Vectorized Programming in Python

or

How to make your scientific Python code blazingly fast on a CPU with Numpy and on a GPU with Cupy

Daniel Wysocki

Rochester Institute of Technology

RIT Scientific Computing Group
September 16th, 2019

R·I·T

Introduction

“Scalar” programming languages

- operate on arrays one element at a time (e.g., with a for loop)
- some examples:
  - C
  - Fortran 77
  - Java
  - Python

Vectorized programming languages

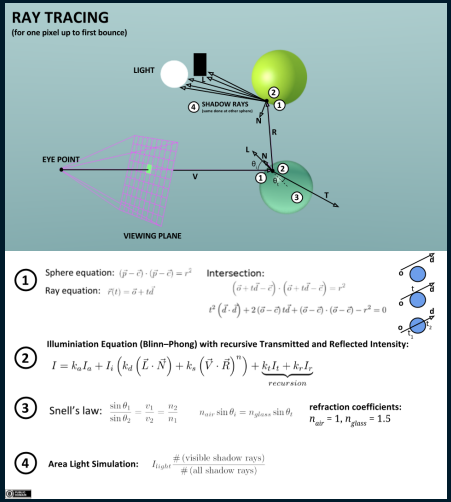
- operate on arrays as a whole
- some examples:
  - APL
  - Python with numpy or cupy
  - R
  - MATLAB
  - IDL
  - Julia

R·I·T

Scientific computing on a *graphics* processor?

R·I·T

Graphics requires lots of linear algebra and trig – really fast



Single Instruction, Multiple Data (SIMD)

Ford assembly line – public domain

R·I·T

NVIDIA CUDA

- Currently the best performing GPUs in the world are made by NVIDIA, and are most efficiently programmed using their proprietary (freeware) language `CUDA`
- `CUDA` is basically an extension to `C/C++`
- Very low-level, requiring understanding of how GPU's operate to get the most efficient code possible

R·I·T

CuPy – CUDA programming in NumPy style

- CuPy is a free and open source Python library, meant as a replacement for NumPy, but using CUDA under the hood
- It includes a large subset of NumPy's features, along with additional tools for low-level GPU stuff, and for converting data between CuPy and NumPy
- Available at https://cupy.chainer.org/

R·I·T

CuPy installation

Assuming you already have CUDA installed, you can install CuPy with the standard Python package manager pip.

```
# (For CUDA 8.0)
$ pip install cupy-cuda80
# (For CUDA 9.0)
$ pip install cupy-cuda90
# (For CUDA 9.1)
$ pip install cupy-cuda91
# (For CUDA 10.0)
$ pip install cupy-cuda100
# (For CUDA 10.1)
$ pip install cupy-cuda101

# (Install CuPy from source)
$ pip install cupy
```

R·I·T

Example: Multiplying arrays

Problem

$$\underbrace{\mathbf{A}}_{\substack{n \times m \\ \text{array}}} \times \underbrace{\mathbf{B}}_{\substack{n \times m \\ \text{array}}}$$

Problem

$$\underbrace{\mathbf{A}}_{\substack{n \times m \\ \text{array}}} \div \underbrace{\mathbf{B}}_{\substack{n \times m \\ \text{array}}}$$

Problem

$$\underbrace{\mathbf{A}}_{\substack{n \times m \\ \text{array}}} + \underbrace{\mathbf{B}}_{\substack{n \times m \\ \text{array}}}$$

Problem

$$\underbrace{\mathbf{A}}_{\substack{n \times m \\ \text{array}}} - \underbrace{\mathbf{B}}_{\substack{n \times m \\ \text{array}}}$$

In C (non-vectorized)

```c
#include "demoutils.h"
#define N 50000000
static int A[N], B[N], C[N];
int main() {
  for (int i = 0; i < N; ++i) { A[i] = i; B[i] = i*i; } /* Init arrays */
  clock_t t_i = clock(); /* Start timer */
  for (int i = 0; i < N; ++i) { C[i] = A[i] * B[i]; } /* Multiply numbers */
  clock_t t_f = clock(); /* End timer */
  printarr_int(C, 10); display_time(t_f - t_i); /* Output */
}
```

R·I·T

In C (non-vectorized)

```c
#include "demoutils.h"
#define N 50000000
static int A[N], B[N], C[N];
int main() {
  for (int i = 0; i < N; ++i) { A[i] = i; B[i] = i*i; } /* Init arrays */
  clock_t t_i = clock(); /* Start timer */
  for (int i = 0; i < N; ++i) { C[i] = A[i] * B[i]; } /* Multiply numbers */
  clock_t t_f = clock(); /* End timer */
  printarr_int(C, 10); display_time(t_f - t_i); /* Output */
}
```

```
$ gcc -lm -o multiply_nonvec multiply_nonvec.c demoutils.c
$ ./multiply_nonvec
0 1 8 27 64 125 216 343 512 729
Time elapsed: 250.00 msec
```

R·I·T

In Python (non-vectorized)

```
1 from demoutils import *
2 A = [i for i in range(50000000)]; B = [i*i for i in range(50000000)] # Init arrays
3 C = []
4
5 t_i = time.clock() # Start timer
6 for i in range(len(A)): C.append(A[i] * B[i]) # Multiply numbers
7 t_f = time.clock() # End timer
8 print(*C[:10]); display_time(t_f - t_i) # Output
```

R·I·T

In Python (non-vectorized)

```
1 from demoutils import *
2 A = [i for i in range(50000000)]; B = [i*i for i in range(50000000)] # Init arrays
3 C = []
4
5 t_i = time.clock() # Start timer
6 for i in range(len(A)): C.append(A[i] * B[i]) # Multiply numbers
7 t_f = time.clock() # End timer
8 print(*C[:10]); display_time(t_f - t_i) # Output
```

```
$ python multiply_nonvec.py
0 1 8 27 64 125 216 343 512 729
Time elapsed: 22.41 sec
```

R·I·T

In Python (vectorized with numpy)

```
1 from demoutils import *
2 A = numpy.arange(50000000); B = A*A # Init arrays
3 t_i = time.clock() # Start timer
4 C = A * B # Multiply arrays
5 t_f = time.clock() # End timer
6 print(*C[:10]); display_time(t_f - t_i) # Output
```

In Python (vectorized with numpy)

```
1 from demoutils import *
2 A = numpy.arange(50000000); B = A*A # Init arrays
3 t_i = time.clock() # Start timer
4 C = A * B # Multiply arrays
5 t_f = time.clock() # End timer
6 print(*C[:10]); display_time(t_f - t_i) # Output
```

```
$ python multiply_vec_np.py
0 1 8 27 64 125 216 343 512 729
Time elapsed: 160.00 msec
```

R·I·T

In Python (vectorized with cupy)

```
1 from demoutils import *
2 A = cupy.arange(50000000); B = A*A # Init arrays
3 t_i = time.clock() # Start timer
4 C = A * B # Multiply arrays
5 t_f = time.clock() # End timer
6 print(*C[:10]); display_time(t_f - t_i) # Output
```

R·I·T

In Python (vectorized with cupy)

```python
from demoutils import *
A = cupy.arange(50000000); B = A*A # Init arrays
t_i = time.clock() # Start timer
C = A * B # Multiply arrays
t_f = time.clock() # End timer
print(*C[:10]); display_time(t_f - t_i) # Output
```

```
$ python multiply_vec_cp.py
0 1 8 27 64 125 216 343 512 729
Time elapsed: 0.00 nsec
```

R·I·T

Operators

Arithmetic

- `A + B` ↔ `xpy.add(A, B)`
- `A - B` ↔ `xpy.subtract(A, B)`
- `A * B` ↔ `xpy.multiply(A, B)`
- `A / B` ↔ `xpy.divide(A, B)`
- `A // B` ↔ `xpy.floor_divide(A, B)`
- `A ** B` ↔ `xpy.power(A, B)`

R·I·T

Analytic functions

- `sin(x)`
- `cos(x)`
- `tan(x)`
- `arcsin(x)`
- `arccos(x)`
- `arctan(x)`
- `exp(x)`
- `log(x)`
- `log10(x)`
- `sqrt(x)`
- `square(x)`
- `reciprocal(x)`

R·I·T

Motivation: Operator versus function

- Operators
  - More concise and readable
- Functions
  - Some things don't have operators (like `sin(x)`)
  - Can enforce whether Numpy or Cupy is used (will raise Exception if mixed up)
  - More control

```
# numpy.add
add(
  x1, x2, /, out=None, *,
  where=True, casting='same_kind', order='K', dtype=None,
  subok=True[, signature, extobj]
)
```

R·I·T

Memory inefficient example

$$y = \exp\Big(\sin\big(\cos(x)\big)\Big)$$

```
1  from demoutils import *
2
3  x = numpy.arange(500000, dtype=float)
4  t_i = time.clock() # Start timer
5  y = numpy.exp(numpy.sin(numpy.cos(x)))
6  t_f = time.clock() # End timer
7
8  print(*y[:4]); display_time(t_f - t_i) # Output
```

R·I·T

Memory inefficient example

$$y = \exp\Big(\sin\big(\cos(x)\big)\Big)$$

```
1  from demoutils import *
2
3  x = numpy.arange(500000, dtype=float)
4  t_i = time.clock() # Start timer
5  y = numpy.exp(numpy.sin(numpy.cos(x)))
6  t_f = time.clock() # End timer
7
8  print(*y[:4]); display_time(t_f - t_i) # Output
```

```
$ python memory_inefficient.py
2.319776824715853 1.67262668915228 0.6674844706966705 0.43343134823715623
Time elapsed: 28.20 msec
```

R·I·T

Memory efficient example

$$y = \exp\Big(\sin\big(\cos(x)\big)\Big)$$

```
1  from demoutils import *
2
3  x = numpy.arange(500000, dtype=float)
4
5  t_i = time.clock() # Start timer
6  y = numpy.empty_like(x) # Initialize
7  numpy.cos(x, out=y); numpy.sin(y, out=y); numpy.exp(y, out=y)
8  t_f = time.clock() # End timer
9
10 print(*y[:4]); display_time(t_f - t_i) # Output
```

Memory efficient example

$$y = \exp\Big(\sin\big(\cos(x)\big)\Big)$$

```
1  from demoutils import *
2
3  x = numpy.arange(500000, dtype=float)
4
5  t_i = time.clock() # Start timer
6  y = numpy.empty_like(x) # Initialize
7  numpy.cos(x, out=y); numpy.sin(y, out=y); numpy.exp(y, out=y)
8  t_f = time.clock() # End timer
9
10 print(*y[:4]); display_time(t_f - t_i) # Output
```

```
$ python memory_efficient.py
2.319776824715853 1.67262668915228 0.6674844706966705 0.43343134823715623
Time elapsed: 26.16 msec
```

Working with N dimensions

N-dimensional arrays

- Arrays in Numpy/Cupy are N-dimensional arrays
- Same data type works for 0D (scalars), 1D (vectors), 2D (matrices), ...

```
>>> import numpy
>>> x = numpy.asarray([[1., 0.],
...                    [0., 1.]])
>>> y = numpy.ones((2, 2))
>>> x + y
array([[2., 1.],
       [1., 2.]])
>>> x.shape
(2, 2)
>>> x.size
4
>>> x.ndim
2
```

Broadcasting

[1] When operating on two arrays, NumPy/Cupy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1

Any dimension that only has one element is treated as if everything were repeated along that dimension enough times to make the two arrays compatible.

[1] From the numpy documentation `docs.scipy.org/doc/numpy/user/basics.broadcasting.html`

Broadcasting shape examples

```
A      (2d array):  5 × 4
B      (1d array):      1
Result (2d array):  5 × 4

A      (2d array):  5 × 4
B      (1d array):      4
Result (2d array):  5 × 4

A      (3d array):  15 × 3 × 5
B      (3d array):  15 × 1 × 5
Result (3d array):  15 × 3 × 5

A      (3d array):  15 × 3 × 5
B      (2d array):      3 × 5
Result (3d array):  15 × 3 × 5
```

R·I·T

Reductions

- Combines elements of an array with some operation, resulting in a new array with a smaller number of dimensions.
- You can specify which dimensions to reduce, or if un-specified, the reduction is applied until only a scalar remains.
- Examples:
  - `sum` `prod`
  - `min` `max` `argmin` `argmax`
  - `mean` `median` `std` `var`

R·I·T

Reduction example: `sum`

```
import numpy
>>> x = numpy.asarray([[0., 1.],
...                    [2., 3.],
...                    [4., 5.]])
>>> numpy.sum(x)
15.0

>>> numpy.sum(x, axis=0)
array([6., 9.])

>>> numpy.sum(x, axis=1)
array([1., 5., 9.])
>>> numpy.sum(x, axis=-1)
array([1., 5., 9.])

>>> numpy.sum(x, axis=(0, 1))
15.0
```

Indexing and control flow

Basic indexing

```
>>> x = array([[0, 1, 2, 3],
               [4, 5, 6, 7]])
>>> x[0,0]
0
>>> x[1,1]
5
>>> x[0,:]
array([0, 1, 2, 3])
>>> x[:,0]
array([0, 4])
>>> x[0]
array([0, 1, 2, 3])
```

R·I·T

Boolean indexing

```
>>> x = array([[0, 1, 2, 3],
               [4, 5, 6, 7]])
>>> idx = x > 5
>>> idx
array([[False, False, False, False],
       [False, False,  True,  True]])
>>> x[idx]
array([6, 7])
```

```
>>> x = array([1.3, 0.1, 20., 15., 0.15, 0.5])
>>> x[x < 0.5] = 0.0
>>> x
array([ 1.3,  0. , 20. , 15. ,  0. ,  0.5])
```

R·I·T

`where`: the vectorized `if` statement

```
>>> x = array([1.3, 0.1, 20., 15., 0.15, 0.5])
>>> where(x < 0.5, 0.0, x)
array([ 1.3,  0. , 20. , 15. ,  0. ,  0.5])

>>> y = array([5., 13., 0.1, 0.7, 8.3, 9.0])
>>> where(x > y, x, y)
array([ 5. , 13. , 20. , 15. ,  8.3,  9. ])
```

Array creation

Empty and constant

```
>>> empty(2) # non-deterministic initialization -- you must overwrite
array([7.74860419e-304, 7.74860419e-304])
>>> zeros(2)
array([0., 0.])
>>> ones(2)
array([1., 1.])
>>> full(2, 1234, dtype=float) # Put whatever fill value you want
array([1234., 1234.])

# Analogous functions which take another array instead of shape, matches size
>>> x = empty(4)
>>> zeros_like(x)
array([0., 0., 0., 0.])
## Same for `empty_like`, `ones_like`, and `full_like`
```

Ranges

```
>>> empty(2) # non-deterministic initialization -- you must overwrite
array([7.74860419e-304, 7.74860419e-304])
>>> zeros(2)
array([0., 0.])
>>> ones(2)
array([1., 1.])
>>> full(2, 1234, dtype=float) # Put whatever fill value you want
array([1234., 1234.])

# Analogous functions which take another array instead of shape, matches size
>>> x = empty(4)
>>> zeros_like(x)
array([0., 0., 0., 0.])
## Same for `empty_like`, `ones_like`, and `full_like`
```

Combining arrays

```
>>> x = linspace(0.0, 1.0, 3)
>>> y = linspace(10.0, 20.0, 3)
>>> row_stack((x, y))
array([[ 0. ,  0.5,  1. ],
       [10. , 15. , 20. ]])
>>> column_stack((x, y))
array([[ 0. , 10. ],
       [ 0.5, 15. ],
       [ 1. , 20. ]])
>>> concatenate((x, y))
array([ 0. ,  0.5,  1. , 10. , 15. , 20. ])
```

R·I·T

Making CuPy and Numpy work together

R·I·T

Mixing CuPy and NumPy arrays (wrong way)

```
>>> cupy.arange(0, 5) + cupy.arange(3, 8)
array([ 3,  5,  7,  9, 11])

>>> cupy.arange(0, 5) + numpy.arange(3, 8)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-9951f4a4a370> in <module>()
----> 1 cupy.arange(0, 5) + numpy.arange(3, 8)

cupy/core/core.pyx in cupy.core.core.ndarray.__add__()

cupy/core/elementwise.pxi in cupy.core.core.ufunc.__call__()

cupy/core/elementwise.pxi in cupy.core.core._preprocess_args()

TypeError: Unsupported type <type 'numpy.ndarray'>
```

Mixing CuPy and NumPy arrays (right way)

- must convert between CuPy and NumPy arrays to mix
  - cupy.asnumpy(): CuPy→NumPy
  - cupy.asarray(): NumPy→CuPy

```
>>> cupy.asnumpy(cupy.arange(0, 5)) + numpy.arange(3, 8)
array([ 3,  5,  7,  9, 11])

>>> type(cupy.asnumpy(cupy.arange(0, 5)) + numpy.arange(3, 8))
numpy.ndarray

>>> type(cupy.arange(0, 5) + cupy.asarray(numpy.arange(3, 8)))
cupy.core.core.ndarray
```

- Beware: slow process, should avoid everywhere possible

Writing CuPy/NumPy agnostic code

- Can write functions that work on both CuPy and NumPy

```
>>> def euler_formula(x):
...     "exp(i*x) = cos(x) + i*sin(x)"
...     # Can be either `numpy` or `cupy`.
...     xpy = cupy.get_array_module(x)
...     # Compute the result with the right library.
...     return xpy.cos(x) + 1j*xpy.sin(x)

>>> type(euler_formula(cupy.arange(10)))
cupy.core.core.ndarray

>>> type(euler_formula(numpy.arange(10)))
numpy.ndarray
```

R·I·T

Writing CuPy/NumPy agnostic code (optimized)

- For extra speed, you can save the call to `cupy.get_array_module`

```
>>> def euler_formula(x, xpy=cupy):
...     "exp(i*x) = cos(x) + i*sin(x)"
...     # Compute the result with the right library.
...     return xpy.cos(x) + 1j*xpy.sin(x)

>>> type(euler_formula(cupy.arange(10), xpy=cupy))
cupy.core.core.ndarray

>>> type(euler_formula(numpy.arange(10), xpy=numpy))
numpy.ndarray

>>> type(euler_formula(numpy.arange(10), xpy=cupy))
TypeError: Unsupported type <type 'numpy.ndarray'>
```

More on CuPy

More power – kernels

```
>>> squared_diff = cupy.ElementwiseKernel(
...     'float32 x, float32 y', # Input arrays
...     'float32 z', # Output array
...     'z = (x - y) * (x - y)', # Compute the result and store in output array.
...     'squared_diff', # Name the kernel
... )

>>> x = cupy.arange(10, dtype=np.float32).reshape(2, 5)
>>> y = cupy.arange(5, dtype=np.float32)
>>> squared_diff(x, y)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
>>> squared_diff(x, 5)
array([[25., 16.,  9.,  4.,  1.],
       [ 0.,  1.,  4.,  9., 16.]], dtype=float32)
```

R·I·T

Type-generic kernels

```
>>> squared_diff_generic = cupy.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_generic',
...     )
```

Map/Reduce kernels

```
>>> l2norm_kernel = cupy.ReductionKernel(
...     'T x',  # input params
...     'T y',  # output params
...     'x * x',  # map
...     'a + b',  # reduce,
...     'y = sqrt(a)',  # post-reduction map
...     '0',  # identity value
...     'l2norm'  # kernel name
... )
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> l2norm_kernel(x, axis=1)
array([ 5.477226 , 15.9687195], dtype=float32)
```

R·I·T

Bonus: utility functions from examples

demoutils.h

```c
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

void printarr_int(int arr[], int size);
void printarr_double(double arr[], int size);
void display_time(clock_t delta_t);
```

R·I·T

demoutils.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
void printarr_int(int arr[], int size) {
  for (int i = 0; i < size-1; ++i) printf("%d ", arr[i]);/*All but last number*/
  printf("%d\n", arr[size-1]);/*Last number and newline*/
}
void printarr_double(double arr[], int size) {
  for (int i = 0; i < size-1; ++i) printf("%f ", arr[i]);/*All but last number*/
  printf("%f\n", arr[size-1]);/*Last number and newline*/
}
void display_time(clock_t delta_t) {
  char s[16];
  double sec = ((double) delta_t) / CLOCKS_PER_SEC;
  if (sec < 1e-6)
    sprintf(s, "%.2f nsec", sec / 1e-9);
  else if (sec < 1e-3)
    sprintf(s, "%.2f usec", sec / 1e-6);
  else if (sec < 1e0)
    sprintf(s, "%.2f msec", sec / 1e-3);
  else
    sprintf(s, "%.2f sec", sec);
  printf("Time elapsed: %s\n", s);
}
```

demoutils.py

```python
import time, warnings, numpy
try:
    import cupy
except ImportError:
    import numpy as cupy
    warnings.warn("Cupy not installed, falling back to Numpy")

def display_time(delta_t):
    if delta_t < 1e-6:
        s = "{:.2f} nsec".format(delta_t / 1e-9)
    elif delta_t < 1e-3:
        s = "{:.2f} usec".format(delta_t / 1e-6)
    elif delta_t < 1e0:
        s = "{:.2f} msec".format(delta_t / 1e-3)
    else:
        s = "{:.2f} sec".format(delta_t)
    print("Time elapsed:", s)
```