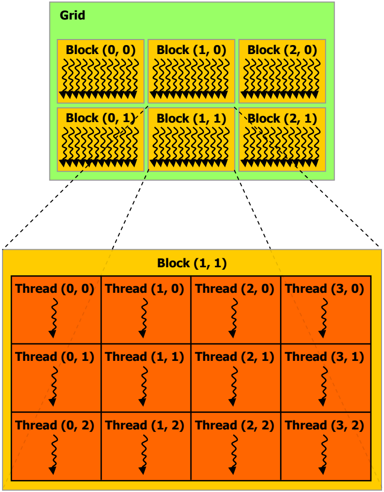
VSSUP PyCuda workshop

Bogdan Opanchuk

July 6th, 2012

A bit about Cuda



Computation unit: thread. Each thread executes the same code (parameterized by thread ID).

Threads can interact with each other inside one block.

Blocks are executed independently.

simple.py: imports

```
# initialize Cuda on the first available device
import pycuda.autoinit

# import some array manimulation functions
from pycuda.gpuarray import to_gpu, empty_like

# import basic kernel creation class
from pycuda.elementwise import ElementwiseKernel

# import python numeric library
import numpy as np
```

simple.py: arrays on CPU

Create two arrays with random numbers:

```
a = np.random.randn(400).astype(np.float32)
b = np.random.randn(400).astype(np.float32)
```

`np.random` is a module,
`randn` is a function,
`randn(400)` creates an array with 400 random numbers,
`astype` is a method of the array which casts it to specified type,
and `np.float32` is a single-precision floating-point type.

Warning: by default numpy arrays have type `float64`

simple.py: kernel

If you do not need interaction between threads, there is a shortcut

```
mul = ElementwiseKernel(
    ”float *dest, float *a, float *b”,
    ”dest[i] = a[i] * b[i]”)
```

Behind the scenes it is being translated to

```
__global__ void mul(float *dest, float *a,
        float *b, int n)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) {
        dest[i] = a[i] * b[i];
    }
}
```

simple.py: arrays on GPU

```
a_gpu = to_gpu(a)
b_gpu = to_gpu(b)
dest_gpu = empty_like(a_gpu)
```

`empty_like(a)` creates an array in video memory with the same type and size as `a`.
`to_gpu(a)` does the same, and copies `a`'s contents to the new array.

N.B.: numpy has the function `empty_like` too, which does the same thing, except that it creates the new array in RAM.

simple.py: calculation

```
mul(dest_gpu, a_gpu, b_gpu)
print (a * b - dest_gpu.get())
```

`mul` invokes the kernel with block and grid size deduced from `dest_gpu` size.
Arrays returned from `to_gpu` and `empty_like` belong to class `GPUArray` and have `get()` method, which returns their contents as a numpy array.
Numpy arrays support arithmetical operations, much like matrices in MatLab.

Application: GPE

Dimensionless GPE for the single-component BEC:

$$\frac{d\psi}{dt}\phi(x,t) = \frac{i}{2}\frac{d^2\psi}{dx^2} - iV\psi - iU|\psi|^2\psi - \gamma\psi,$$

where $\psi(x,t)$ is the wavefunction, $V(x) = vx^2/2$ is the trap potential, U is the strength of nonlinear interaction, and γ is the linear loss coefficient.

Integration

We will use the **semi-implicit** integration method (**"SI"** in XMDS). Given the equation

$$d\psi(x,t) = f(x,\psi,dt),$$

on each step with $t = t_n$ and known $\psi(x,t_n) \equiv \psi_n$ we perform $K-1$ iterations to the middle of the step:

$$\psi_{n+1,k} = \psi_{n,0} + f(x,\psi_{n+1,k-1}, dt/2),$$

starting from $\psi_{n+1,0} \equiv \psi_n$. The last iteration is made using the full step:

$$\psi_{n+1,K} = \psi_{n,0} + f(x,\psi_{n+1,K-1}, dt).$$

Fourier space propagation

In our case

$$f = \frac{i}{2}\frac{d^2\psi}{dx^2}dt + \left(-iV\psi - iU|\psi|^2\psi - \gamma\psi\right)dt$$

Problem: How do we calculate d^2/dx^2?
In Fourier space derivatives are just multiplications:

$$F[df/dx] = ikF[f](k)$$

Therefore

$$\frac{d^2\psi}{dx^2} = F^{-1}\left[-k^2F[\psi]\right].$$

bec1.py: coordinate spaces

```
x = np.linspace(domain[0], domain[1], lattice_size)
k = np.fft.fftfreq(lattice_size, dx) * 2.0 * np.pi
```

`linspace(min, max, points)` returns an array with `points` values ranging from `min` to `max`.
`fftfreq(points, step)` returns an array with frequencies corresponding to points in Fourier space.

bec1.py: FFT

Creation:

```
plan = Plan((lattice_size,), dtype=np.complex64)
```

Usage:

```
plan.execute(source, dest)
plan.execute(source, dest, inverse=True)
```

If `dest` is not given, the FFT is performed inplace.

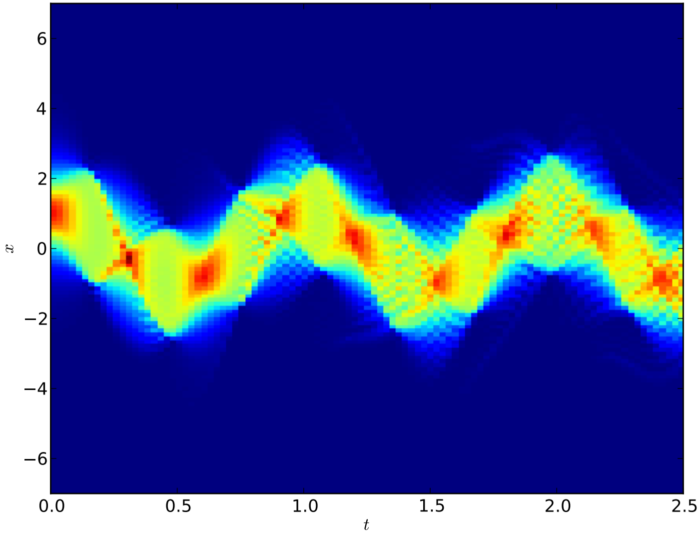
bec1.py: copying in videomemory

```
memcpy_dtod(psi_copy_gpu.gpudata,
    psi_gpu.gpudata, psi_gpu.nbytes)
```

This function is more low-level than `GPUArray` methods, and needs to be given actual pointers to video memory (`.gpudata`) and buffer length (`.nbytes`).

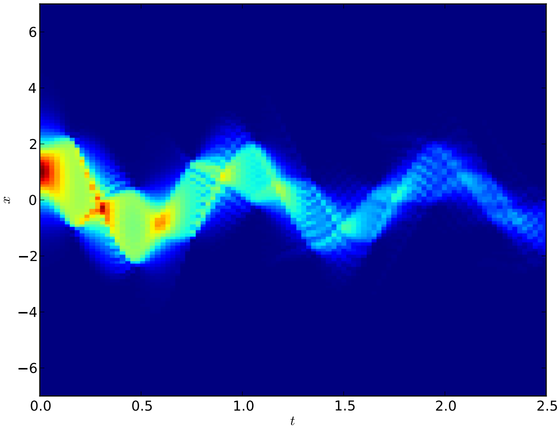
Results: bouncing soliton

Initial condition: $\phi(x,0) = \alpha\,\mathrm{sech}(x-d)$



Bouncing soliton with losses

Exercise: add linear losses to the code (see bec2.py for help). Result should look something like this:



Hard mode: from GPE to SDE

Dimensionless stochastic equation:

$$\frac{d\psi}{dt}\phi(x,t) = \frac{i}{2}\frac{d^2\psi}{dx^2} - iV\psi - iU|\psi|^2\psi - \gamma\psi + \sqrt{\gamma}Z(x,t),$$

where Z is a c-valued Wiener process.
Technically,

$$Z(x,t) = \sqrt{\frac{1}{2}}(\eta_1(x,t) + i\eta_2(x,t)),$$

where η_1 and η_2 are sets of normally distributed random numbers with variance $1/\Delta x \Delta t$ on a finite grid

Initial state and density

Initial state is a classical state plus Gaussian noise:

$$\psi(x,0) = \psi_0(x) + \frac{1}{\sqrt{2}}\zeta(x),$$

where $\zeta(x)$ are normally distributed complex random numbers with variance $1/\Delta x$ for each point on the grid.

Density can be obtained from the solution of the SDE as

$$n(x,t) = \langle|\psi(x,t)|^2\rangle - \frac{1}{2\Delta x}$$

bec3.py: random values

`psi` array new shape: `(paths, lattice_size)`.
Normally distributed random values can be obtained as:

```
np.random.normal(size=(paths, lattice_size))
```

C-valued randoms are a normalized sum of two real-valued arrays:

```
re = np.random.normal(size=(paths, lattice_size))
im = np.random.normal(size=(paths, lattice_size))
random_normals = (re + 1j * im) / np.sqrt(2)
```

bec3.py: initial state

```
initial_noise = random_normals / np.sqrt(2 * dx)
psi0 = np.tile(psi0, (paths, 1)) + initial_noise
psi0 = psi0.astype(np.complex64)
```

First line scales random values.
`np.tile(psi0, (paths, 1))` creates a 2-dimensional array of size `(paths, lattice_size)` with repeating rows.
The combination of the tiled classical state and the noise is our initial state.

bec3.py: randoms on GPU

Getting randoms from numpy during the integration is too slow. We have to use the GPU generator.

Importing (this one is a wrapper for CURANDOM)

```
from pycuda.curandom import
    XORWOWRandomNumberGenerator as RNG
```

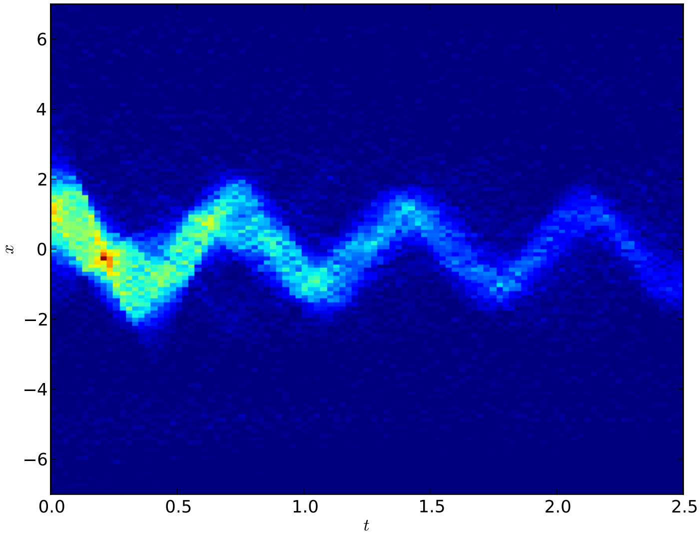
Initializing:

```
rng = RNG()
```

Filling the prepared array (generates normally distributed real randoms with variance 1)

```
rng.fill_normal(noise_gpu)
```

Results: noisy soliton

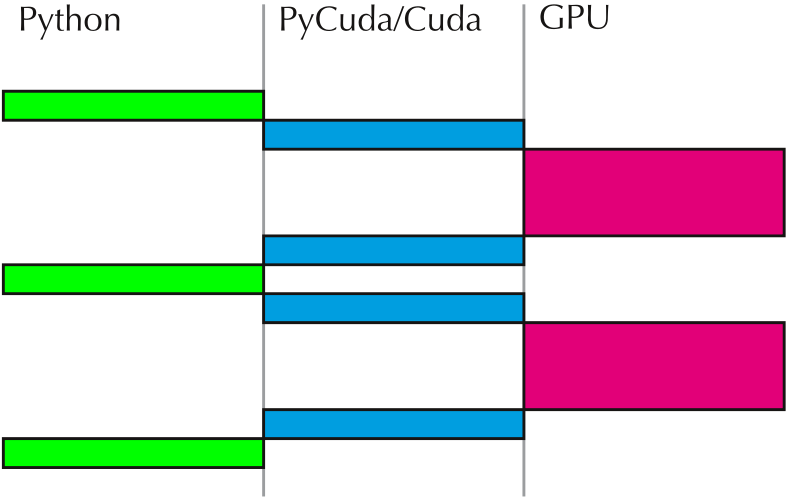


What to add?

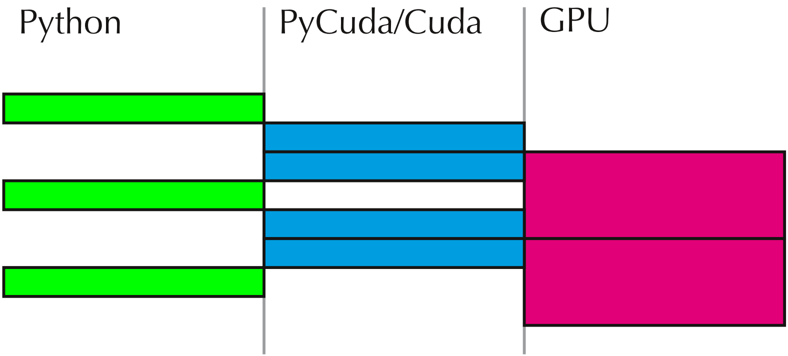
- Asynchronous calls
- Timing
- Error estimation
- Double precision

See **bec4.py** with all this implemented

Synchronous calls



Asynchronous calls



bec4.py: asynchronous calls

```
from pycuda.driver import Stream, memcpy_dtod_async
stream = Stream()
```

Kernel call:

```
kPropagate(psi_kspace_gpu, prop_k_gpu, stream=
    stream)
```

Asynchronous copy:

```
cuda.memcpy_dtod_async(psi_copy_gpu.gpudata,
    psi_gpu.gpudata, psi_gpu.nbytes,
    stream=stream)
```

Synchronize with the stream: `stream.synchronize()`

bec4.py: timing

```
import time

t1 = time.time()
# do something
t2 = time.time()
print t2 - t1 # returns time in seconds
```

bec4.py: double precision

Double precision numpy types: `float64` and `complex128`.
Or, use metaprogramming:

```
from pycuda.compyte.dtypes import dtype_to_ctype
```

```
complex_dtype = np.complex128
print "{cname} *psi".format(
    cname=dtype_to_ctype(complex_dtype))
```

returns `"pycuda::complex<double> *psi"`

Results: noisy soliton, 1024 paths