HPC with Python

International HPC Summer School 2026

Ramses van Zon — Perth, July 15

In this morning's session...

SciNet

Digital Research Alliance of Canada

1. Software setup
2. Performance and Python
3. Speeding up Python
4. Parallel computing on CPUs with Python
5. GPU computing with Python
6. Conclusions

1. Software setup

Software: Setting up for this session

To get set up so you can follow along, perform the following steps.

- Login to bridges:

```
$ ssh USERNAME@bridges2.psc.edu
```

- Copy code for this session:

```
$ cp -r /ocean/projects/tra210016p/rzon/hpcpy $HOME
```

- Request interactive resources:

```
$ cd $HOME/hpcpy
$ ./interactive4.sh
```

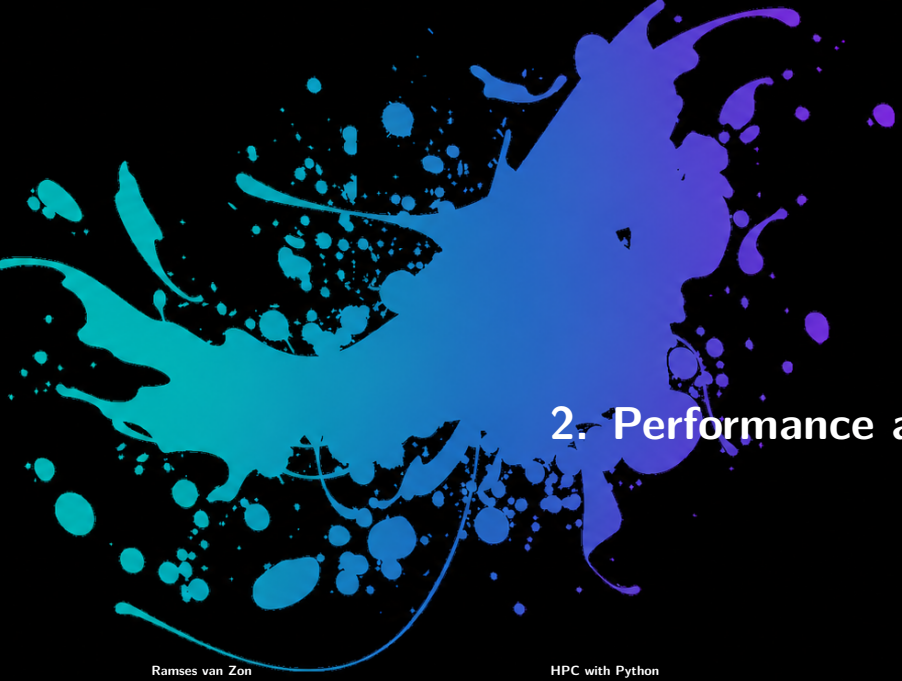
- Setup the environment:

```
$ source $HOME/hpcpy/engage
```

(do this any time you log back in)

This `engage` sources the usual virtualenv's `activate`, but also loads some software modules and sets some environment variables.

The environment is active if the prompt shows `(hpcpy314)`.



2. Performance and Python

Performance and Python

- Python is a high-level, interpreted language.
- Those defining features are often at odds with “high performance”.
- Python is fairly easy to learn, very expressive, and, not surprisingly, very popular.
- Development in Python can be substantially easier (and thus faster) than when using compiled languages.

Example: 2D thermal diffusion

Suppose we are interested in the time evolution of the two-dimensional thermal diffusion equation:

$$\frac{\partial u(x,y,t)}{\partial t} = D\left(\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2}\right)$$

on domain $[x_1, x_2] \otimes [x_1, x_2]$,

with $u(x,y,t) = 0$ at all times for all points on the domain boundary,

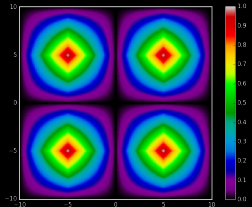
with some given initial condition $u(x,y,t) = u_0(x,y)$.

Here:

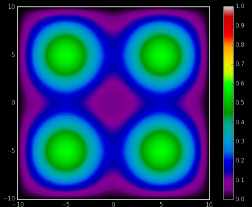
- u: temperature
- x, y: spatial coordinates
- t: time
- D: thermal diffusion constant

Example: 2D thermal diffusion, result

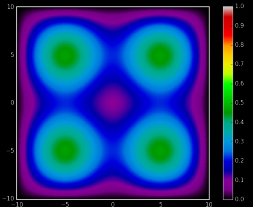
$x_1 = -10$, $x_2 = 10$, $D = 1$, four-peak initial condition.



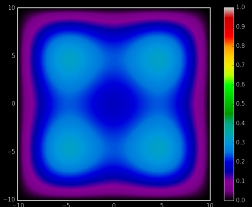
t=0



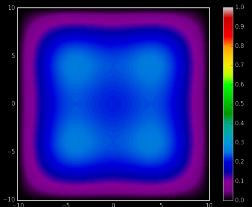
t=1



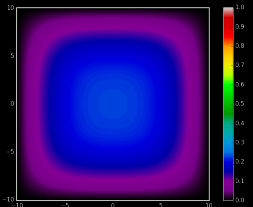
t=2



t=4

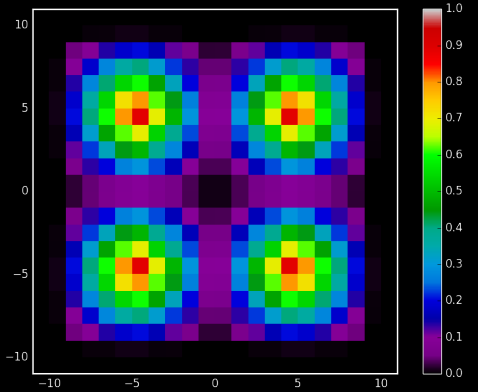


t=6



t=10

Example: 2D thermal diffusion, the algorithm



- Discretize space in both directions (points dx apart)
- Replace derivatives with finite differences.
- Explicit finite time stepping scheme (time step set by dx)
- For graphics: Matplotlib for Python, pgplot for C++/Fortran, every `outtime` time units (but switched off)

Parameters in file params.py
(also used by C++ and Fortran versions).

```
D        =   1.0;
x1       = -10.0;
x2       =  10.0;
runtime  =  10.0;
dx       =  0.05;
outtime  =  0.5;
plot     =  False;
```

Example: 2D thermal diffusion, performance

SciNet Digital Research Alliance of Canada

`diff2d.cpp`, `diff2d.f90` and `diff2d.py` contain the same code in C++, Fortran, and Python.

```
$ time make diff2d_cpp.ex diff2d_f90.ex

g++ -c -O3 -march=native -o diff2d_cpp.o diff2d.cpp
g++ -c -O3 -march=native -o diff2dplot_cpp.o diff2dplot.cpp
g++  -o diff2d_cpp.ex diff2d_cpp.o diff2dplot_cpp.o -lcpgplot -lpgplot -lX11 -lxcb -ldl -lXau -lgfortran -lpng
gfortran -c -O3 -march=native -o pgplot90.o pgplot90.f90
gfortran -c -O3 -march=native -o diff2dplot_f90.o diff2dplot.f90
gfortran -c -O3 -march=native -o diff2d_f90.o diff2d.f90
gfortran  -o diff2d_f90.ex diff2d_f90.o diff2dplot_f90.o pgplot90.o -lcpgplot -lpgplot -lX11 -lxcb -ldl -lXau -lgfortran

Elapsed: 3.21  seconds
```

```
$ time ./diff2d_cpp.ex

Elapsed: 1.88 seconds
```

```
$ time ./diff2d_f90.ex

Elapsed: 1.74 seconds
```

```
$ time python diff2d.py

Elapsed: 1137.4
```

The Python version is **600× slower** than the compiled versions!

This doesn't look promising for HPC with Python!

Then why do we bother with Python?

SciNet

Digital Research Alliance of Canada

```
from diff2dplot import plottemp
from params import D,x1,x2,runtime,dx,outtime,plot
nrows  = int((x2-x1)/d
ncols  = nrows
npnts  = nrows + 2
dx     = (x2-x1)/nrows
dt     = 0.25*dx**2/D
nsteps = int(runtime/dt)
nper   = int(outtime/dt)
if nper==0: nper = 1
x=[x1+((i-1)*(x2-x1))/nrows for i in range(npnts)]
u1     = [[0.]*npnts for i in range(npnts)]
u2     = [[0.]*npnts for i in range(npnts)]
simtime= 0*dt
for i in range(1,npnts-1):
 a = 1 - abs(1 - 4*abs((x[i]-(x1+x2)/2)/(x2-x1)))
 for j in range(1,npnts-1):
  b = 1 - abs(1 - 4*abs((x[j]-(x1+x2)/2)/(x2-x1)))
  u1[i][j] = a*b
print(simtime)
if plot: plottemp(u1,x[0],x[-1],first=True)
laplacian = [[0.]*npnts for i in range(npnts)]
```

```
for s in range(nsteps):
 for i in range(1,nrows+1):
  for j in range(1,ncols+1):
   laplacian[i][j] = ( u1[i+1][j]+u1[i-1][j]
                      +u1[i][j+1]+u1[i][j-1]
                      -4*u1[i][j] )
 for i in range(1,nrows+1):
  for j in range(1,ncols+1):
   u2[i][j]=u1[i][j]+(D/dx**2)*dt*laplacian[i][j]
 u1, u2 = u2, u1
 simtime += dt
 if (s+1)%nper == 0:
  print(simtime)
  if plot: plottemp(u1,x[0],x[-1])
```

```
def plottemp(u,x1,x2,first=False):
 import os
 import matplotlib.pyplot as plt
 if first: plt.clf(); plt.ion()
 plt.imshow(u,interpolation='none',aspect='equal',
  extent=(x1,x2,x1,x2),vmin=0.,vmax=1.,cmap='nipy_spectral')
 if first: plt.colorbar()
 plt.show();plt.pause(.1)
```

Then why do we bother with Python?

SciNet Digital Research Alliance of Canada

Fast development

- Python lends itself easily to writing clear, concise code.
- Python is very flexible: large set of very useful packages.
- Easy of use $\rightarrow$ shorter development time

Performance hit depends on application

- Python's performance hit is most prominent on 'tightly coupled' calculation on fundamental data types that are known to the CPU (integers, doubles), which is exactly the case for the 2d diffusion.
- It does much less worse on file I/O, text comparisons, etc.
- Hooks to compiled libraries to remove worst performance pitfalls.
- Some Python packages compile computations on the fly.

Profiling = measuring performance

- Performance is about maximizing the utility of **a** resource.
- This could be CPU processing power, memory, network, file I/O, etc.
- We will focus on **wall-clock performance** here, i.e., how long does it take.
- ***You need to measure where any bottlenecks are before your can fix them.***

Time Profiling by function

- To consider the computational performance of functions, but not of individual lines in your code, there is the package called `cProfile`.

Time Profiling by line

- To find cpu performance bottlenecks by line of code, there are packages like `line_profiler` and `scalene`.

The cProfile package

- Use `cProfile` or `profile` to know in which functions your script spends its time.
- You usually do this on a smaller but representative case.
- The code should be modular, i.e., with separate functions for different tasks, for cProfile to be useful.

Example

```
$ python -m cProfile -s cumulative diff2d.py
...
         321223 function calls (321221 primitive calls) in 575.740 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      3/1    0.000    0.000  575.740  575.740 {built-in method builtins.exec}
        1    0.005    0.005  575.740  575.740 diff2d.py:1(<module>)
        1  575.667  575.667  575.732  575.732 diff2d.py:14(main)
   320800    0.062    0.000    0.062    0.000 {built-in method builtins.abs}
        2    0.000    0.000    0.004    0.002 <frozen importlib._bootstrap>:1360(_find_and_load)
        2    0.000    0.000    0.003    0.002 <frozen importlib._bootstrap>:1308(_find_and_load_unlocked)
        2    0.000    0.000    0.002    0.001 <frozen importlib._bootstrap>:914(_load_unlocked)
```

The line_profiler package

- Use `line_profiler` to know, line-by-line, where your script spends its time.
- You usually do this on a smaller but representative case.
- First thing to do is to have your code in a function.
- You also need to modify your script slightly, i.e., decorate your function(s) with `@profile`.
- Run your script on the command line with

```
$ kernprof -l -v SCRIPTNAME
```

line_profiler script instrumentation

Script before:

```
x=[1.0]*(2048*2048)
a=str(x[0])
a+="\nis a one\n"
del x
print(a)
```

Script after:

```
#file: profileme.py
@profile
def profilewrapper():
  x=[1.0]*(2048*2048)
  a=str(x[0])
  a+="\nis a one\n"
  del x
  print(a)
profilewrapper()
```

Run at the command line:

```
$ kernprof -l -v profileme.py
```

Output of line_profiler

SciNet Digital Research Alliance of Canada

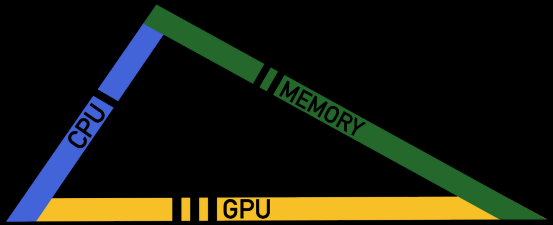
```
$ kernprof -l -v profileme.py
1.0
is a one

Wrote profile results to 'profileme.py.lprof'
Timer unit: 1e-06 s

Total time: 0.0302836 s
File: profileme.py
Function: profilewrapper at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def profilewrapper():
     4         1      14878.9  14878.9     49.1    x=[1.0]*(2048*2048)
     5         1         14.0     14.0      0.0    a=str(x[0])
     6         1          3.9      3.9      0.0    a+="\nis a one\n"
     7         1      15264.0  15264.0     50.4    del x
     8         1        122.8    122.8      0.4    print(a)
```

The Scalene package



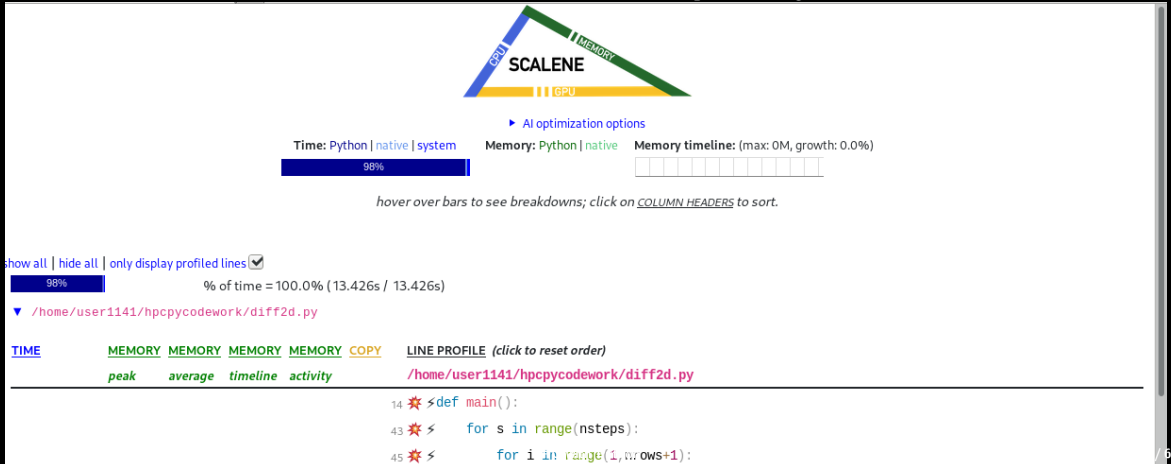
- Python Profiler for CPU, memory, and GPU
- Fast
- Accurate
- Distinguishes Python from native (i.e. C) code and system calls.
- No decorator required
  (remove them if you still had those!)

Scalene usage

```
$ scalene run -o diff2d-profile.json diff2d.py
```

This saves profile data to a file called “diff2d-profile.json”.

To view the result to in a browser, use: `$ scalene view diff2d-profile.json`.

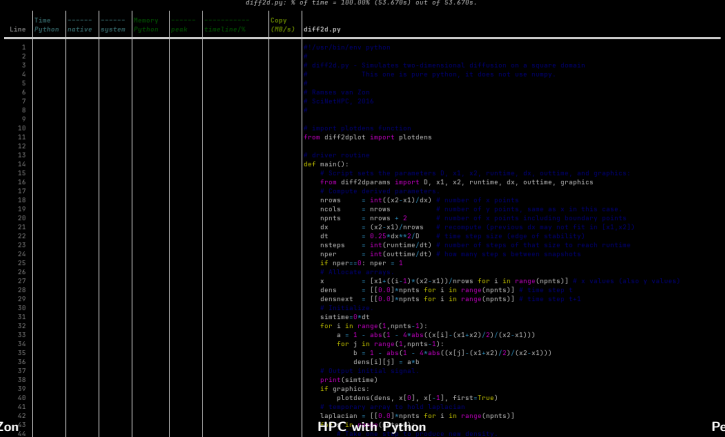


Scalene usage on the command line

The ‘hpcpy314’ environment contains a browser called ‘Pale Moon’, but this is very slow over the internet.

You can also tell scalene not to generate the HTML and open a browser, but, instead, to report the results to the command line instead:

```
$ scalene view --cli diff2d-profile.json
```



Let's profile diff2d.py

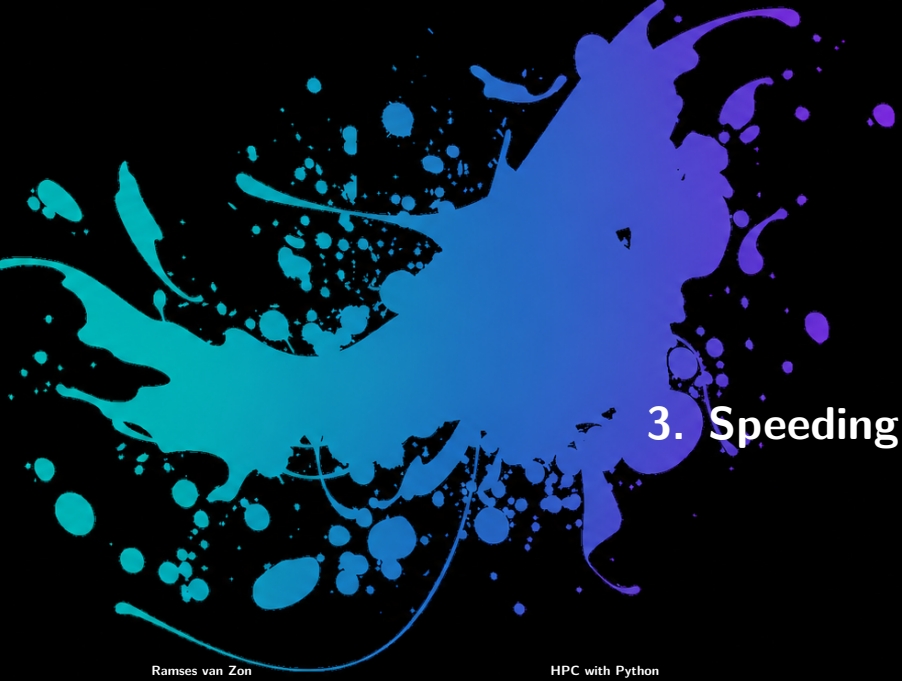
We make the resolution coarser by increasing the 'pixel size' from dx=0.05 to dx=0.5 in `params.py`.

part 1 / 2

- Run diff2d through `line_profiler`.
- Identify the two or three most costly lines in the python code.

part 2 / 2

- Try the same with `scalene`.
- Are the same lines identified?



3. Speeding up Python

Lists aren't the ideal data type for numerics

Python lists can do funny things that you don't expect, if you're not careful.

- Lists are just a collection of items, of any type.
- If you do mathematical operations on a list, you won't get what you expect.
- These are not the ideal data type for scientific computing.
- Arrays are a much better choice, but are not a native Python data type.

```
>>> a = [1,2,3,4]
```

```
>>> a
[1, 2, 3, 4]
```

```
>>> b = [3,5,5,6]
```

```
>>> b
[3, 5, 5, 6]
```

```
>>> 2*a
[1, 2, 3, 4, 1, 2, 3, 4]
```

```
>>> a+b
[1, 2, 3, 4, 3, 5, 5, 6]
```

The NumPy package

- Almost everything that you want to do starts with NumPy.
- Contains arrays of various types and forms: zeros, ones, linspace, etc.

```
>>> from numpy import zeros, ones
```

```
>>> zeros(5)
array([0., 0., 0., 0., 0.])
```

```
>>> ones(5, dtype=int)
array([1, 1, 1, 1, 1])
```

```
>>> zeros([2,2])
array([[0., 0.],
       [0., 0.]])
```

```
>>> from numpy import arange
```

```
>>> arange(5)
array([0, 1, 2, 3, 4])
```

```
>>> from numpy import linspace
```

```
>>> linspace(1,5)
array([1.        , 1.08163265, 1.16326531, 1.24489796, 1.32653061,
       1.40816327, 1.48979592, 1.57142857, 1.65306122, 1.73469388,
       1.81632653, 1.89795918, 1.97959184, 2.06122449, 2.14285714,
       2.2244898 , 2.30612245, 2.3877551 , 2.46938776, 2.55102041,
       2.63265306, 2.71428571, 2.79591837, 2.87755102, 2.95918367,
       3.04081633, 3.12244898, 3.20408163, 3.28571429, 3.36734694,
       3.44897959, 3.53061224, 3.6122449 , 3.69387755, 3.7755102 ,
       3.85714286, 3.93877551, 4.02040816, 4.10204082, 4.18367347,
       4.26530612, 4.34693878, 4.42857143, 4.51020408, 4.59183673,
       4.67346939, 4.75510204, 4.83673469, 4.91836735, 5.        ])
```

```
>>> linspace(1,5,6)
array([1. , 1.8, 2.6, 3.4, 4.2, 5. ])
```

Element-wise arithmetic

vector-vector & vector-scalar multiplication

1-D arrays are often called 'vectors'.

- When vectors are multiplied with $*$, you get element-by-element multiplication.
- When vectors are multiplied by a scalar (a 0-D array), you also get element-by-element multiplication.
- To get an inner product, use `@`.
  (Or use the 'dot' method in Python < 3.5)

```
>>> import numpy as np
```

```
>>> a = np.arange(4)
>>> b = np.arange(3., 7.)
>>> c = 2
```

```
>>> a, b, c
(array([0, 1, 2, 3]), array([3., 4., 5., 6.]), 2)
```

```
>>> a * b
array([ 0.,  4., 10., 18.])
```

```
>>> a * c
array([0, 2, 4, 6])
```

```
>>> b * c
array([ 6.,  8., 10., 12.])
```

```
>>> a @ b
32.0
```

Does changing to NumPy arrays help?

Let's return to our 2D diffusion example.

Note: Restore the original diff2dparam.py (dx=0.05)!

Pure Python implementation:

```
$ time python diff2d.py
Elapsed: 1137.4 seconds
```

NumPy implementation:

```
$ time python diff2d_slow_numpy.py
Elapsed: 6159.0 seconds
```

Hmm, not really.

Really not! More than 5x slower!

So what gives?

Let's inspect the code

```
#diff2d_slow_numpy.py
from diff2dplot import plottemp
from params import D,x1,x2,runtime,dx,outtime,plot
import numpy as np
nrows  = int((x2-x1)/d
ncols  = nrows
npnts  = nrows + 2
dx     = (x2-x1)/nrows
dt     = 0.25*dx**2/D
nsteps = int(runtime/dt)
nper   = int(outtime/dt)
if nper==0: nper = 1
x        = np.linspace(x1-dx,x2+dx,num=npnts)
u1       = np.zeros((npnts,npnts))
u2       = np.zeros((npnts,npnts))
simtime  = 0*dt
for i in range(1,npnts-1):
 a = 1 - abs(1 - 4*abs((x[i]-(x1+x2)/2)/(x2-x1)))
 for j in range(1,npnts-1):
  b = 1 - abs(1 - 4*abs((x[j]-(x1+x2)/2)/(x2-x1)))
  u1[i][j] = a*b
print(simtime)
if plot: plottemp(u1,x[0],x[-1],first=True)
```

Look at all those loops and indices!

Look at all those loops and indices!

```
laplacian = np.zeros((npnts,npnts))
for s in range(nsteps):
 for i in range(1,nrows+1):
  for j in range(1,ncols+1):
   laplacian[i][j] = (u1[i+1][j]+u1[i-1][j]
                     +u1[i][j+1]+u1[i][j-1]
                     -4*u1[i][j])
 for i in range(1,nrows+1):
  for j in range(1,ncols+1):
   u2[i][j]=u1[i][j]+(D/dx**2)*dt*laplacian[i][j]
 u2, u1 = u1, u2
 simtime += dt
 if (s+1)%nper == 0:
  print(simtime)
  if plot: plottemp(u1,x[0],x[-1])
```

"Why does that matter?" you ask?

Python overhead

- Python's overhead comes mainly from its interpreted and dynamic nature.
- The `diff2d_slow_numpy.py` code uses NumPy arrays, but still has loops over indices.
- In each iteration, Python code has to be interpreted and integer manipulations have to be performed, regardless of whether you're using NumPy arrays.
- NumPy will not give much speedup until you use its element-wise **'vectorized'** operations.

How to write vectorized Python code

Instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in range(100):
  c[i] = a[i] + b[i]
```

You would use slices:

```
a = np.linspace(0.0,1.0,100)
b = np.linspace(1.0,2.0,100)
c = a + b
```

And to deal with shifts, instead of

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = np.ndarray(100)
for i in range(100):
  c[i] = a[i] + b[i+1]
```

You would write:

```
a = np.linspace(0.0,1.0,101)
b = np.linspace(1.0,2.0,101)
c = a[0:100] + b[1:101]
```

Vectorization results in:

- shorter Python code
- less repeatedly interpreted lines
- calls to C or Fortran functions by NumPy.

Does changing to NumPy really help?

Pure Python implementation:

```
$ time python diff2d.py
Elapsed: 1137.4 seconds
```

NumPy vectorized implementation:

```
$ time python diff2d_numpy.py
Elapsed: 8.40 seconds
```

Yeah! **~130 × speed-up**

How? Numpy-vectorized implementation:

SciNet Digital Research Alliance of Canada

`$ diff diff2d_slow_numpy.py diff2d_numpy.py`

```
u1 = np.zeros((npnts,npnts)) # time step t
for i in range(1,npnts-1):
  a = 1 - abs(1 - 4*abs((x[i]-(x1+x2)/2)/(x2-x1)))
  for j in range(1,npnts-1):
    b = 1 - abs(1 - 4*abs((x[j]-(x1+x2)/2)/(x2-x1)))
    u1[i][j] = a*b
```

```
a = 1 - abs(1 - 4*abs((x-(x1+x2)/2)/(x2-x1)))
u1 = np.outer(a,a)
```

```
for i in range(1,nrows+1):
 for j in range(1,ncols+1):
  laplacian[i,j] = (u1[i+1,j]+u1[i-1,j]+u1[i,j+1]
                    +u1[i,j-1]-4*u1[i,j])
```

```
laplacian[1:nrows+1,1:ncols+1]=(
  u1[2:nrows+2,1:ncols+1]+u1[0:nrows+0,1:ncols+1]
 +u1[1:nrows+1,2:ncols+2]+u1[1:nrows+1,0:ncols+0]
 -4*u1[1:nrows+1,1:ncols+1])
```

```
for i in range(1,nrows+1):
 for j in range(1,ncols+1):
  u2[i,j] = u1[i,j] + (D/dx**2)*dt*laplacian[i,j]
```

```
u2[:,:] = u1 + (D/dx**2)*dt*laplacian
```

Reality check: NumPy vs. compiled code

SciNet

Digital Research Alliance of Canada

NumPy vectorized implementation:

```
$ time python diff2d_numpy.py
Elapsed: 8.40 seconds
```

Compiled versions:

```
$ time ./diff2d_cpp.ex
Elapsed: 1.88 seconds
```

```
$ time ./diff2d_f90.ex
Elapsed: 1.74 seconds
```

Typically, Python+NumPy is still ~5× slower than compiled.

What about Cython?

- Cython is a compiler for Python code.
- Almost all Python is valid Cython.
- Typically used for packages, to be used in regular Python scripts.

Let's look at the timing first:

```
$ make -f Makefile_cython
...
python diff2dnumpylibsetup.py build_ext --inplace
```

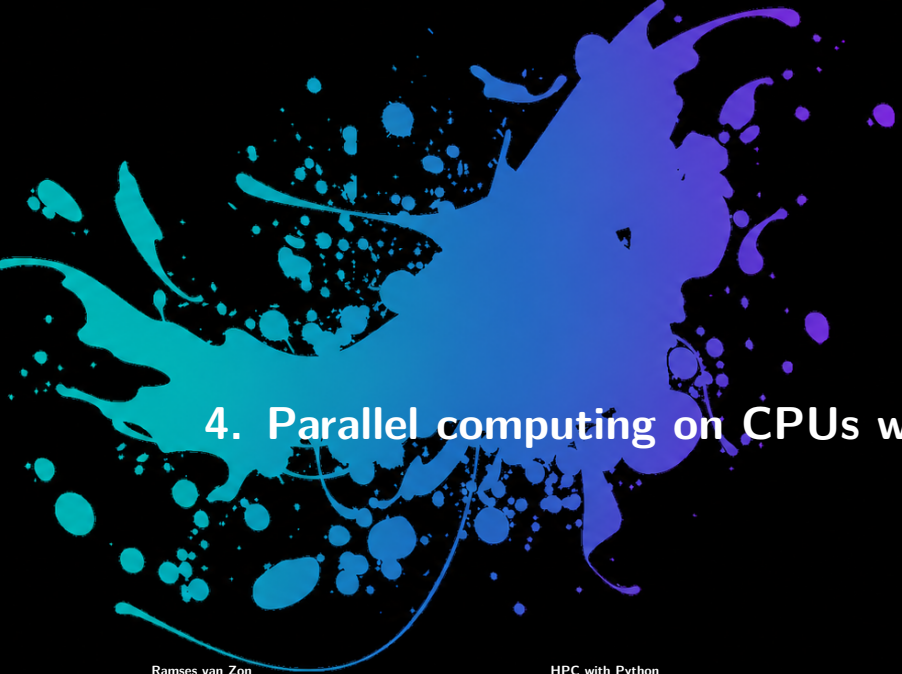
```
$ time python diff2d_numpy_cython.py
Elapsed: 8.40 seconds
```

```
$ time python diff2d_numpy_cython.py
Elapsed: 9.08 seconds
```

Not faster?!

Not faster because it's still Python!

- The compilation preserves the pythonic nature of the language, i.e, garbage collection, range checking, reference counting, etc, are still done: *no performance enhancement.*
- If you want to get around that, you need to use Cython specific extensions that use C types.
- That would be a whole session in and of itself.



4. Parallel computing on CPUs with Python

Parallel Python packages

SciNet Digital Research Alliance of Canada

We will look at a number of approaches to parallel CPU programming with Python:

Package	Functionality
a) NumExpr	threaded parallelization of certain NumPy expressions
b) Numba	just-in-time compiler for Python functions
c) MPI4py	message passing between processes

Note: Threads in Python exist but do not run simultaneously because of the Global Interpreted Lock (GIL). The GIL can be removed in Python 3.14, but this requires a comletely separate python installation, and very few Python packages support this out-of-the-box yet.

4a. The numexpr package

The numexpr package is useful if you're doing array algebra:

- It is essentially a just-in-time compiler for NumPy.
- It takes matrix expressions, breaks things up into threads, and does the calculation in parallel.
- Somewhat awkwardly, it takes its input in as a string.
- In some situations using numexpr can significantly speed up your calculations.
- This is a bit like "OpenMP-ing a loop" in C/C++/Fortran.

Numexpr in a nutshell

- Give it an array arithmetic expression, and it will compile and run it, and return or store the output.
- Supported operators:

  `+ - * / % << >> < <= == != >= > & | ~ **`
- Supported functions:
  `where, sin, cos, tan, arcsin, arccos arctan, arctan2, sinh, cosh, tanh, arcsinh, arccosh arctanh, log, log10, log1p, exp, expm1, sqrt, abs, conj, real, imag, complex, contains`

Using the numexpr package

```
from time import time
import numpy as np
a = np.random.rand(250000000)
b = np.random.rand(250000000)
c = np.zeros(250000000)
```

Without numexpr:

```
t = time()
c = a**2 + b**2 + 2*a*b
print("Elapsed: %f seconds" % (time()-t))
```

```
Elapsed: 2.352491 seconds
```

With numexpr:

```
import numexpr as ne
ne.set_num_threads(1)
t = time()
c = ne.evaluate('a**2 + b**2 + 2*a*b')
print("Elapsed: %f seconds" % (time()-t))
```

```
Elapsed: 1.331458 seconds
```

```
ne.set_num_threads(2)
t = time()
c = ne.evaluate('a**2 + b**2 + 2*a*b')
print("Elapsed: %f seconds" % (time()-t))
```

```
Elapsed: 1.231132 seconds
```

```
ne.set_num_threads(3)
t = time()
c = ne.evaluate('a**2 + b**2 + 2*a*b')
print("Elapsed: %f seconds" % (time()-t))
```

```
Elapsed: 0.582886 seconds
```

Another way to set the number of threads used by numexpr:

```
$ export NUMEXPR_NUM_THREADS=3
```

before starting Python or a Python script.

Numexpr for diff2d

- Numexpr has no facilities for slicing or offsets, etc.
- And it wants contiguous data.
- This poses an issue for our diffusion code, in which we have to do something like:

```
laplacian[1:nrows+1,1:ncols+1] = (u1[2:nrows+2,1:ncols+1] +
                                  u1[0:nrows+0,1:ncols+1] +
                                  u1[1:nrows+1,2:ncols+2] +
                                  u1[1:nrows+1,0:ncols+0] -
                                  4*u1[1:nrows+1,1:ncols+1])
```

- We would need to make a copy of `u1[2:nrows+2,1:ncols+1]` etc. into a new NumPy array before we can use `numexpr`, but such memory copies are expensive.
- We want `numexpr` to use the same data as in u, but *viewed* differently

Numexpr for diff2d (continued)

- We want `numexpr` to use the same data as in u, but *viewed* differently.
- That is tricky, and requires knowledge of the data's memory structure.
- `diff2d_numexpr.py` shows one possible solution.

```
$ time python diff2d_numpy.py
Elapsed: 8.4 seconds
```

```
$ export NUMEXPR_NUM_THREADS=4
$ time python diff2d_numexpr.py
Elapsed: 7.0 seconds
```

- Nice, some speed up.
  (You would better speed-up if you increase the grid, i.e., decrease dx).

Numexpr for diff2d (details)

To get the diffusion algorithm in a form that has no slices or offsets, we need to linearize the 2d arrays into 1d arrays, but in a way that avoids copying the data.

This is how this is achieved in `diff2d_numexpr`:

```
u1 = u1.ravel()
u2 = u2.ravel()
uL = u1[npnts-1:-npnts-1] # same data one cell left
uR = u1[npnts+1:-npnts+1] # same data one cell right
uU = u1[0:-2*npnts]       # same data one cell up
uD = u1[2*npnts:]         # same data one cell down
uC = u1[npnts:-npnts]
ne.evaluate('uC + (D/dx**2)*dt*(uL+uR+uU+uD-4*uC)',
            out=u2[npnts:-npnts])
u1 = u1.reshape((npnts,npnts))
u2 = u2.reshape((npnts,npnts))
```

4b. The Numba package

- Numba allows compilation of selected portions of Python code to native code.
- Decorator based: compile a function.
- It can use multi-dimensional arrays and slices, like NumPy.
- Very convenient.
- Numba can use GPUs, but you're programming them like CUDA kernels.

Numba for the thermal diffusion code

For the diffusion code, we change the time step into a function with a decorator:

Before:

```
 # Take one step to produce new temperature.
 laplacian[1:nrows+1,1:ncols+1]=u1[2:nrows+2,1:ncols+1]+u1[0:nrows+0,1:ncols+1]+u1[1:nrows+1,2:ncols+2]+u1[1:nrows+1,0:n
 u2[:,:] = u1 + (D/dx**2)*dt*laplacian
```

```
$ time python diff2d_numpy.py
Elapsed: 8.40 seconds
```

After:

```
from numba import njit
@njit
def timestep(laplacian,u1,u2,nrows,ncols,D,dx,dt):
  laplacian[1:nrows+1,1:ncols+1]=u1[2:nrows+2,1:ncols+1]+u1[0:nrows+0,1:ncols+1]+u1[1:nrows+1,2:ncols+2]+u1[1:nrows+1,0:
  u2[:,:] = u1 + (D/dx**2)*dt*laplacian
...
  timestep(laplacian,u1,u2,nrows,ncols,D,dx,dt)
```

```
$ time python diff2d_numba.py
Elapsed: 31.98 seconds
```

Why the poor performance of Numba?

- Numba can compile more complicated code than e.g. numexpr, but this compilation takes some time.
- We already optimized the Python code by using vectorized operations.
- So the same NumPy routines are called!
- But for codes that are not (so easily) vectorized, Numba can help a lot with very little code changes.

Numba for the thermal diffusion, second try

SciNet Digital Research Alliance of Canada

Bring back the loops!

```
@njit
def timestep(laplacian,u1,u2,nrows,ncols,D,dx,dt):
   for i in range(1,nrows+1):
      for j in range(1,ncols+1):
         laplacian[i][j] = u1[i+1][j]+u1[i-1][j]+u1[i][j+1]+u1[i][j-1]-4*u1[i][j]
   for i in range(1,nrows+1):
      for j in range(1,ncols+1):
         u2[i][j] = u1[i][j]+(D/dx**2)*dt*laplacian[i][j]
```

```
$ time python diff2d_numba_loop.py >diff2d_numba_loop.out
Elapsed: 4.67 seconds
```

That's better!

We're getting close to compiled speeds!

Numba for thermal diffusion code, parallel

We can ask numba to use multiple cores too.

It can do work-sharing of loops, much like OpenMP, if you use prange instead of range.

```
from numba import njit, prange
@njit(parallel=True)
def timestep(laplacian,u1,u2,nrows,ncols,D,dx,dt):
   for i in prange(1,nrows+1):
      for j in range(1,ncols+1):
         laplacian[i][j] = u1[i+1][j]+u1[i-1][j]+u1[i][j+1]+u1[i][j-1]-4*u1[i][j]
   for i in prange(1,nrows+1):
      for j in range(1,ncols+1):
         u2[i][j] = u1[i][j]+(D/dx**2)*dt*laplacian[i][j]
```

```
$ time python diff2d_numba_par_loop.py
Elapsed: 4.20 seconds
```

Even (slightly) better!

Note: Here too, by increasing the resolution to e.g. dx=0.025, you'll get much better scaling.

4c. The MPI4py package

The previous parallel techniques could only use CPUs on a single node of a cluster.

Using more than one node requires these nodes to communicate.

MPI is a common way of doing that communication.

- MPI is a standard library interface for C and Fortran.
- Parallel tasks are processes running on separate CPUs.
- Each process/CPU has only its own, private memory.
- Processes need not be on the same compute node.
- You can scale up the size of your system to as many nodes as you have access to.

Mpi4py features

- mpi4py is a python wrapper around the MPI library
- Point-to-point communication (sends, receives)
- Collective (broadcasts, scatters, gathers) communications of any picklable Python object,
- Optimized communications of Python object exposing the single-segment buffer interface (NumPy arrays, builtin bytes/string/array objects).
- Names of functions much the same as in C/Fortran, but are methods of the communicator (object-oriented).

Mpi4py in a nutshell

- MPI communication is govered by a communicator:

```
from mpi4py import MPI # this calls MPI_Init
comm = MPI.COMM_WORLD
```

- Every process started through `mpirun` runs the same code, i.e., the full python script, at the same time.
- Every process has a rank, which is the only feature of that distinguises it from its siblings.

```
rank = comm.Get_rank()
```

- The number of processes is the communicator's size:

```
size = comm.Get_size()
```

- Processes can send values to other ranks:

```
comm.send(variable, dest=torank)
```

- Processes can receive things from other ranks:

```
variable = comm.recv(source=fromrank)
```

- Sends and receives must match or your program will hang. The combined `comm.sendrecv` can help avoid this deadlock.
- Processes can do collective actions, like summing up values:

```
result = comm.reduce(value2sum, op=MPI.SUM, root=0)
```

MPI4Py example

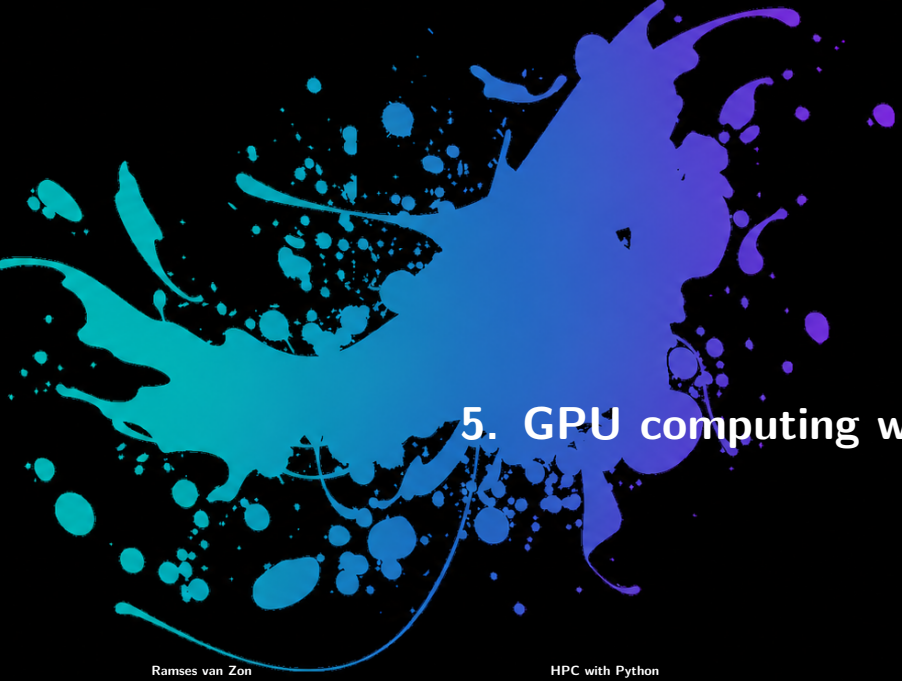
```python
# fifthmessage.py
from mpi4py import MPI

def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    tag = 1
    left = rank - 1
    if left < 0: left = size-1
    right = rank + 1;
    if right >= size: right = 0
    msgsent = rank*rank
    msgrcvd = comm.sendrecv(msgsent, right, source=left, sendtag=tag, recvtag=tag)
    print(f"{rank}: Sent {msgsent} and got {msgrcvd}")

if __name__ == '__main__':
    main()
```

How to run:

```
$ cd $HOME/hpcpy
$ source engage
$ mpirun -n 4 python fifthmessage.py
2: Sent 4 and got 1
1: Sent 1 and got 0
0: Sent 0 and got 9
3: Sent 9 and got 4
$
```



5. GPU computing with Python

Graphic Processing Units (GPUs)

- These are devices containing thousands of execution units that can run simultaneously.
- The units can apply the same operation to many data elements at the same time.
- Each execution unit by itself is less powerful than the CPU.
- Its power can only be utilized with massive data parallelism.
- GPUs (typically) are situated on a host node, and the CPU must manage program execution of so-called GPU kernels.
- GPU kernels must be compiled for the GPU specifically.
- Starting GPU kernels incurs overhead.
- GPUs have a separate memory space, so data needs to be transferred from CPU to GPU, which is overhead.

Using GPUs with Python

There are a few ways to program GPUs in Python:

- Using a package like **CuPy** that moves NumPy-like operations to the GPU.
- By using packages that allow you to write CUDA-like kernels, like **Numba**.
- By using CUDA in Python with the **cuda-python** package.

Sometimes, you can use a formalism that uses GPUs in its implementation, like:

- Tensorflow: Google's framework to build, train, and deploy deep neural networks.
- PyTorch: Python library to accelerates the development of deep learning models.
- Rapids: GPU Accelerated Data Science

6a. The CuPy package

CuPy

- CuPy aims to be a drop-in replacement for NumPy and SciPy.
- CuPy implements a subset of the NumPy API.
- CuPy implements a subset of the SciPy API.
- "Simply replace `numpy` with `cupy`."*

  * *Conditions apply.*

CuPy example

First, how to get a GPU on Bridges?

- Exit any existing interactive salloc session.
- Write a job script to run the python script using "`--reservation=IHPCSSPythonGPU -p GPU-shared --gres=gpu:v100-32:1`".
- Submit that script with sbatch.
- Or use the gpurunpy wrapper:

```
$ gpurunpy PYTHONSCRIPT
```

which creates a job script and submits it.

Note:

CuPy arrays live on the GPU.
Operations on CuPy arrays are performed on the GPU (by generated kernels).

Try submitting this python script:

```
#!/usr/bin/env python
# cupyexample.py
import cupy as cp
z = cp.zeros((2,3))
print(z)
[[0. 0. 0.]
 [0. 0. 0.]]
print(z.device)
y = z.get()
print(y.device)
```

```
$ gpurunpy cupyexample.py
```

```
Submission script: submit_cupyexample.slurm
Submitted batch job 42209149
Output file will be: output_cupyexample_42209149.txt
```

```
$ cat output_cupyexample_*.txt
[[0. 0. 0.]
 [0. 0. 0.]]
<CUDA Device 0>
cpu
elapsed 3.077 seconds
```

CuPy for thermal diffusion code

SciNet

Digital Research Alliance of Canada

```
$ diff diff2d_numpy.py diff2d_cupy.py
```

```
import numpy as np
```

```
import cupy as cp
```

```
x  = np.linspace(x1-dx,x2+dx,num=npnts)
u2 = np.zeros((npnts,npnts))
```

```
x  = cp.linspace(x1-dx,x2+dx,num=npnts)
u2 = cp.zeros((npnts,npnts))
```

```
u1 = np.outer(a,a)
```

```
u1 = cp.outer(a,a)
```

```
laplacian = np.zeros((npnts,npnts))
```

```
laplacian = cp.zeros((npnts,npnts))
```

```
plottemp(u1, x[0], x[-1])
```

```
plottemp(u1.get(), x[0].get(), x[-1].get())
```

Note: To minimize memory copying, we are keeping the data on the GPU, except for plotting.

Performance

Timeing results:

```
$ time python diff2d_numpy.py
Elapsed: 8.40 seconds
$ time python diff2d_numba_par_loop.py # 4 cores
Elapsed: 4.20 seconds
```

```
$ gpurunpy diff2d_cupy.py
$ # wait for job to finish...
$ tail -n1 output_diff2d_cupy_*.txt
Elapsed: 3.31 seconds
```

But only slightly faster than the numba parallel loop with 4 CPU cores.

Let's increase our resolution: $dx = 0.05 \longrightarrow dx = 0.02$

```
$ time python diff2d_numba_par_loop.py  # 4 cores
Elapsed: 261.07 seconds
```

```
$ gpurunpy diff2d_cupy.py
$ # wait for job to finish...
$ tail -n1 output_diff2d_cupy_*.txt
Elapsed: 3.31 seconds
Elapsed: 26.42 seconds
```

10x better!

But we could use more CPU cores and get close to the GPU speed.

Can we improve the CuPy result?

CuPy overhead

There are three main contributions to overhead cost in CuPy:

- Copying data from CPU to GPU and vice versa.
- Compiling GPU kernels (although these are saved on disk and reused).
- Launching GPU kernels.

We minimized 1 already, 2 is automatic, but have not addressed 3 yet.

Our computational load is (still) mainly in these lines:

```
laplacian[1:nrows+1,1:ncols+1] = (u1[2:nrows+2,1:ncols+1] + u1[0:nrows+0,1:ncols+1]
     + u1[1:nrows+1,2:ncols+2] + u1[1:nrows+1,0:ncols+0] - 4*u1[1:nrows+1,1:ncols+1])
 u2[:,:] = u1 + (D/dx**2)*dt*laplacian
```

Each operation here becomes a kernel to be compiled and launched. That's about 7 kernels.

CuPy minimize overhead

There are three main contributions to overhead cost in CuPy:

- Copying data from CPU to GPU and vice versa; **mitigated by keeping data on GPU**
- Compiling GPU kernels; **mitigated by caching.**
- Launching GPU kernels; **mitigated by fusing kernels with** `cp.fuse`

cp.fuse

This works though a decorator; put all code that you want to fuse into one kernel into a function and put `@cp.fuse` in it. Similar to the numba `timestep` approach:

```
@cp.fuse
def timestep(laplacian,u1,u2,nrows,ncols,D,dx,dt):
    laplacian[1:nrows+1,1:ncols+1] = u1[2:nrows+2,1:ncols+1] + u1[0:nrows+0,1:ncols+1] + u1[1:nrows+1,2:ncols+2] + u1[1:
    u2[:,:] = u1 + (D/dx**2)*dt*laplacian
```

Now without slicing

Unfortunately, `cp.fuse` does not support slices, so this won't work.

This gives flashbacks to the same situation in `numexpr`, we know what to do: define the slices as views:

```
@cp.fuse()
def timestep(uL,uR,uU,uD,uC,factor):
    return uC + factor*(uU + uD + uR + uL - 4*uC)
...
uL = u1[1:-1, 0:-2]
uR = u1[1:-1, 2:  ]
uU = u1[0:-2, 1:-1]
uD = u1[2:  , 1:-1]
uC = u1[1:-1, 1:-1]
factor = cp.array((D/dx**2)*dt)  # scalar on the GPU
u2[1:-1, 1:-1] = timestep(uC,uU,uD,uR,uL,factor)
```

Now let's do timing again:

```
$ time python diff2d_numba_par_loop.py # 4 cores
Elapsed: 261.07 seconds
```

```
$ gpurunpy diff2d_cupy_fuse.py
$ # wait for the job to finish...
$ tail -n1 output_diff2d_cupy_fuse_*.txt
Elapsed: 5.76 seconds.
```

That's more like it!

6b. The Numba package for GPUs

Numba is a NumPy optimizing library using the just-in-time (JIT) compiler LLVM.

The numba-cuda package by NVIDIA uses the low-level cupy-python bindings from CUDA Python under the hood and allows to run Numba on GPUs.

So where:

- CuPy feels like NumPy on the GPU,
- Numba on multi-core CPUs is like OpenMP in C/Fortran,

Numba on GPUs is exactly like programming CUDA kernels, but with those kernels written in Python.

CUDA programming would also be a full course in itself.

We'll suffice here by presenting the thermal diffusion example with `numba.cuda`.

Numba.cuda kernel for thermal diffusion

SciNet Digital Research Alliance of Canada

```
from numba import cuda

# kernel:
@cuda.jit
def timestepkernel(u1,u2,nrows,ncols,factor):
   i,j = cuda.grid(2)
   if i>=1 and i<=nrows and j >= 1 and j <=ncols:
      laplacian = u1[i+1][j]+u1[i-1][j]+u1[i][j+1]+u1[i][j-1]-4*u1[i][j]
      u2[i][j] = u1[i][j]+factor*laplacian

# driver:
def timestep(u1,u2,nrows,ncols,factor):
   import math
   threadsperblock = (16, 16)
   blockspergrid_x = math.ceil((nrows+2) / threadsperblock[0])
   blockspergrid_y = math.ceil((ncols+2) / threadsperblock[1])
   blockspergrid = (blockspergrid_x, blockspergrid_y)
   timestepkernel[blockspergrid,threadsperblock](u1,u2,nrows,ncols,factor)
```

Data on the device

It is important that u1 and u2 are created as device arrays:

```
hu = np.zeros((npnts,npnts)) # host copy
u1 = cuda.device_array(shape=(npnts,npnts), dtype=hu.dtype) # t
u2 = cuda.device_array(shape=(npnts,npnts), dtype=hu.dtype) # t+1
a = 1 - abs(1 - 4*abs((x-(x1+x2)/2)/(x2-x1)))
hu = np.outer(a,a)
u1.copy_to_device(hu)
```

Only at plotting do we retrieve the data:

```
if plot:
    u1.copy_to_host(ary=hu)
    plottemp(hu, x[0], x[-1])
```

Numba.cuda timing

```
$ gpurunpy diff2d_numba_gpu_kernels.py
$ # wait for job to finish
$ tail -n1 output_diff2d_numba_gpu_kernels_*.txt
Elapsed: 5.82 seconds.
```

Essentially the same speed as CuPy.

6. Conclusions

Conclusions: Performance

- Getting performance out of Python involves getting out of Python.
- Find your performance with scalene or line_profiler before optimizing.
- Numpy, when used with vectorized expressions helps.
  Then NumExpr can help even more.
- Numba, when **not** used with vectorized expressions helps.
- Many Python packages are written in C or Fortran, and just expose an interface to Python.
  (pandas, scipy, scikit-learn, matplotlib, . . .)

Conclusions: Parallel computing

- NumExpr for the simplest cases
- Numba, but don't use vectorized expressions.
- For non-lightweight tasks, look into the `multiprocessing` package.
- MPI4py is an option
- Cupy for CPUS, with vectorized expressions.
- Numba for GPUS, like CUDA kernels.

Note:

- Many ML/AI packages now support GPU accelleration.
- For workflows with dependencies, check out Dask or Ray.