ICNPG 2023

Python y un poquito de Julia

GPUs y python

Hace pasos de optimización durante la compilación

- Numba
- Cupy
- RAPIDS
- PyCUDA
- Etc…

desactualizada

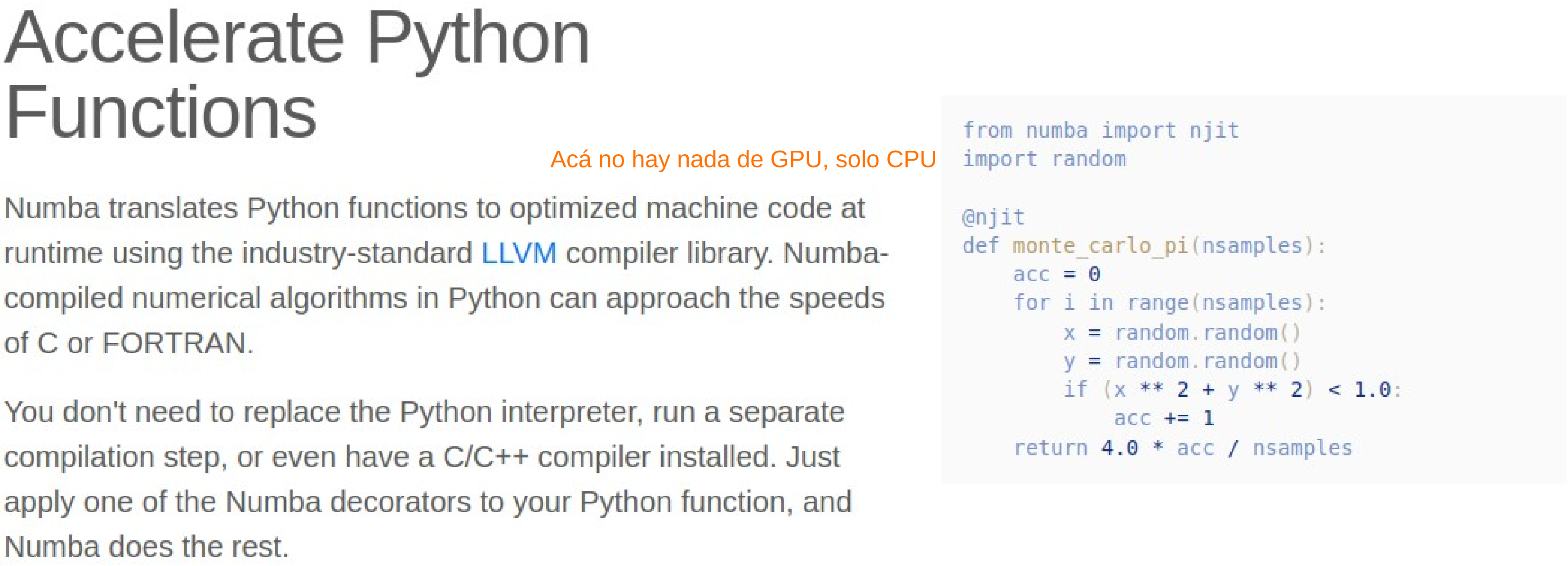
- **Interpreted languages:**
  - **Python, Ruby, or JavaScript, etc.**
  - **Execute code directly from the source code without the need for a compilation step.**
  - **often have a simpler syntax and are easier to learn, read, and write.**
  - **better support for dynamic typing and dynamic memory management.**
  - **slower than compiled languages because they must interpret the code at runtime.**
- Compiled languages:

  Lenguaje compilado que se parece mucho a python

  - C, C++, Fortran, Julia, etc.
  - require a compilation step to translate the source code into machine code.
  - tend to have a more complex syntax and require more experience to learn, read, and write.
  - They often have better support for static typing and memory management.

Numba

https://numba.pydata.org/



Accelerate Python Functions

Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.

You don't need to replace the Python interpreter, run a separate compilation step, or even have a C/C++ compiler installed. Just apply one of the Numba decorators to your Python function, and Numba does the rest.

```
from numba import njit
import random

@njit
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Acá no hay nada de GPU, solo CPU

Hay que hacer esto cuando se usa Cython por ejemplo

Numba

https://numba.pydata.org/

Portable Compilation

Ship high performance Python applications without the headache of binary compilation and packaging. Your source code remains pure Python while Numba handles the compilation at runtime. We test Numba continuously in more than 200 different platform configurations.

Numba supports Intel and AMD x86, POWER8/9, and ARM CPUs (including Apple M1), NVIDIA GPUs, Python 3.7-3.10, as well as Windows/macOS/Linux. Precompiled Numba binaries for most systems are available as conda packages and pip-installable wheels.

GPU Acceleration

With support for NVIDIA CUDA, Numba lets you write parallel GPU algorithms entirely from Python.

https://numba.readthedocs.io/en/stable/cuda/index.html

Numba

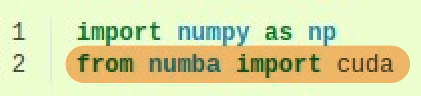
https://numba.pydata.org/

https://numba.readthedocs.io/en/stable/cuda/index.html

GPU Acceleration

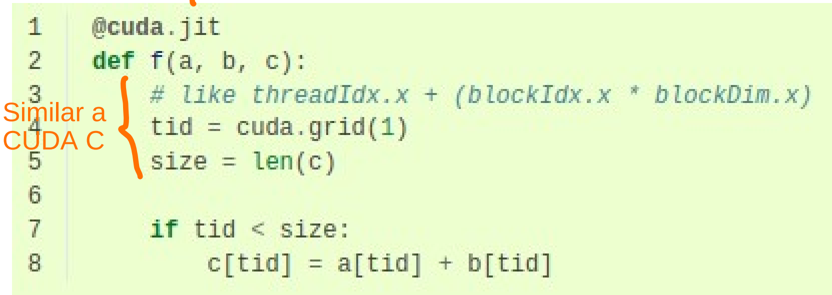
NVIDIA.
CUDA

With support for NVIDIA CUDA, Numba lets you write parallel GPU algorithms entirely from Python.



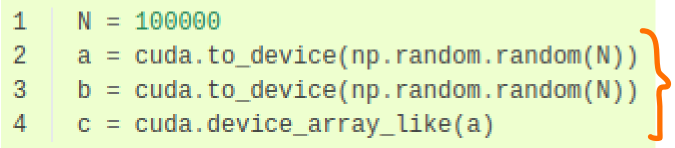
```python
import numpy as np
from numba import cuda
```

Por este decorador sabe que es un kernel



```python
@cuda.jit
def f(a, b, c):
    # like threadIdx.x + (blockIdx.x * blockDim.x)
    tid = cuda.grid(1)
    size = len(c)

    if tid < size:
        c[tid] = a[tid] + b[tid]
```

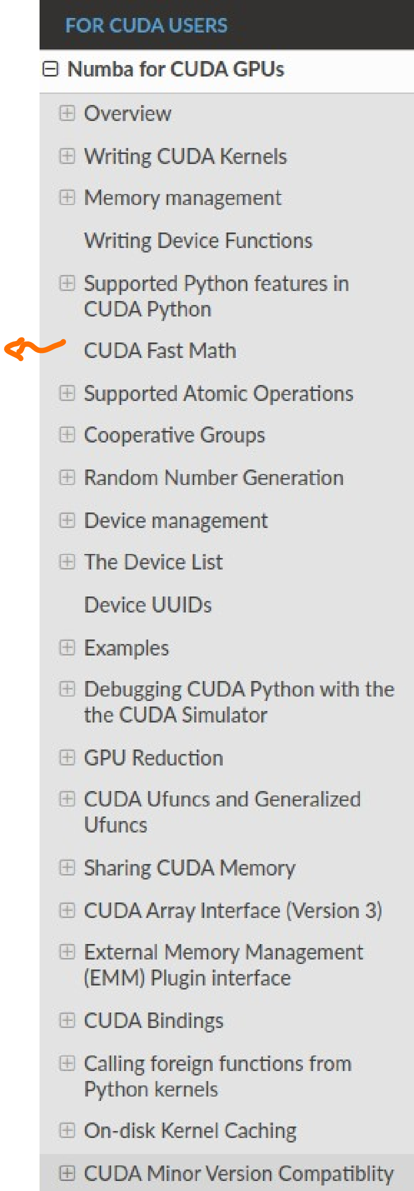
Similar a CUDA C



```python
N = 100000
a = cuda.to_device(np.random.random(N))
b = cuda.to_device(np.random.random(N))
c = cuda.device_array_like(a)
```

Por debajo se hace la alocación de memoria

Funciones optimizadas para GPU



FOR CUDA USERS

⊟ Numba for CUDA GPUs

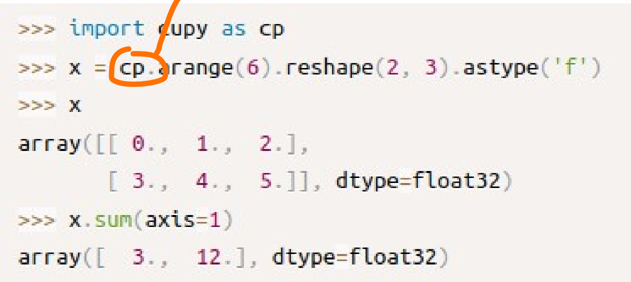
- ⊞ Overview
- ⊞ Writing CUDA Kernels
- ⊞ Memory management
- Writing Device Functions
- ⊞ Supported Python features in CUDA Python
- CUDA Fast Math
- ⊞ Supported Atomic Operations
- ⊞ Cooperative Groups
- ⊞ Random Number Generation
- ⊞ Device management
- ⊞ The Device List
- Device UUIDs
- ⊞ Examples
- ⊞ Debugging CUDA Python with the the CUDA Simulator
- ⊞ GPU Reduction
- ⊞ CUDA Ufuncs and Generalized Ufuncs
- ⊞ Sharing CUDA Memory
- ⊞ CUDA Array Interface (Version 3)
- ⊞ External Memory Management (EMM) Plugin interface
- ⊞ CUDA Bindings
- ⊞ Calling foreign functions from Python kernels
- ⊞ On-disk Kernel Caching
- ⊞ CUDA Minor Version Compatiblity

NumPy/SciPy-compatible Array Library for GPU-accelerated Computing with Python

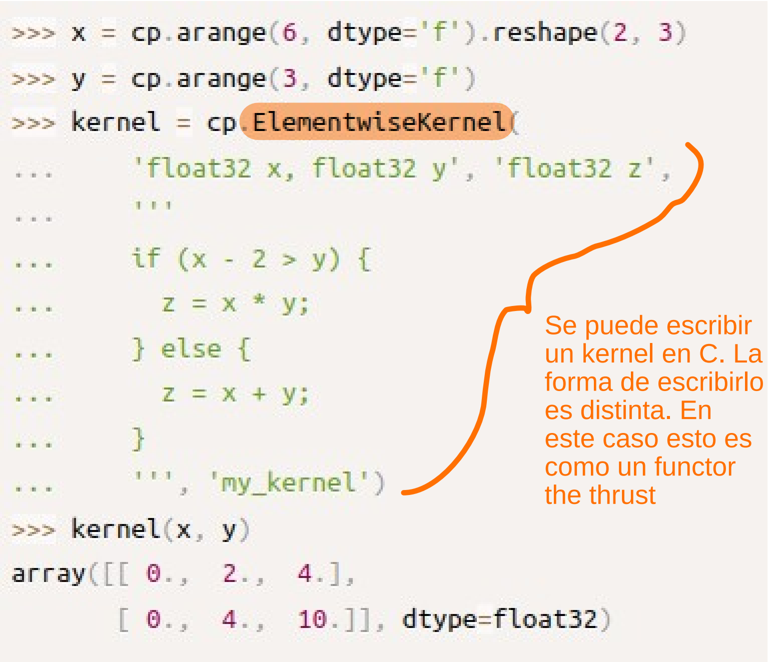
https://cupy.dev/

HIGHLY COMPATIBLE WITH NUMPY & SCIPY

Todo lo que arranque en "cp"
va a vivir en ((GPU))

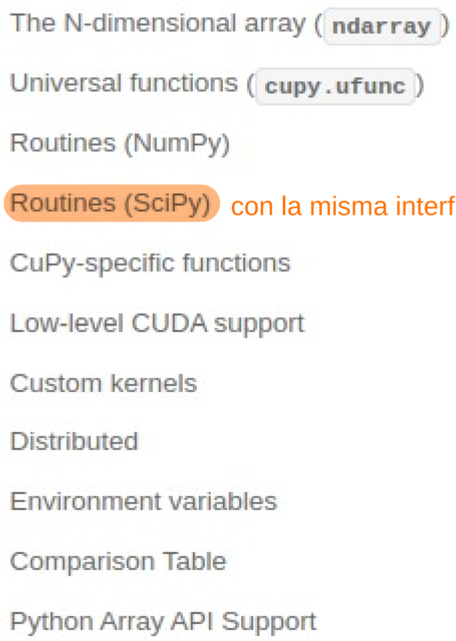


```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2, 3).astype('f')
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.sum(axis=1)
array([  3.,  12.], dtype=float32)
```



```
>>> x = cp.arange(6, dtype='f').reshape(2, 3)
>>> y = cp.arange(3, dtype='f')
>>> kernel = cp.ElementwiseKernel(
...     'float32 x, float32 y', 'float32 z',
...     '''
...     if (x - 2 > y) {
...       z = x * y;
...     } else {
...       z = x + y;
...     }
...     ''', 'my_kernel')
>>> kernel(x, y)
array([[ 0.,  2.,  4.],
       [ 0.,  4.,  10.]], dtype=float32)
```

Se puede escribir un kernel en C. La forma de escribirlo es distinta. En este caso esto es como un functor the thrust



The N-dimensional array (`ndarray`)

Universal functions (`cupy.ufunc`)

Routines (NumPy)

Tiene Routines (SciPy) con la misma interface!

CuPy-specific functions

Low-level CUDA support

Custom kernels

Distributed

Environment variables

Comparison Table

Python Array API Support

RAPIDS

(librería)

GPU Accelerated Data Science

FASTER PANDAS WITH CUDF

Para cuando se trabaja con muchos datos

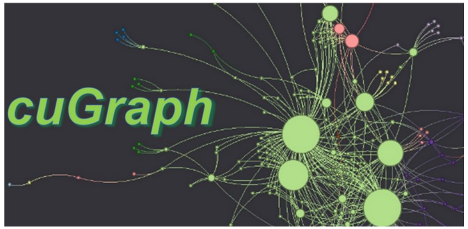
cuDF is a near drop in replacement to pandas for most use cases and has greatly improved performance.

FASTER SCIKIT-LEARN WITH CUML

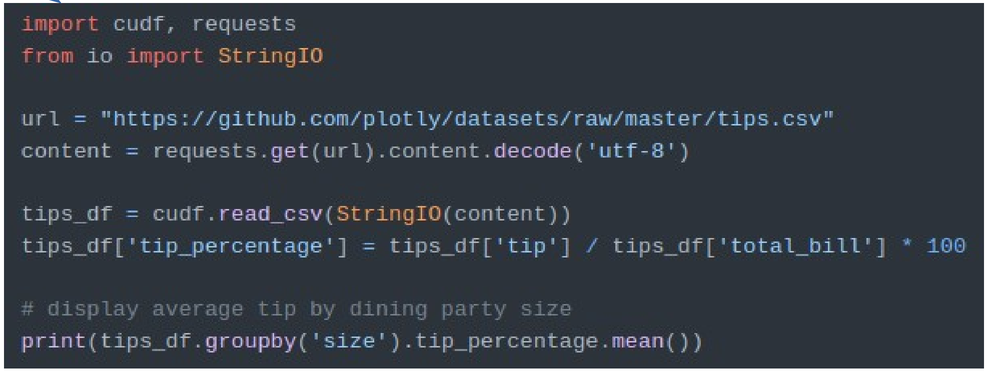
cuML brings huge speedups to ML modeling with an API that matches scikit-learn.

FASTER NETWORKX WITH CUGRAPH

cuGraph makes migration from networkX easy, accelerates graph analytics, and allows scaling far beyond existing tools.



- Dataframe processing with **cuDF** (similar API to pandas)
- Machine learning with **cuML** (similar API to scikit-learn)
- Graph processing with **cuGraph** (similar API to networkX)
- Spatial analytics with **cuSpatial** (similar API to geoPandas)
- Image processing with **cuCIM** (similar API to scikit-image)
- Signal processing **cuSignal** (similar API to scipy.signal)
- Seamless cross-filtered dashboards with **cuxfilter**
- Low level compute primitives with **RAFT**
- Apache Spark acceleration with **Spark RAPIDS**

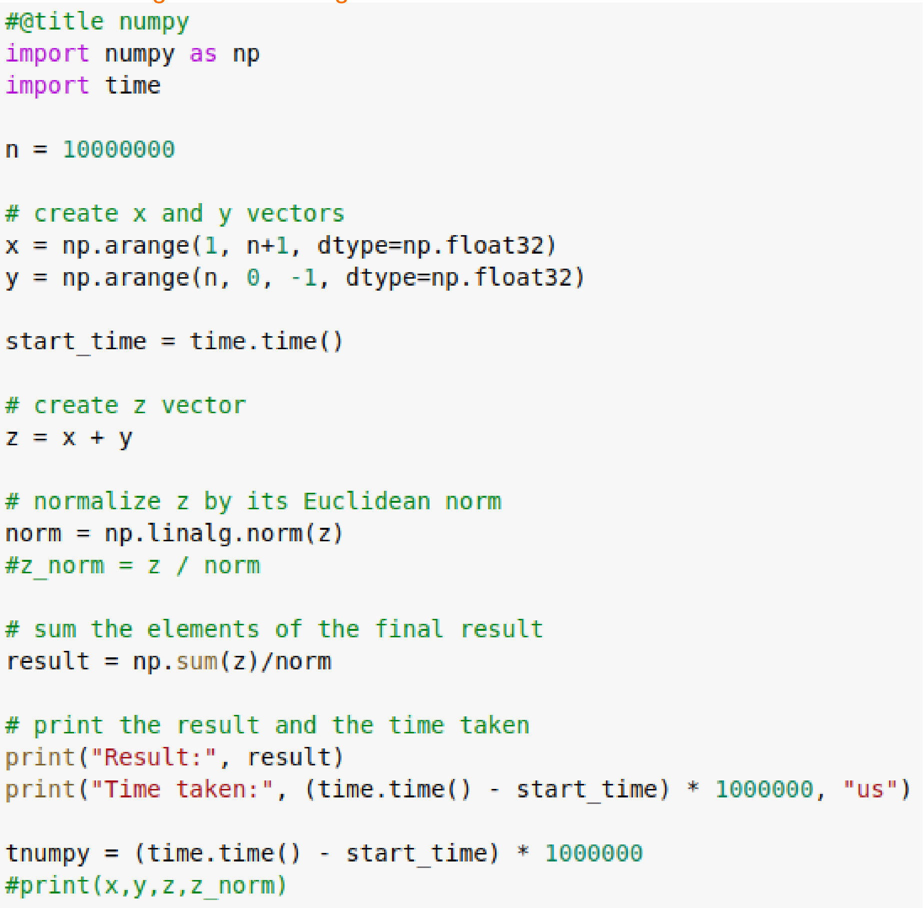


```python
import cudf, requests
from io import StringIO

url = "https://github.com/plotly/datasets/raw/master/tips.csv"
content = requests.get(url).content.decode('utf-8')

tips_df = cudf.read_csv(StringIO(content))
tips_df['tip_percentage'] = tips_df['tip'] / tips_df['total_bill'] * 100

# display average tip by dining party size
print(tips_df.groupby('size').tip_percentage.mean())
```

Permite acelerar muchas acciones usando GPU

Ejercicio

El código está en Google Colab



```
#@title numpy
import numpy as np
import time

n = 10000000

# create x and y vectors
x = np.arange(1, n+1, dtype=np.float32)
y = np.arange(n, 0, -1, dtype=np.float32)

start_time = time.time()

# create z vector
z = x + y

# normalize z by its Euclidean norm
norm = np.linalg.norm(z)
#z_norm = z / norm

# sum the elements of the final result
result = np.sum(z)/norm

# print the result and the time taken
print("Result:", result)
print("Time taken:", (time.time() - start_time) * 1000000, "us")

tnumpy = (time.time() - start_time) * 1000000
#print(x,y,z,z_norm)
```

Esta operación es una reducción porque S es un escalar

$$S = \frac{\sum_{i=0}^{n}(x_i + y_i)}{\sqrt{\sum_{i=0}^{n}(x_i + y_i)^2}}$$

Consideraciones:
* Los redondeo pueden no ser iguales en CPU y GPU
* Recordar que al medir los tiempos hay que hacer un "precalentamiento"

En orden de magnitud, respecto a Numpy
* Cupy tiene una aceleración x10
* Thrust tiene una aceleración x20
* Thrust-optimized tiene una aceleración x40

Ejercicio: hacerlo en CUPY y NUMBA

Consideraciones:
NRTC es un compilador de nvidia para compilar durante la ejecución del programa! De este modo es posible compilar un kernel con CuPy usando un string de python

Tasa de contagio

$$\frac{dS}{dt} = -\beta S I,$$

$$\frac{dI}{dt} = \beta S I - \gamma I,$$

$$\frac{dR}{dt} = \gamma I.$$

Método de Euler

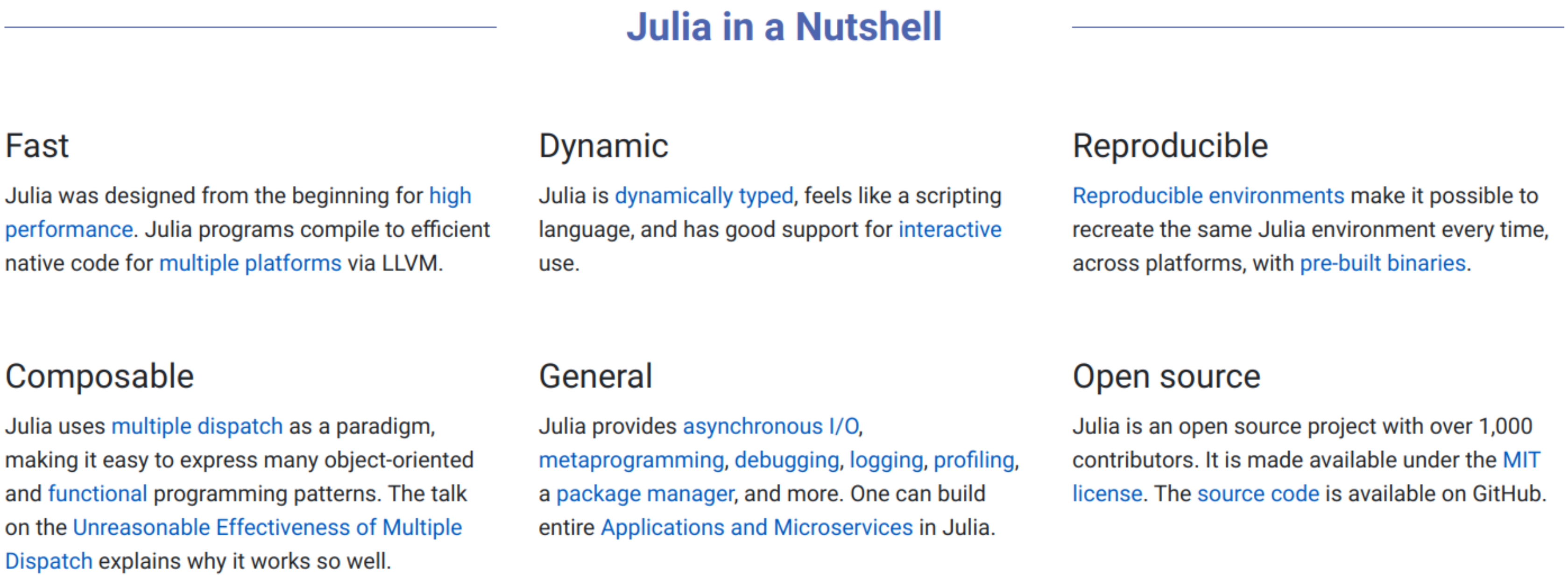
$$S(t+\Delta t) = S(t) + \Delta t[-\beta I(t) S(t)]$$

$$I(t+\Delta t) = I(t) + \Delta t[\beta I(t) S(t) - \gamma I(t)]$$

$$R(t+\Delta t) = R(t) + \Delta t[\gamma I(t)]$$

Suceptible, **I**nfectado, **R**ecuperado https://julialang.org/

Julia is a good choice for numerical computing and scientific computing tasks that require **high performance and parallelism**, while Python is a good choice for general-purpose programming, web development, and data analysis tasks that require a large ecosystem of libraries and tools.



Julia in a Nutshell

Fast

Julia was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM.

Dynamic

Julia is dynamically typed, feels like a scripting language, and has good support for interactive use.

Reproducible

Reproducible environments make it possible to recreate the same Julia environment every time, across platforms, with pre-built binaries.

Composable

Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns. The talk on the Unreasonable Effectiveness of Multiple Dispatch explains why it works so well.

General

Julia provides asynchronous I/O, metaprogramming, debugging, logging, profiling, a package manager, and more. One can build entire Applications and Microservices in Julia.

Open source

Julia is an open source project with over 1,000 contributors. It is made available under the MIT license. The source code is available on GitHub.

CUDA programming in Julia

The CUDA.jl package is the main entrypoint for programming NVIDIA GPUs in Julia. The package makes it possible to do so at various abstraction levels, from easy-to-use arrays down to hand-written kernels using low-level CUDA APIs.