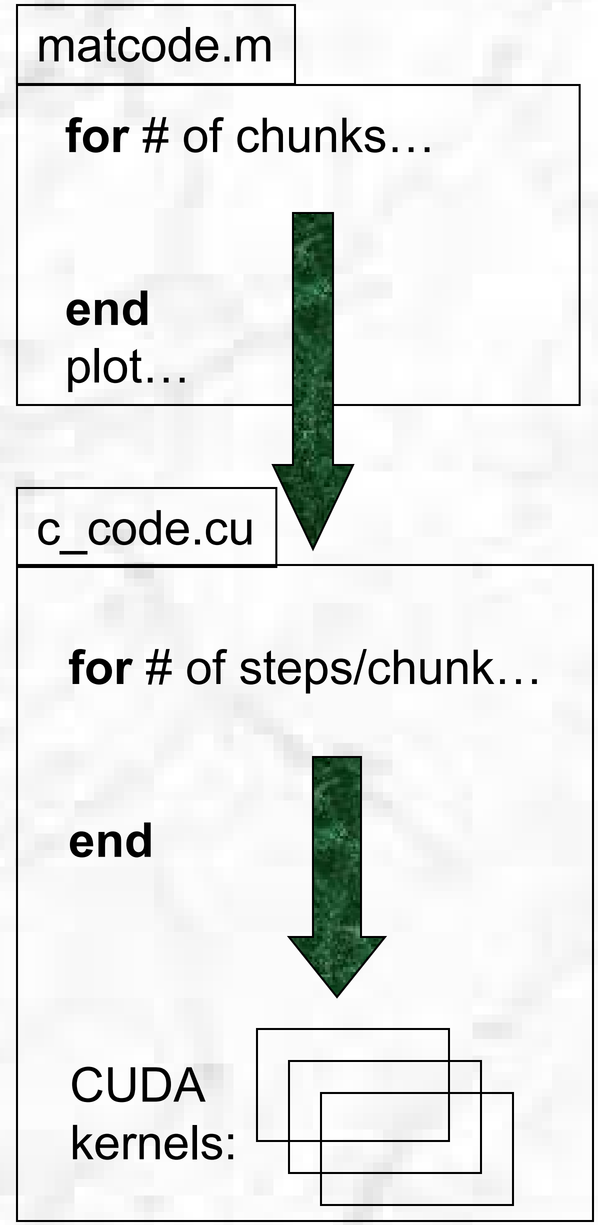
Works for any time-dependent complex PDE and in any dimension

Applying to NLSE: $$\nabla^2 \Psi_0 \approx \left[\frac{\nabla^2 \Psi_1}{\Psi_1} + \frac{V_1 - V_0}{a} + \frac{s}{a}\left(|\Psi_1|^2 - |\Psi_0|^2\right)\right]\Psi_0$$

CUDA Code Implementation

MATLAB: allows easy analysis and visuals. Can compile custom C-code MEX files that use CUDA with nvmex.





CUDA Code Implementation

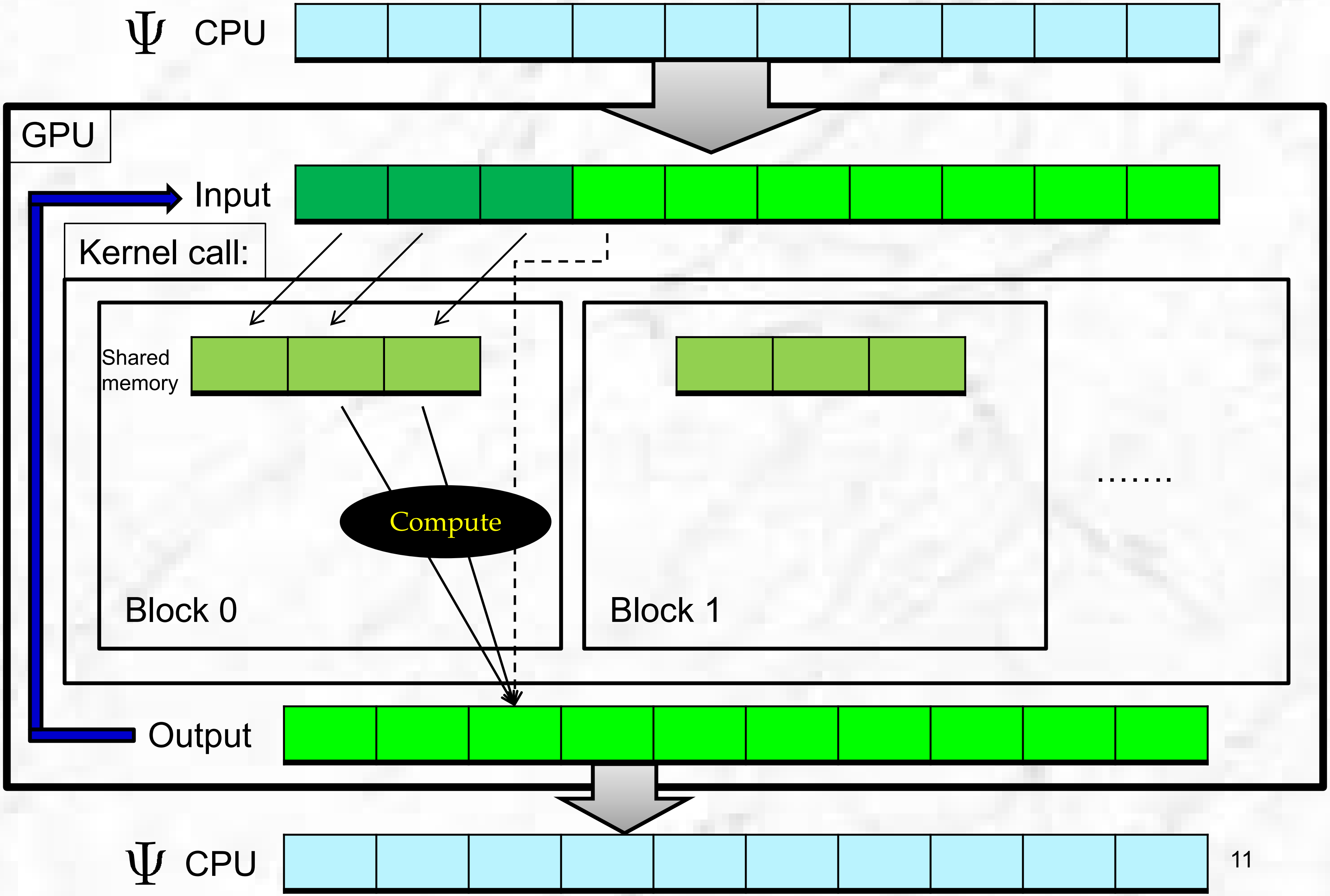
Vectors transferred to GPU, then do chunk of time steps:

$$D_\Psi = D(\Psi)$$
$$k_{tot} = F(\Psi, D_\Psi, V)$$
$$\Psi_{tmp} = \Psi + \frac{\Delta t}{2} k_{tot}$$
$$D_\Psi = D(\Psi_{tmp})$$
$$k_{tmp} = F(\Psi_{tmp},, D_\Psi, V)$$
$$k_{tot} = k_{tot} + 2k_{tmp}$$
$$\Psi_{tmp} = \Psi + \frac{\Delta t}{2} k_{tmp}$$
$$D_\Psi = D(\Psi_{tmp})$$
$$k_{tmp} = F(\Psi_{tmp}, D_\Psi, V)$$
$$k_{tot} = k_{tot} + 2k_{tmp}$$
$$\Psi_{tmp} = \Psi + \Delta t k_{tmp}$$
$$D_\Psi = D(\Psi_{tmp})$$
$$k_{tmp} = F(\Psi_{tmp}, D_\Psi, V)$$
$$\Psi = \Psi + \frac{\Delta t}{6}(k_{tot} + k_{tmp})$$

```
for (j = 0; j<chunk_size; j++)
{
  compute_dx2 <<<dimGrid,dimBlock>>>(dx2r,dx2i,Uoldr_gpu,U
  compute_F   <<<dimGrid,dimBlock>>>(ktotr,ktoti,Uoldr_gpu
  /*Evaluate Utmp:*/
  vec_add     <<<dimGrid,dimBlock>>>(Utmpr,Utmpi,Uoldr_gpu
  compute_dx2 <<<dimGrid,dimBlock>>>(dx2r,dx2i,Utmpr,Utmpi
  compute_F   <<<dimGrid,dimBlock>>>(ktmpr,ktmpi,Utmpr,Utm
  /*Collect k and evaluate new Utmp*/
  double_vec_add   <<<dimGrid,dimBlock>>>(ktotr, ktoti,kto
                                          Utmpr,Utmpi,Uold
  compute_dx2 <<<dimGrid,dimBlock>>>(dx2r,dx2i,Utmpr,Utmpi
  compute_F   <<<dimGrid,dimBlock>>>(ktmpr,ktmpi,Utmpr,Utm
  /*Collect k and evaluate new Utmp again*/
  double_vec_add   <<<dimGrid,dimBlock>>>(ktotr, ktoti,kto
                                          Utmpr,Utmpi,Uold
  compute_dx2 <<<dimGrid,dimBlock>>>(dx2r,dx2i,Utmpr,Utmpi
  compute_F   <<<dimGrid,dimBlock>>>(ktmpr,ktmpi,Utmpr,Utm
  /*Collect k and evaluate new step*/
  vec_add3  <<<dimGrid,dimBlock>>>(Uoldr_gpu,Uoldi_gpu,kto
}
/*Make sure everything is computed*/
cudaThreadSynchronize();
```

Transfer vector back to CPU for analysis and plotting.

CUDA Code Implementation



CUDA Code Implementation

Simple CUDA kernel code

```
__global__ void vec_add(double* Ar,double* Ai,double* Br,double* Bi,
                        double* Cr,double* Ci,double   k,int N)
{
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if(i<N)
    {
        Ar[i]  = Br[i] + k*Cr[i];
        Ai[i]  = Bi[i] + k*Ci[i];
    }
}
```

Global memory accesses

CUDA Code Implementation

CUDA kernel using shared memory to compute D_j

```
/*If cell is not boundary...*/
if (j >0 && j < N-1)
{
    /*If cell is not on shared memory boundary...*/
    if (i > 0 && i < blockDim.x-1)
    {
        dx2r[j] = (sUtmpr[i+1] - 2*sUtmpr[i] + sUtmpr[i-1])/h2;
        dx2i[j] = (sUtmpi[i+1] - 2*sUtmpi[i] + sUtmpi[i-1])/h2;
    }
    /*If on LHS of shared memory boudary, have to use j-1 global*/
    if(i==0){
        dx2r[j] = (sUtmpr[i+1] - 2*sUtmpr[i] + Utmpr[j-1])/h2;
        dx2i[j] = (sUtmpi[i+1] - 2*sUtmpi[i] + Utmpi[j-1])/h2;
    }
    /*If on RHS of shared memory boudary, have to use j+1 global*/
    if(i==blockDim.x-1){
        dx2r[j] = (Utmpr[j+1] - 2*sUtmpr[i] + sUtmpr[i-1])/h2;
        dx2i[j] = (Utmpi[j+1] - 2*sUtmpi[i] + sUtmpi[i-1])/h2;
    }
}                                                               illed.*/
/*Boundery Conditions*/
if(j == 0 || j == N-1){
```

Global memory accesses

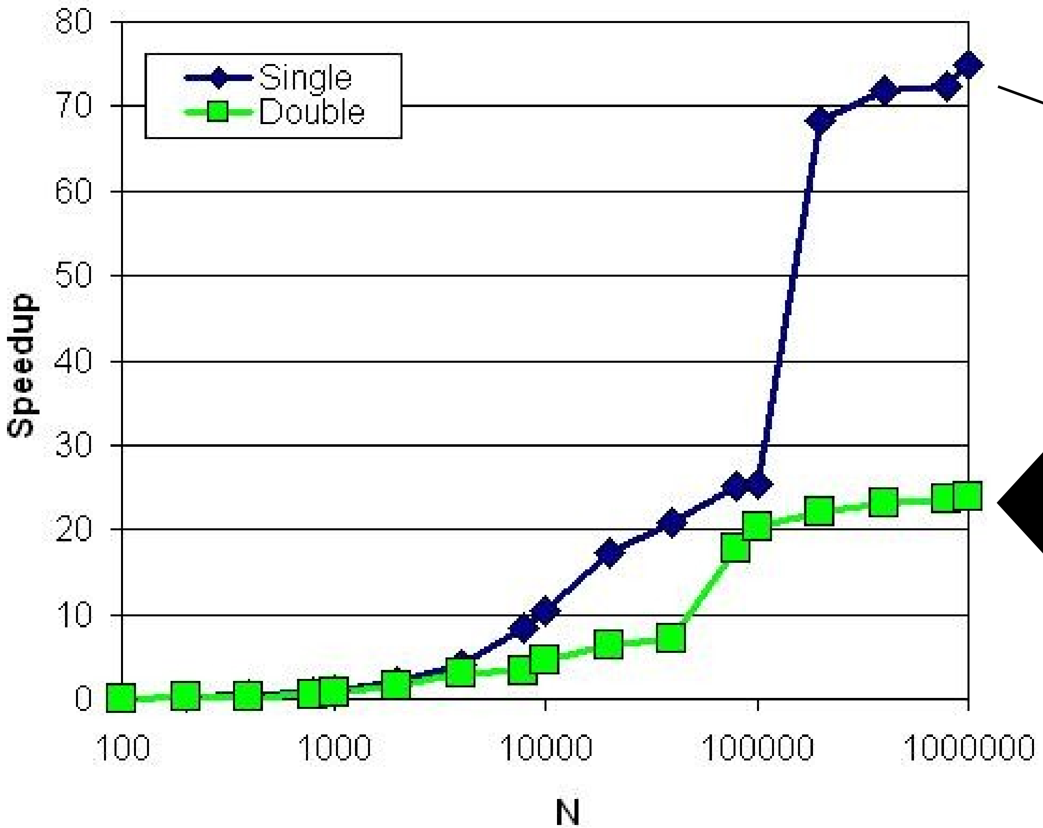
Shared memory accesses (much faster)

Speedup Results

NVIDIA GeForce GTX 260
192 Cores, 896MB RAM
Price: ≈ $200

(Single precision)

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**



CPU: ~ 1 hour
GPU: ~ 1 **minute**

Double precision has noticeable performance hit

Conclusion

- Using GPU for simulations is very useful and cost efficient
- Large speedup observed even for a computationally simple numerical scheme
- MSD boundary condition simple and effective
- Plans to develop 2D and 3D versions of the code.