Scientific Software Development with Python

High performance computing and big data

Simon Pfreundschuh
Department of Space, Earth and Environment

CHALMERS
UNIVERSITY OF TECHNOLOGY

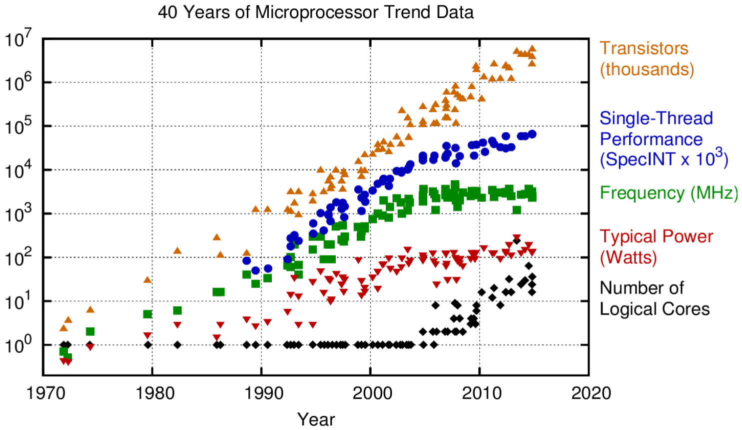
In this lecture

CHALMERS UNIVERSITY OF TECHNOLOGY

1. Solve heat equation on a GPU
2. Solve the heat equation in a smarter way
3. Distributed computing using IPython Parallel

CHALMERS
UNIVERSITY OF TECHNOLOGY

- Need to go parallel to go faster



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Image source: https://www.karlrupp.net

1. Introduction

2. Programming hardware accelerators

3. The heat equation revisited

4. Distributed computing with IPython Parallel

- Hardware accelerators are special computer hardware designed to speed up specific tasks
- Most commonly used today: Graphic processing units (GPUs)
- Originally designed to display 3D graphics
- Example: Nvidia T4 (Available on Vera@C3SE)
  - More than 2500 cores
  - 320 of which special 'tensor' cores for machine learning

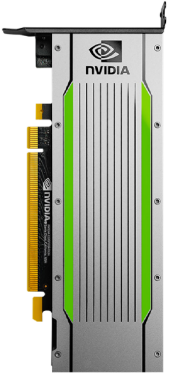
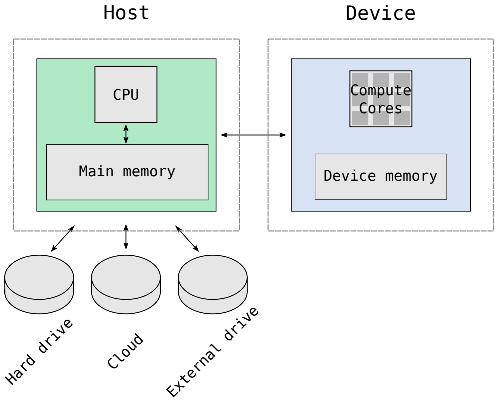


Image source: nvidia.com

CHALMERS UNIVERSITY OF TECHNOLOGY

Difficulties

- Usually have their own separate memory (transfer bottleneck)
- Require special programming techniques to program



GPU programming with Python

- Python won't run directly on GPU.
- GPUs used through some kind of special array or tensor type
- Wide range of packages that allow almost platform-agnostic[1] computing across different hardware
- Many of them are behind the recent consolidation of applicaitons of deep learning in science.

PyTorch TensorFlow CuPy

[1] Platform agnostic: Same code can run on CPU, GPU or whatever hardware is available.

Introducing CuPy

- CUDA is NVIDIA[2]'s computing and programming platform
- CuPY provides drop-in replacement for numpy arrays to accelerate array operations.
- Not all numpy operations implemented but this is the easiest way to perform calculations on GPU.

Installation

```
pip install cupy
```

- Or depending on your CUDA version:

```
pip install cupy-cudaXX
```

[2]NVIDIA is essentially the Intel of GPUs

Matrix multiplication example

- Before it can be used on the GPU, data must be transferred to the device.
- This is done by converting the `numpy.array` into a `cupy.array`

```
import numpy as np
import cupy as cp

n = 2048
# Create matrix and vector on host.
matrix = np.random.rand(n, n)
vector = np.random.rand(n)

# Transfer matrix and vector to GPU.
matrix_gpu = cp.asarray(matrix)
vector_gpu = cp.asarray(vector)

result = np.dot(matrix, vector
result_gpu = cp.do(matrix_gpu, vector_gpu)
```

Platform agnostic matrix multiplication

- Use `cupy.get_array_module` to get module object (`np` or `cp`) corresponding to array.

```
def matrix_multiplication(matrix, vector):
    xp = cp.get_array_module(matrix)
    return xp.dot(matrix, vector)

result = matrix_multiplication(matrix, vector)
result_gpu = matrix_multiplication(matrix_gpu, vector_gpu)
```

Performance exmample

- NVIDIA Tesla T4 vs. Intel Xeon (2 cores)
- Task probably not heavy enough to show full potential of GPUs.

```
%timeit matrix_multiplication(matrix, vector)
>>> 1000 loops, best of 3: 1.51 ms per loop
%timeit matrix_multiplication(matrix_gpu, vector_gpu)
>>> 10000 loops, best of 3: 139 µs per loop
```

Exercise 1

- Exercise 1 from notebook
- Time: 15 minutes

Summary

- If you have GPU and CUDA installed `cupy` can provide an easy way to accelerate your computations.
- However, using your GPU through Python will be limited by the functionality provided by the package you are using.[3]
- Other packages that can be used to compute on GPUs:
  - Theano, Numba, TensorFlow, PyTorch, ...

[3] But it is much simpler to do.

1. Introduction

2. Programming hardware accelerators

3. The heat equation revisited

4. Distributed computing with IPython Parallel

The heat equation

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} \right) \tag{1}$$

- For the simple case of $\alpha = 1$, the heat equation can be solved explicitly.

1. Assuming that u can be written as a function of the form

$$u(t, x, y) = T(t) \cdot X(x) \cdot Y(y) \tag{2}$$

2. The problem can be transformed to a system of coupled *ordinary* differential equations:

$$\frac{\partial^2 X}{\partial^2 x} = A \cdot X \tag{3}$$

$$\frac{\partial^2 Y}{\partial^2 y} = B \cdot Y \tag{4}$$

$$\frac{\partial T}{\partial t} = (A + B) \cdot Y \tag{5}$$

- From this we find that a general solution of the heat equation is given by:

$$u(t, x, y) = \sum_{m,n} A_{m,n}\ e^{i\frac{2\pi m}{N}x}\ e^{i\frac{2\pi n}{N}y}\ e^{-\frac{4\pi^2(n^2+m^2)}{N^2}t} \tag{6}$$

- We can thus also solve the heat equation as follows:
  1. Use a 2D Fourier transform to calculate the Fourier coefficients $A_{m,n}(0)$ from the initial heat distribution $u(0, x, y)$.
  2. Multiply coefficients $A_{m,n}(0)$ by $e^{-\frac{4\pi^2(n^2+m^2)}{N^2}t}$ to obtain coefficients $A_{m,n}(t)$
  3. Calculate $u(t, x, y)$ by calculating the inverse Fourier transform of $A_{m,n}(t)$

Exercise 2

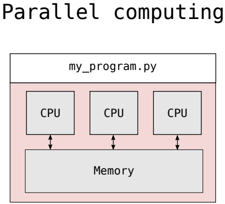
- Exercise 2 from notebook.
- Time: 20 minutes.

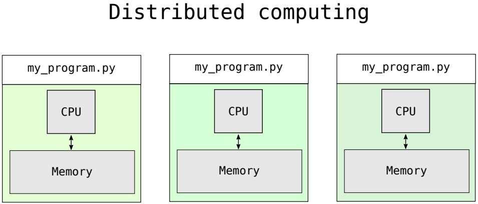
Agenda

CHALMERS UNIVERSITY OF TECHNOLOGY

1. Introduction
2. Programming hardware accelerators
3. The heat equation revisited
4. **Distributed computing with IPython Parallel**

- A program that executes concurrently across different computers
- Instances of the program typically do not share memory
- Special messaging functions required for communication
- Popular software packages:
  - High performance computing: Message passing interface (MPI)
  - Big data: Hadoop, Dask





Shared-memory parallelism

- Typically implemented using threads
- Processes can communicate through shared memory
- Low overhead
- Limited to one computer

Distributed parallelism

- Typically implemented using processes
- Larger overhead than threads
- Can usually run on multiple computers

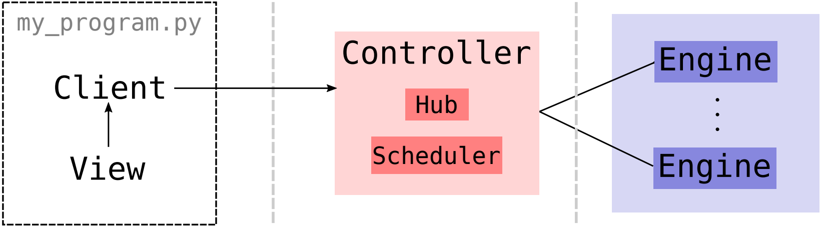
Note

It is not uncommon to see shared-memory parallelism combined, i.e. to have a programm running multiple threads in multiple processes distributed across a compute cluster.

CHALMERS
UNIVERSITY OF TECHNOLOGY

IPython Parallel (`ipyparallel`)

- Distributed-computing package for IPython
- Engines can run *locally* or on different computers (through e.g. SSH or MPI)
- Can be used interactively



Engines

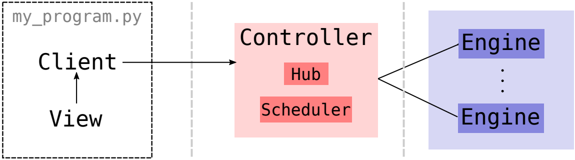
- A Python process to which you can send code for execution

Controller

- Local process to which engines connect
- Interface through which communication with engines takes place

`Client` and `View`

- Python objects to connect to controller and interact with engines.



Installation

```
pip install ipyparallel
```

Starting controller and engines

```
ipcluster start -n 4 # Will start controller and 4 engines locally
```

Connecting to the controller

```
from ipyparallel import Client
client = Client()
print(client.ids) # Prints: [0, 1, 2, 3]
view = client.direct_view()
```

Distributed hello world

- A view can be used to execute code on the engines.
- `apply` executes a method on all engines.
- However, since these engines run in different processes no output is produced in the client application.

```
def say_hi():
    import os
    print(f"Hi from process {os.getpid()}")

results = view.apply(say_hi) # Prints nothing.
```

Distributed hello world

- However, the returned `AsyncResult` object let's us access the output from each process:

```
results.display_outputs()
```

Output

```
[stdout:0] Hi from process 10557
[stdout:1] Hi from process 10569
[stdout:2] Hi from process 10570
[stdout:3] Hi from process 10573
```

- Complete exercise 3 from notebook
- Time: 10 minutes

Blocking and non-blocking execution

- `apply` executes the given function *asynchonously*, i.e. it returns immediately and returns an `AsyncResult` as place holder
- `apply_sync` is a blocking version of `apply` and returns results immediately

```
def get_integer():
    return int()

result = view.apply_sync(get_integer)
print(result) # Prints: [0, 0, 0, 0]
```

- Most other methods accept a `block` keyword arguments which defines their behavior.
- I will use `blocking` behavior in the following example because it makes effects directly visible.
- In general, however, asynchronous behavior is more powerful because it allows monitoring the processing state.

CHALMERS
UNIVERSITY OF TECHNOLOGY

Complications

- The client program and the engines don't share state:

```
import os

def say_hi():
    print(f"Hi from process {os.getpid()}")

results = view.apply_async(say_hi)
```

```
[0:apply]:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)<string> in <module>
<ipython-input-49-727b728ca0b2> in say_hi()
NameError: name 'os' is not defined

[1:apply]:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)<string> in <module>
<ipython-input-49-727b728ca0b2> in say_hi()
NameError: name 'os' is not defined

[2:apply]:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)<string> in <module>
<ipython-input-49-727b728ca0b2> in say_hi()
NameError: name 'os' is not defined
```

CHALMERS
UNIVERSITY OF TECHNOLOGY

Normal code

ipyparallel

1. `sync_imports`:
    - Works only with `DirectView` objects[4]
    - Can't assign aliases for imports

```python
with view.sync_imports():
    import numpy
```

2. `execute`:
    - Executes code on engines.

```python
view.execute('import numpy as np')
```

3. `require` decorator

```python
@ipp.require('os') # Or ipp.require(os) is os is already imported
def say_hi():
    print(f"Hi from process {os.getpid()}")
```

[4] We'll see more details later.

1. `push` and `pull`

```
view.push({a: 1, b: 2})
a = view.pull('a', block=True)
print(a) # Prints: [1, 1, 1, 1]
```

2. Dictionary interface

```
view['a'] = 2
a = view['a']
print(a) # Prints: [2, 2, 2, 2]
```

3. `scatter` and gather:
   - *Distributes* data across engines:

```
view.scatter('a', [1] * 16)
sums = view.apply(lambda : sum(a))
print(sums) # Prints: [4, 4, 4, 4]
a = view.gather('a', block=True)
print(a) # Prints: [1, ..., 1]
```

Executing code on engines

1. `execute`:
    - Executes code give as string

```
view.execute('import numpy as np')
```

2. `apply`, `apply_async` and `apply_sync`:
    - Executes function on engines

```
def my_function(a, b):
    return a + b
result = view.apply_sync(my_function, 1, 2)
print(result) # Prints: [3, 3, 3, 3]
```

3. `map`:
    - Maps function to range of arguments across engines:

```
@ipp.require('os')
def get_pid(dummy_argument):
    return os.getpid()
result = view.map(get_pid, range(4), block=True)
print(result) # Prints: [10557, 10569, 10570, 10573]
```

Direct view

- Provides *direct* access to engines
- Created from `Client` object using either `direct_view` method or list indexing:

```
view = client[::2]
result = view.map(get_pid, range(4), block=True)
print(result) # Prints: [10557, 10557, 10570, 10570]
```

Load balanced view

- Tasks are distributed dynamically in order to balance load
- Can’t target specific engines

Exercise 4

- Exercise 4 in notebook
- Time: 5 minutes

%px

- Executes line of code on engines

```
%px import numpy as np
%px a = np.random.rand(1)
print(view.scatter('a')) # Prints: [0.05926484 0.23279085 0.74808488 0.80716102]
```

%%px

- This is cell magic, i.e. work only in notebooks
- Executes all cell content on engines
- `%%px --block` will display last result from each engine

%autopx

- Will execute all subsequent commands on engines until next occurrence of `autopx`

Exercise 5

- Exercise 4 in notebook
- Time: 15 minutes