GPU Programming using Python
(PyCUDA and PyOpenCL)

Ki-Hwan Kim

Department of Physics, Korea University

2010/5/28

Outline

Outline

Scientific computing environments

Matlab, Mathematica, Maple, ...

- simple and clean syntax
- high productivity
- tight integration of calculation and visualization

=> Do not work for some problems, at least not in an easy way

Traditional languages (Fortran, C, C++, JAVA)

- flexible and versatile
- high performance

=> Complicated work, Low productivity

Scientific computing environments

Matlab, Mathematica, Maple, ...

- simple and clean syntax
- high productivity
- tight integration of calculation and visualization

=> Do not work for some problems, at least not in an easy way

Traditional languages (Fortran, C, C++, JAVA)

- flexible and versatile
- high performance

=> Complicated work, Low productivity

Scientific computing environments

Matlab, Mathematica, Maple, ...

- simple and clean syntax
- high productivity
- tight integration of calculation and visualization

=> Do not work for some problems, at least not in an easy way

Traditional languages (Fortran, C, C++, JAVA)

- flexible and versatile
- high performance

=> Complicated work, Low productivity

Scientific computing environments

Matlab, Mathematica, Maple, ...

- simple and clean syntax
- high productivity
- tight integration of calculation and visualization

=> Do not work for some problems, at least not in an easy way

Traditional languages (Fortran, C, C++, JAVA)

- flexible and versatile
- high performance

=> Complicated work, Low productivity

Scientific computing environments (Cont.)

Scripting using Python

- simple and clean syntax
- gluing other favorate simulation, visualization, data analysis
- high productivity
- high performance (gluing Fortran, C)
- Easy integration with GPU computing (PyCUDA, PyOpenCL)

=> Build my own Matlab-like scientific computing environment, tailored to my specific needs with high performance

Scientific computing environments (Cont.)

Scripting using Python

- simple and clean syntax
- gluing other favorate simulation, visualization, data analysis
- high productivity
- high performance (gluing Fortran, C)
- Easy integration with GPU computing (PyCUDA, PyOpenCL)

=> Build my own Matlab-like scientific computing environment, tailored to my specific needs with high performance

Outline

Mathematical Formula

Gaussian and Sin functions

$$\begin{cases} a(x) &= e^{-\frac{x^2}{2}} \\ b(x) &= \sin(5x) \end{cases}$$

$$\Rightarrow c(x) = a(x) \times b(x) = e^{-\frac{x^2}{2}} \sin(5x)$$

Discretize Condition

$$-5 \leq x < 5 \quad and \quad \Delta_x = 0.01$$

$$\rightarrow n = 1000$$

Mathematical Formula

Gaussian and Sin functions

$$\begin{cases} a(x) & = e^{-\frac{x^2}{2}} \\ b(x) & = \sin(5x) \end{cases}$$

$$\Rightarrow c(x) = a(x) \times b(x) = e^{-\frac{x^2}{2}} \sin(5x)$$

Discretize Condition

$$-5 \leq x < 5 \quad and \quad \Delta_x = 0.01$$

$$\rightarrow n = 1000$$

Outline

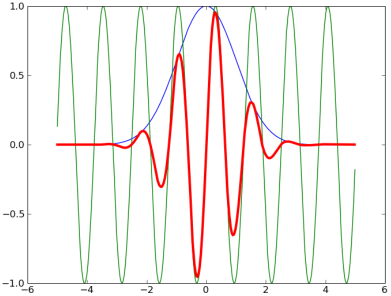
1. Why Python for scientific computations?
   - Scientific computing environments
2. Example 1) 1-D Array Multiplication
   - Formula
   - Python, PyCUDA, PyOpenCL
   - PyCUDA vs PyOpenCL
3. Example 2) 2-D Wave Equation
   - Formula
   - Python, PyCUDA
   - Shared Memory Optimization
   - Utilize Multi-GPUs with MPI

Python

```
#!/usr/bin/env python
import numpy as np

x = np.arange(-5,5,0.01,'f')
a = np.exp(-(x**2)/2)
b = np.sin(5*x)

c = a*b
```

Python (Plot the graph)

```
import matplotlib.pyplot as pl

pl.plot(x,a,x,b)
pl.plot(x,c,linewidth=3)
pl.savefig('./pics/mul_array.png')
pl.show()
```



PyCUDA

```
#!/usr/bin/env python
import numpy as np

x = np.arange(-5,5,0.01,'f')
a = np.exp(-(x**2)/2)
b = np.sin(5*x)

import pycuda.driver as cuda
import pycuda.autoinit

a_gpu = cuda.to_device(a)
b_gpu = cuda.to_device(b)
c_gpu = cuda.mem_alloc(a.nbytes)

gpu_mul(np.int32(a.size), a_gpu, b_gpu, c_gpu,
          block=(256,1,1),grid=(4,1))

c = np.zeros_like(a)
cuda.memcpy_dtoh(c, c_gpu)
```

PyCUDA (Cont.)

```
kernels = """
  __global__ void multiply(int nx, float *a,
        float *b, float *c){

    int idx = threadIdx.x;

    if(idx<nx) c[idx] = a[idx]*b[idx];
  }
"""

from pycuda.compiler import SourceModule
mod = SourceModule(kernels)
gpu_mul = mod.get_function("multiply")
```

PyOpenCL

```
#!/usr/bin/env python
import numpy as np

x = np.arange(-5,5,0.01,'f')
a = np.exp(-(x**2)/2)
b = np.sin(5*x)

import pyopencl as cl
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf=cl.mem_flags
a_gpu = cl.Buffer(ctx, mf.COPY_HOST_PTR, hostbuf=a)
b_gpu = cl.Buffer(ctx, mf.COPY_HOST_PTR, hostbuf=b)
c_gpu = cl.Buffer(ctx, mf.COPY_HOST_PTR, size=a.nbytes)

gpu_mul(queue, (a.size,) np.int32(a.size), a_gpu, b_gpu, c_gpu)

c = np.zeros_like(a)
cl.enqueue_read_buffer(queue, c_gpu, c)
```

PyOpenCL (Cont.)

```
kernels = """
  __kernel void multiply(int nx, __global float *a,
         __global float *b, __global float *c) {

    int idx = get_global_id(0);

    if(idx<nx) c[idx] = a[idx]*b[idx];
  }
"""

prg = cl.Program(ctx, kernels).build()
gpu_mul = prg.multiply
```

Outline

1. Why Python for scientific computations?
   - Scientific computing environments
2. Example 1) 1-D Array Multiplication
   - Formula
   - Python, PyCUDA, PyOpenCL
   - PyCUDA vs PyOpenCL
3. Example 2) 2-D Wave Equation
   - Formula
   - Python, PyCUDA
   - Shared Memory Optimization
   - Utilize Multi-GPUs with MPI

PyCUDA vs PyOpenCL

```
import pycuda.driver as cuda
import pycuda.autoinit

a_gpu = cuda.to_device(a)
b_gpu = cuda.to_device(b)
c_gpu = cuda.mem_alloc(a.nbytes)
gpu_mul(np.int32(a.size), a_gpu, b_gpu, c_gpu,
        block=(256,1,1),grid=(4,1))
cuda.memcpy_dtoh(c, c_gpu)
```

```
import pyopencl as cl
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf=cl.mem_flags
a_gpu = cl.Buffer(ctx, mf.COPY_HOST_PTR, hostbuf=a)
b_gpu = cl.Buffer(ctx, mf.COPY_HOST_PTR, hostbuf=b)
c_gpu = cl.Buffer(ctx, mf.COPY_HOST_PTR, size=a.nbytes)
gpu_mul(queue, (a.size,) np.int32(a.size), a_gpu, b_gpu, c_gpu)
cl.enqueue_read_buffer(queue, c_gpu, c)
```

PyCUDA vs PyOpenCL (Cont.)

```
kernels = """
  __global__ void multiply(int nx, float *a,...) {
    int idx = threadIdx.x;
    if(idx<nx) c[idx] = a[idx]*b[idx];
  }
"""

from pycuda.compiler import SourceModule
mod = SourceModule(kernels)
gpu_mul = mod.get_function("multiply")
```

```
kernels = """
  __kernel void multiply(int nx, __global float *a,...) {
    int idx = get_global_id(0);
    if(idx<nx) c[idx] = a[idx]*b[idx];
  }
"""

prg = cl.Program(ctx, kernels).build()
gpu_mul = prg.multiply
```

Outline

Mathematical Formula

2-D Wave Equation

$$\frac{\partial^2 f}{\partial t^2} = c^2 \nabla^2 f$$
$$\rightarrow \frac{\partial^2 f}{\partial t^2} = c^2 \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right)$$

Simple Finite-Difference Scheme

$$\frac{f_{i,j}^{n+1} - 2f_{i,j}^{n} + f_{i,j}^{n-1}}{\Delta_t^2} = c_{i,j}^2 \left(\frac{f_{i+1,j}^{n} - 2f_{i,j}^{n} + f_{i-1,j}^{n}}{\Delta_x^2} + \frac{f_{i,j+1}^{n} - 2f_{i,j}^{n} + f_{i,j-1}^{n}}{\Delta_y^2} \right)$$

$$f_{i,j}^{n+1} = \tilde{c}_{i,j}^2 \left(f_{i+1,j}^{n} + f_{i-1,j}^{n} + f_{i,j+1}^{n} + f_{i,j-1}^{n} - 4f_{i,j}^{n} \right) + 2f_{i,j}^{n} - f_{i,j}^{n-1}$$

Mathematical Formula

2-D Wave Equation

$$\frac{\partial^2 f}{\partial t^2} = c^2 \nabla^2 f$$
$$\rightarrow \frac{\partial^2 f}{\partial t^2} = c^2 \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right)$$

Simple Finite-Difference Scheme

$$\frac{f_{i,j}^{n+1} - 2f_{i,j}^{n} + f_{i,j}^{n-1}}{\Delta_t^2} = c_{i,j}^2 \left(\frac{f_{i+1,j}^{n} - 2f_{i,j}^{n} + f_{i-1,j}^{n}}{\Delta_x^2} + \frac{f_{i,j+1}^{n} - 2f_{i,j}^{n} + f_{i,j-1}^{n}}{\Delta_y^2} \right)$$

$$f_{i,j}^{n+1} = \tilde{c}_{i,j}^2 \left(f_{i+1,j}^{n} + f_{i-1,j}^{n} + f_{i,j+1}^{n} + f_{i,j-1}^{n} - 4f_{i,j}^{n} \right) + 2f_{i,j}^{n} - f_{i,j}^{n-1}$$

Outline

Python

```
import numpy as np

c = np.ones((1000,1000),'f')*0.25
f = np.zeros_like(c)
g = np.zeros_like(c)

ii = (slice(1,-1), slice(1,-1))
for tn in xrange(1000):
    g[400,500] += np.sin(0.1*tn)
    f[ii] = c[ii]*(g[2:,1:-1]+g[:-2,1:-1]+g[1:-1,2:]+g[1:-1,:-2]
                -4g[ii])+2g[ii]-f[ii]
    g[ii] = c[ii]*(f[2:,1:-1]+f[:-2,1:-1]+f[1:-1,2:]+f[1:-1,:-2]
                -4f[ii])+2f[ii]-g[ii]
```

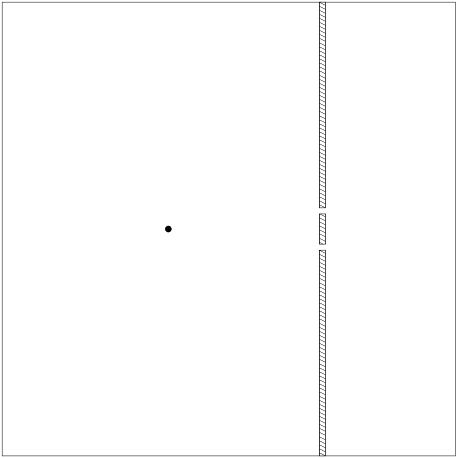
PyCUDA

```
kernels = """
  __global__ void update_src(int idx, int tn, float *g) {
    g[idx] += sin(0.1*tn);
  }

  __global__ void update(int nx, int ny, float *c, float *f, float *g) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    int i = idx/ny, j = idx%ny;

    if(i>0 && j>0 && i<nx-1 && j<ny-1)
      f[idx] = c[idx]*(g[idx+ny]+g[idx-ny]+g[idx+1]+g[idx-1]
                      -4*g[idx])+2*g[idx]-f[idx];
  }
"""

from pycuda.compiler import SourceModule
mod = SourceModule(kernels)
update_src = mod.get_function("update_src")
update = mod.get_function("update")
```

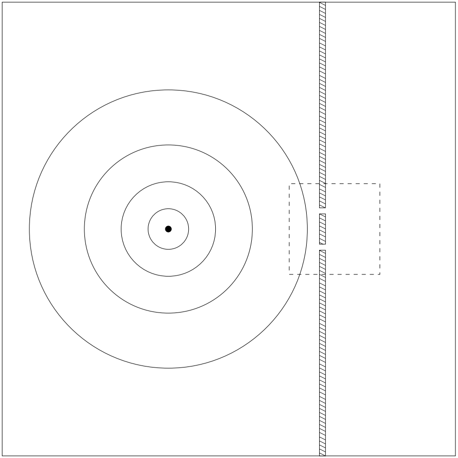
PyCUDA (Cont.)

```
import pycuda.driver as cuda
import pycuda.autoinit

c = np.ones((nx,ny),'f')*0.25
f = np.zeros_like(c)
c_gpu = cuda.to_device(c)
f_gpu = cuda.to_device(f)
g_gpu = cuda.to_device(f)
...
cuda.memcpy_htod(c_gpu,c)

Db, Dg = (256,1,1), (nx*ny/256+1,1)
nnx, nny = np.int32(nx), np.int32(ny)
src_pt = np.int32(nx/2*ny + ny/2)
...
for tn in xrange(10000):
    update_src(src_pt,np.int32(tn),f_gpu,(1,1,1),(1,1))
    update(nnx,nny,c_gpu,f_gpu,g_gpu,Db,Dg)
    update(nnx,nny,c_gpu,g_gpu,f_gpu,Db,Dg)
    if tn%100==0: cuda.memcpy_dtoh(f,f_gpu)
```

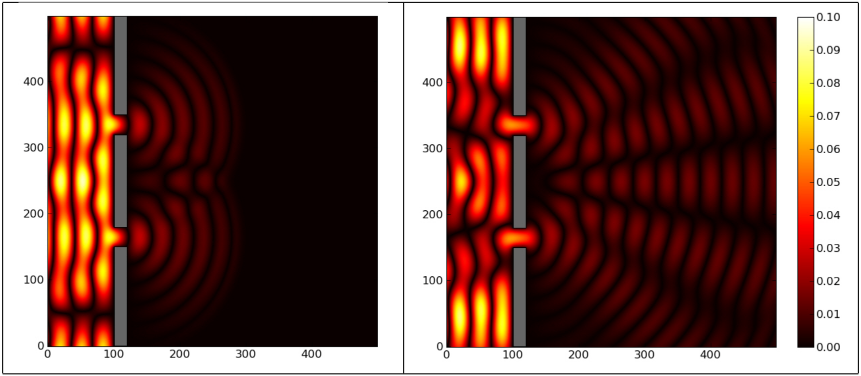
Example - Double Slit Interference



Double Slit Interference (source point)



Double Slit Interference (result)



Outline

Shared Memory Optimize

```
__global__ void update(int nx, int ny, float *c, float *f,
    float *g) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    int i = idx/ny, j = idx%ny;

    if(i>0 && j>0 && i<nx-1 && j<ny-1)
        f[idx] = c[idx]*(g[idx+ny]+g[idx-ny]
            +g[idx+1]+g[idx-1]-4*g[idx])+2*g[idx]-f[idx];
}
```

Duplicated arrays in a thread block
=> transfer to shared memory, and reuse
=> reduce global memory access
Target: g[idx+1], g[idx-1], g[idx], g[idx]

Shared Memory Optimize (Cont.)

```
__global__ void update(int nx, int ny, float *c, float *f,
    float *g) {
    int tx = threadIdx.x;
    int idx = blockIdx.x*blockDim.x + tx;

    extern __shared__ float gs[];
    gs[tx+1] = g[idx];
    int i = idx/ny, j = idx%ny;
    if(j>0 && j<ny-1) {
        if(tx==0) gs[tx] = g[idx-1];
        if(tx==blockDim.x-1) gs[tx+2] = g[idx+1];
    }
    __syncthreads()

    if(i>0 && j>0 && i<nx-1 && j<ny-1)
        f[idx] = c[idx]*(g[idx+ny]+g[idx-ny]
            +gs[tx+2]+gs[tx]-4*gs[tx+1])+2*gs[tx+1]-f[idx];
}
```

```
update(nnx,nny,c_gpu,f_gpu,g_gpu,Db,Dg,shared=(256+2)*4)
update(nnx,nny,c_gpu,g_gpu,f_gpu,Db,Dg,shared=(256+2)*4)
```

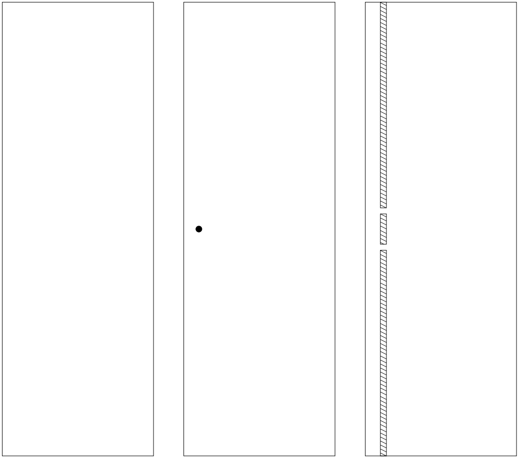
Shared Memory Optimize (Cont.)

```
__global__ void update(int nx, int ny, float *c, float *f,
    float *g) {
    int tx = threadIdx.x;
    int idx = blockIdx.x*blockDim.x + tx;

    extern __shared__ float gs[];
    gs[tx+1] = g[idx];
    int i = idx/ny, j = idx%ny;
    if(j>0 && j<ny-1) {
        if(tx==0) gs[tx] = g[idx-1];
        if(tx==blockDim.x-1) gs[tx+2] = g[idx+1];
    }
    __syncthreads()

    if(i>0 && j>0 && i<nx-1 && j<ny-1)
        f[idx] = c[idx]*(g[idx+ny]+g[idx-ny]
            +gs[tx+2]+gs[tx]-4*gs[tx+1])+2*gs[tx+1]-f[idx];
}
```

```
update(nnx,nny,c_gpu,f_gpu,g_gpu,Db,Dg,shared=(256+2)*4)
update(nnx,nny,c_gpu,g_gpu,f_gpu,Db,Dg,shared=(256+2)*4)
```

Outline

Divide Domain



Move to MPI

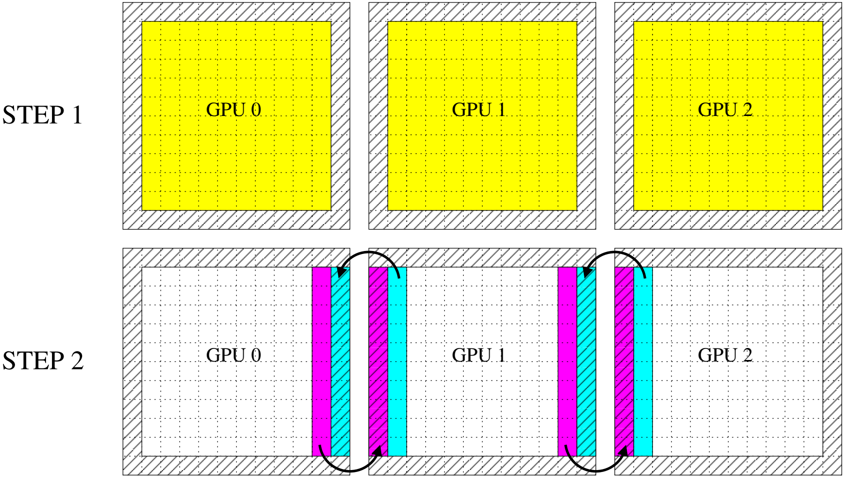
```
import pycuda.driver as cuda
import pycuda.autoinit

c = sc.ones((nx,ny),'f')*0.25
f = sc.zeros_like(c)
c_gpu = cuda.to_device(c)
f_gpu = cuda.to_device(f)
g_gpu = cuda.to_device(f)

# set the c array with geometry
cuda.memcpy_htod(c_gpu,c)

Db, Dg = (256,1,1), (nx*ny/256+1,1)
nnx, nny = sc.int32(nx), sc.int32(ny)
src_pt = sc.int32(nx/2*ny + ny/2)

# SourceModule

for tn in xrange(10000):
    update_src(src_pt,...)
    update(...,f_gpu,g_gpu,...)
    update(...,g_gpu,f_gpu,...)

    if tn%100==0:
        cuda.memcpy_dtoh(f,f_gpu)
```

```
import pycuda.driver as cuda
import boostmpi as mpi

cuda.init()
dev = cuda.Device(mpi.rank)
ctx = dev.make_context()

# memory allocate

if mpi.rank == 2:
    # set the c array with geometry
    cuda.memcpy_htod(c_gpu,c)

Db, Dg = ...
nnx, nny = ...
if mpi.rank == 1:
    src_pt = ...

# SourceModule

for tn in xrange(10000):
    if mpi.rank == 1:
        update_src(src_pt,...)
    update(...,f_gpu,g_gpu,...)
    update(...,g_gpu,f_gpu,...)

ctx.pop()
```

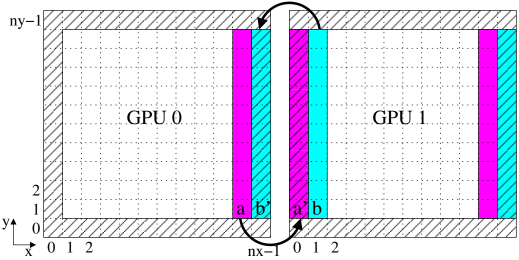
Gather data

```
nxx, nyy = nx-2, ny-2

if mpi.rank == 0:
    output = sc.zeros(3*nxx, nyy, 'f')

for tn in xrange(1000):
    ...
    if tn > 1000:
        if mpi.rank == 0:
            cuda.memcpy_dtoh(f, f_gpu)
            output[:nxx,:] = f[1:-1,1:-1]
            output[nxx:2*nxx,:] = mpi.world.recv(1,3)
            output[2*nxx:3*nxx,:] = mpi.world.recv(2,3)
        else:
            cuda.memcpy_dtoh(f, f_gpu)
            mpi.world.send(0, 3, f[1:-1,1:-1])
```

Exchange Boundaries



Memcpy Offset



offset

a: int(a_gpu) + ((nx-2)*ny+1)*sof

a': int(a_gpu) + 1*sof

b: int(a_gpu) + (ny+1)*sof

b': int(a_gpu) + ((nx-1)*ny+1)*sof

Exchange Boundaries

```
nbof = np.nbytes['float32']
dtof = np.dtype('float32')

def send(rank, tag_mark, nx, ny, a_gpu):
    if mpi.rank < rank:
        offset = int(a_gpu)+((nx-2)*ny+1)*nbof
    elif mpi.rank > rank:
        offset = int(a_gpu)+(ny+1)*nbof
    mpi.world.send(rank, tag_mark, \
            cuda.from_device(offset, (ny-2,), dtof) )

def recv(rank, tag_mark, nx, ny, a_gpu):
    if mpi.rank > rank:
        offset = int(a_gpu)+1*nbof
    elif mpi.rank < rank:
        offset = int(a_gpu)+((nx-1)*ny+1)*nbof
    cuda.memcpy_htod(offset, mpi.world.recv(rank, tag_mark))
```

Exchange Boundaries (Cont.)

```
def exchange(nx, ny, a_gpu):
    if mpi.rank == 0:
        send(1, 0, nx, ny, a_gpu)
        recv(1, 0, nx, ny, a_gpu)
    if mpi.rank == 1:
        recv(0, 0, nx, ny, a_gpu)
        send(0, 1, nx, ny, a_gpu)
        send(2, 0, nx, ny, a_gpu)
        recv(2, 1, nx, ny, a_gpu)
    if mpi.rank == 2:
        recv(1, 0, nx, ny, a_gpu)
        send(1, 1, nx, ny, a_gpu)
```

```
for tn in xrange(10000):
    if mpi.rank == 1:
        update_src(src_pt, ...)
    update(..., f_gpu, g_gpu, ...)
    exchange(nx, ny, f_gpu)
    update(..., g_gpu, f_gpu, ...)
    exchange(nx, ny, g_gpu)
```

Have your Fun with Python!

THANK YOU