# Data Science Upskilling Workshop

## Session 4: Advanced Techniques: Parallel Computing and Machine Learning

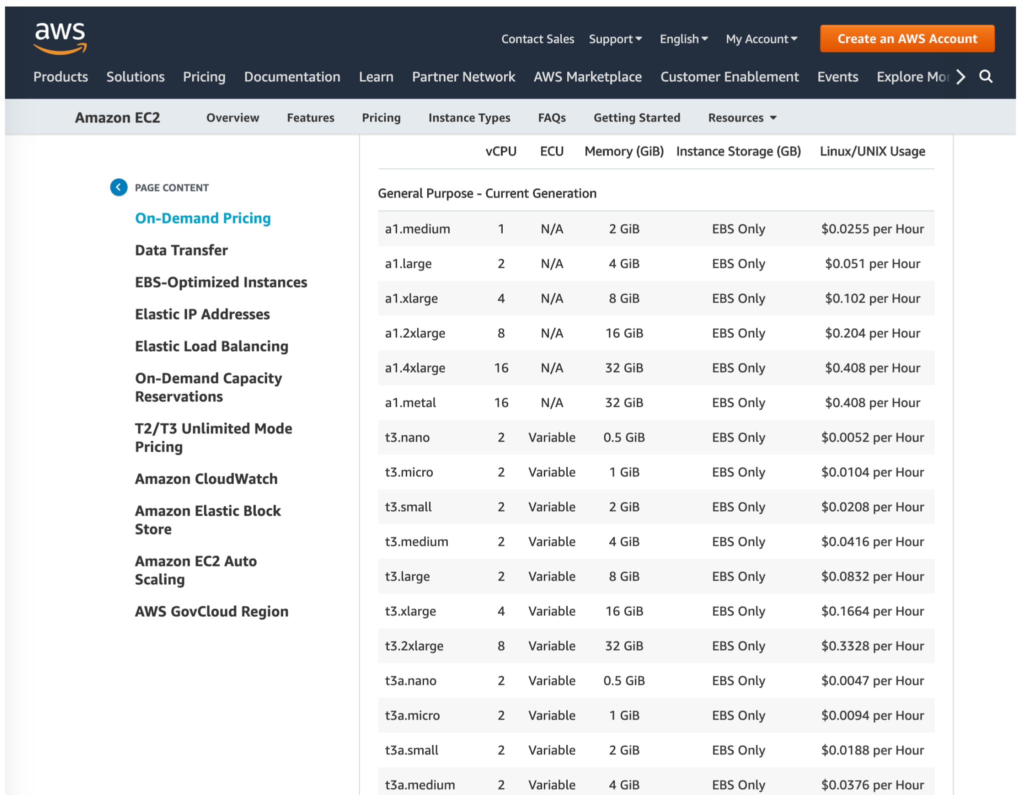
Oren Livne and Jiangang Hao
Educational Testing Service

NCME 2022 Training Workshop – 4/11/2022

# Speeding up Your Code

# Speeding-up your codes matters

Get things done vs. get things done under time and budget constraints



aws

Contact Sales | Support | English | My Account | Create an AWS Account

Products | Solutions | Pricing | Documentation | Learn | Partner Network | AWS Marketplace | Customer Enablement | Events | Explore Mor

Amazon EC2 | Overview | Features | Pricing | Instance Types | FAQs | Getting Started | Resources

PAGE CONTENT

- On-Demand Pricing
- Data Transfer
- EBS-Optimized Instances
- Elastic IP Addresses
- Elastic Load Balancing
- On-Demand Capacity Reservations
- T2/T3 Unlimited Mode Pricing
- Amazon CloudWatch
- Amazon Elastic Block Store
- Amazon EC2 Auto Scaling
- AWS GovCloud Region

| | vCPU | ECU | Memory (GiB) | Instance Storage (GB) | Linux/UNIX Usage |
|---|---|---|---|---|---|
| General Purpose - Current Generation | | | | | |
| a1.medium | 1 | N/A | 2 GiB | EBS Only | $0.0255 per Hour |
| a1.large | 2 | N/A | 4 GiB | EBS Only | $0.051 per Hour |
| a1.xlarge | 4 | N/A | 8 GiB | EBS Only | $0.102 per Hour |
| a1.2xlarge | 8 | N/A | 16 GiB | EBS Only | $0.204 per Hour |
| a1.4xlarge | 16 | N/A | 32 GiB | EBS Only | $0.408 per Hour |
| a1.metal | 16 | N/A | 32 GiB | EBS Only | $0.408 per Hour |
| t3.nano | 2 | Variable | 0.5 GiB | EBS Only | $0.0052 per Hour |
| t3.micro | 2 | Variable | 1 GiB | EBS Only | $0.0104 per Hour |
| t3.small | 2 | Variable | 2 GiB | EBS Only | $0.0208 per Hour |
| t3.medium | 2 | Variable | 4 GiB | EBS Only | $0.0416 per Hour |
| t3.large | 2 | Variable | 8 GiB | EBS Only | $0.0832 per Hour |
| t3.xlarge | 4 | Variable | 16 GiB | EBS Only | $0.1664 per Hour |
| t3.2xlarge | 8 | Variable | 32 GiB | EBS Only | $0.3328 per Hour |
| t3a.nano | 2 | Variable | 0.5 GiB | EBS Only | $0.0047 per Hour |
| t3a.micro | 2 | Variable | 1 GiB | EBS Only | $0.0094 per Hour |
| t3a.small | 2 | Variable | 2 GiB | EBS Only | $0.0188 per Hour |
| t3a.medium | 2 | Variable | 4 GiB | EBS Only | $0.0376 per Hour |

# Ways to speed up

- Better algorithms
- Better programming practices
- Compilation into binary executable
- Parallelization
- Hardware acceleration

# Speed up by compilation

- **Source code**: what the human writes
- **Bytecode**
  - Interpreted languages (such as Java, Python, Matlab), compile source codes to a set of instructions for a virtual machine.
  - The language interpreter is an implementation of that virtual machine
  - The instructions are in an intermediate format called bytecode
- **Machine code**
  - Binary codes that can be executed by CPU

- Complied language, such as C, C++, Fortran, directly compile the source codes into machine codes, so it is fast
- Can we do some something similar for Python to speed up?

# JIT

- Just In Time Compilation: compile the source codes to machine codes when (actually slightly before) running the program.
- Numba package in Python

Numba

Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

Learn More | Try Numba »

## 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.

Learn More » | Try Now »

```python
from numba import jit
import random

@jit(nopython=True)
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
```

# Ufunc and Vectorization

- Ufunc: universal function is a function that operates on ndarrays in an element-by-element fashion.
- Ufuncs are used to implement *vectorization* in NumPy which is way faster than iterating over elements.
- Vectorization: converting iterative statements into a vector-based operation. It is faster as modern CPUs are optimized for such operations.
- Numba also makes it easy to create ufuncs

https://numpy.org/doc/stable/reference/ufuncs.html

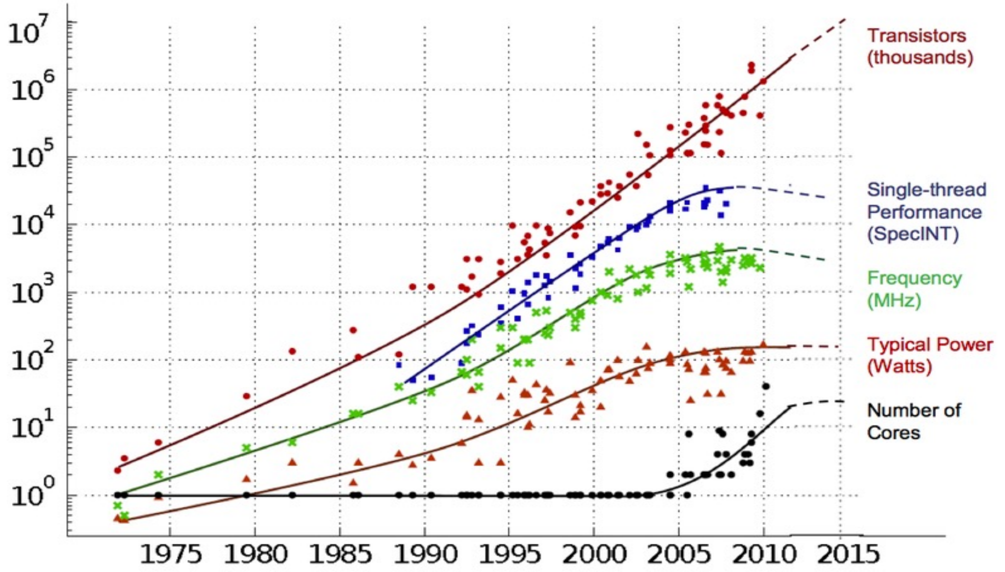
Notebook demo

# Important Note

- Performance is important.
- Readability and reliability are also important.
- KISS - keep it stupid and simple.

# Speed up by parallelization

- Single core performance is not increasing much since 2005
- A single CPU now has increasingly more cores
- Parallelization is the trend

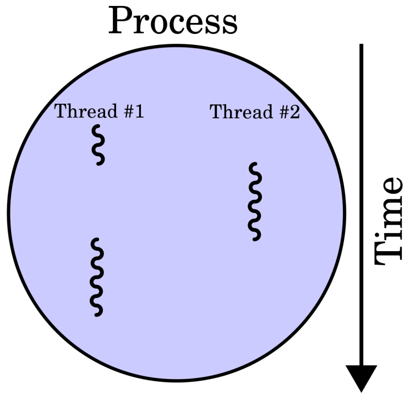



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

# Process and thread

- A **process** is the execution of a program that allows you to perform the appropriate actions specified in a program.
- It can be defined as an execution unit where a program runs.
- Memory is not shared among different processes.

- A **thread** is an execution unit that is part of a process. A process can have multiple threads, all executing at the same time.
- It is a unit of execution in concurrent programming.
- Memory is shared among threads of the same process.




Using multiple threads is tricky. We discuss parallelism using multiple processes only.

# Multiprocessing Package

- Multiprocessing is a package that supports spawning processes using an API.
- Multiprocessing module allows the programmer to fully leverage multiple processors on a given machine.

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

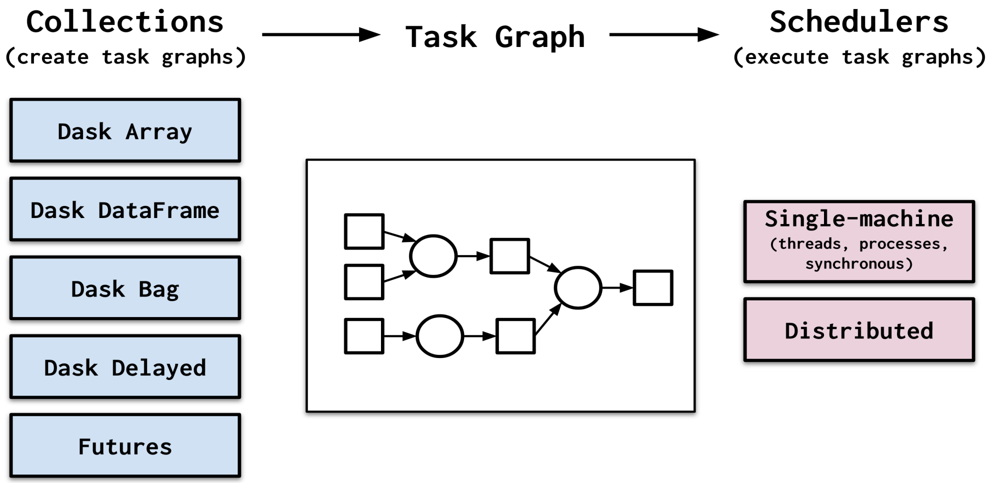
will print to standard output

```
[1, 4, 9]
```

https://docs.python.org/3/library/multiprocessing.html

# Dask

- Dask is a flexible library for parallel computing in Python https://docs.dask.org.
- Dask has two parts:
  - **"Big Data" collections** like parallel arrays, dataframes, and lists that extend common interfaces like *NumPy, Pandas, or Python iterators* to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.
  - **Dynamic task scheduling** optimized for computation. This is similar to *Airflow, Luigi, Celery, or Make*, but optimized for interactive computational workloads.



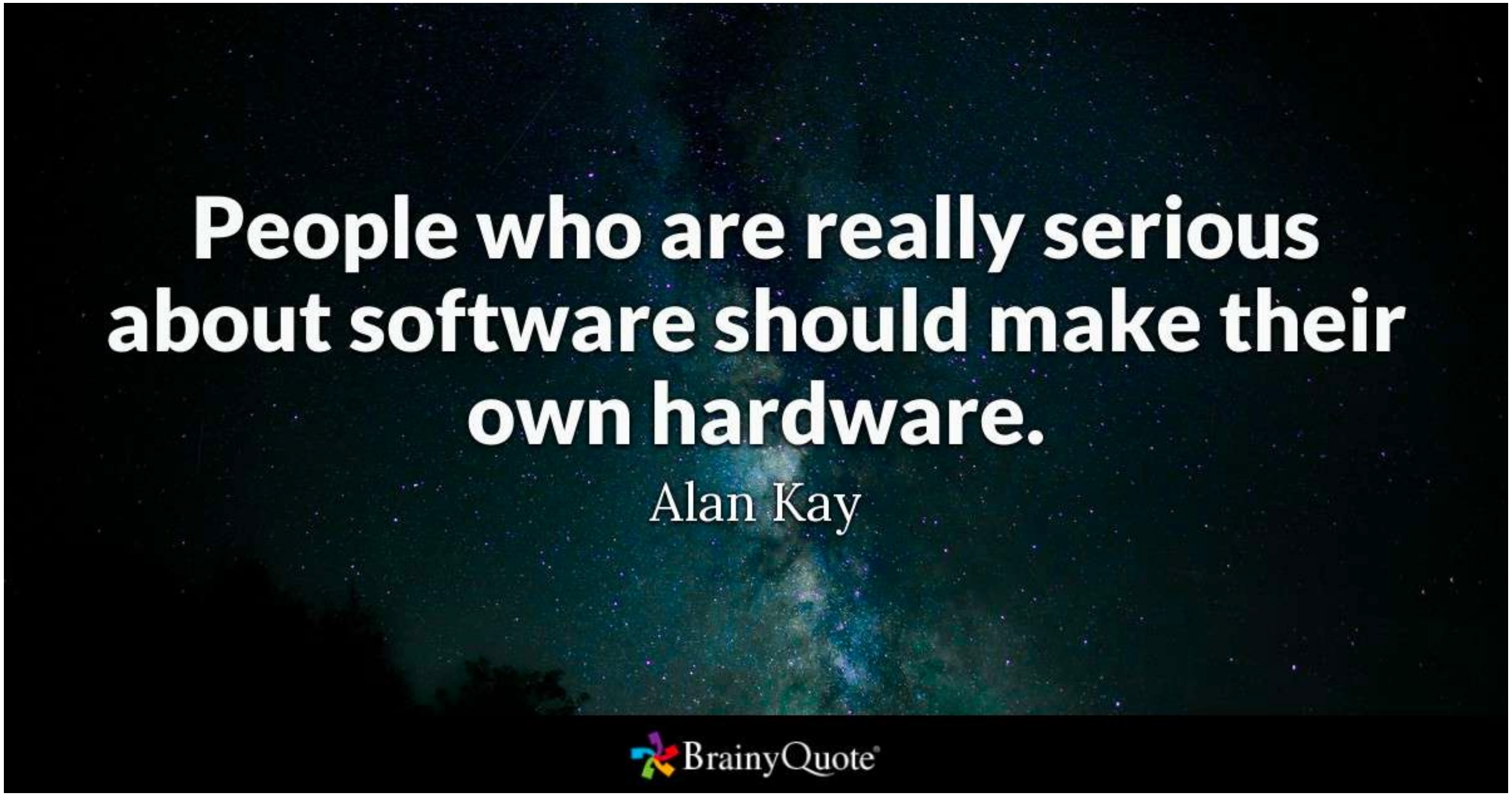
# Dask: Data Collections

- **Array** implements a subset of the NumPy ndarray interface using blocked algorithms, cutting up the large array into many small arrays. This lets us compute on arrays larger than memory using all of our cores.
- **Bag** implements operations like map, filter, fold, and groupby on collections of generic Python objects. It does this in parallel with a small memory footprint using Python iterators.
- **DataFrame** is a large parallel DataFrame composed of many smaller Pandas DataFrames, split along the index. These Pandas DataFrames may live on disk for larger-than-memory computing on a single machine, or on many different machines in a cluster. One Dask DataFrame operation triggers many operations on the constituent Pandas DataFrames.
- **Delayed**: sometimes problems don't fit into one of the collections like dask.array or dask.dataframe. In these cases, users can parallelize custom algorithms using the simpler dask.delayed interface. This allows one to create graphs directly with a light annotation of normal python code.

# Scheduling

- Single machine
  - Thread
  - Process
  - Synchronous – single thread
- Distributed
  - Single machine
  - Cluster

# Speed up by hardware



# CPU, GPU, and TPU

## Performance

As a comparision, consider this:

- CPU can handle tens of operation per cycle
- GPU can handle tens of thousands of operation per cycle
- TPU can handle upto 128000 operations per cycle

## Purpose,

- Central Processing Unit (CPU): A processor designed to solve every computational problem in a general fashion. The cache and memory design is designed to be optimal for any general programming problem.
- Graphics Processing Unit (GPU): A processor designed to accelerate the rendering of graphics.
- Tensor Processing Unit (TPU): A co-processor designed to accelerate deep learning tasks develop using TensorFlow (a programming framework); Compilers have not been developed for TPU which could be used for general purpose programming; hence, it requires significant effort to do general programming on TPU

## Usage

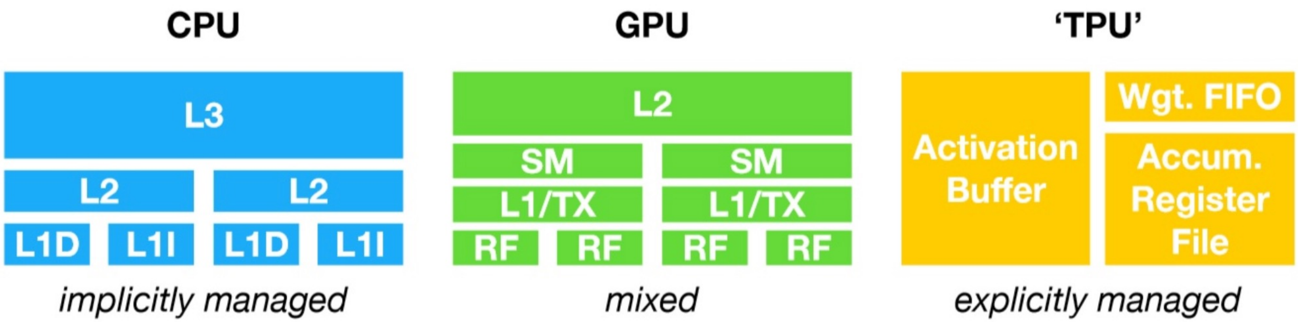
- Central Processing Unit (CPU): General purpose programming problem
- Graphics Processing Unit (GPU): Graphics rendering, Machine Learning model training and inference, efficient for programming problem with parallelization scope, General purpose programming problem
- Tensor Processing Unit (TPU): Machine Learning model (only in TensorFlow model) training and inference

## Manufacturers

- Central Processing Unit (CPU): Intel, AMD, Qualcomm, NVIDIA, IBM, Samsung, Hewlett-Packard, VIA, Atmel and many others
- Graphics Processing Unit (GPU): NVIDIA, AMD, Broadcom Limited, Imagination Technologies (PowerVR)
- Tensor Processing Unit (TPU): Google

# CPU, GPU, and TPU

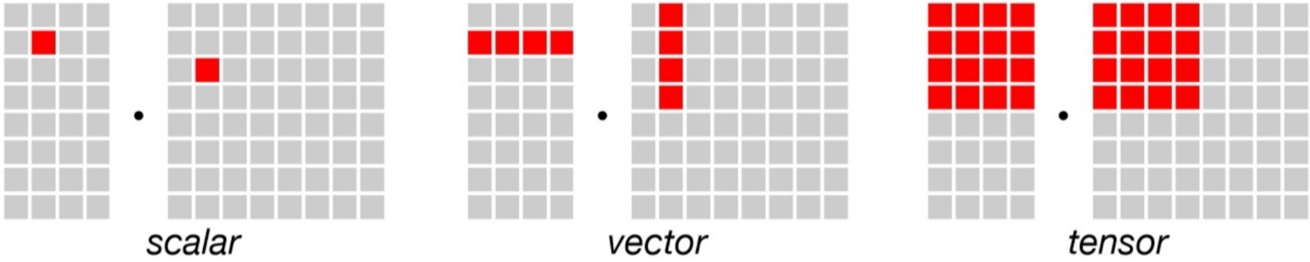
**Memory Subsystem Architecture**




**Compute Primitive**

This image summarizes the compute primitive (smallest unit) in CPU, GPU and TPU:

**Compute Primitive**




https://iq.opengenus.org/cpu-vs-gpu-vs-tpu/

# GPU Demo

https://colab.research.google.com/notebooks/gpu.ipynb