## Acknowledgement of Country

The National Computational Infrastructure acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

# Outline

**We will cover:**

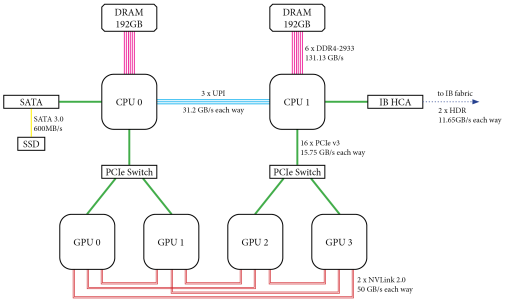
- Limitation of Python
- Numba
- CuPy
- Dask

# Cluster Support

- http://are.nci.org.au
- project: vp91
- modules: python3/3.8.5 cuda/12.0.0
- venv: /scratch/vp91/python_for_hpc/python3.8-venv

# Why do we need Parallelism?

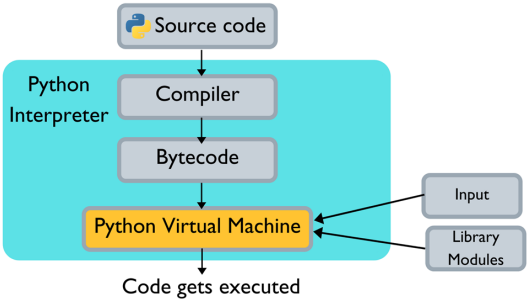
- Resource utilization is minimal:
  1. > 90% compute comes from GPU.
  2. CPU threads are under utilized.



## Limitation of

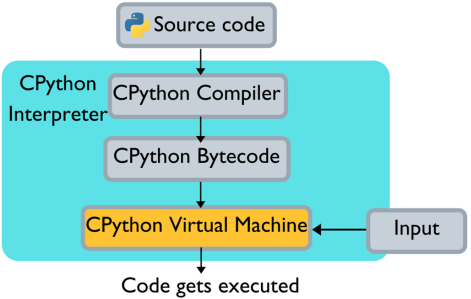
# Python under the hood

- Python is an interpreted language.
- Interpreter first compiles the source code into platform-independent Bytecode.
- Bytecode is executed by Python Virtual Machine.

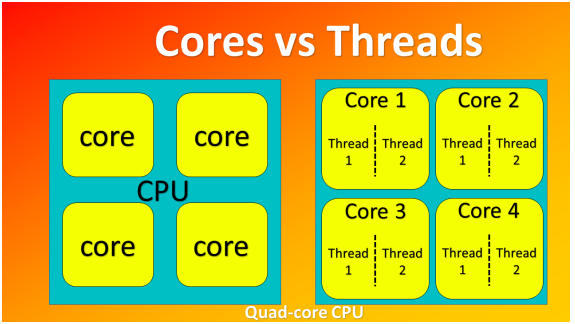


# Python under the hood

- CPython - implements the Python interpreter using both C and Python.
- standard reference implementation of Python.
- Not *thread-safe*

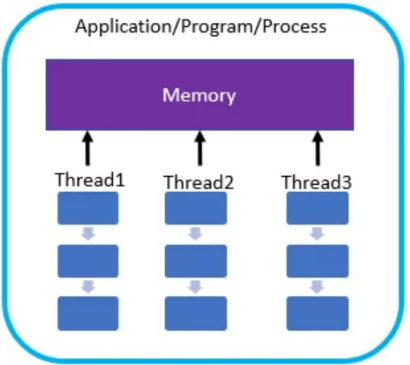


# Threads


Cores vs Threads
core
core
CPU
core
core
Core 1
Thread 1
Thread 2
Core 2
Thread 1
Thread 2
Core 3
Thread 1
Thread 2
Core 4
Thread 1
Thread 2
Quad-core CPU

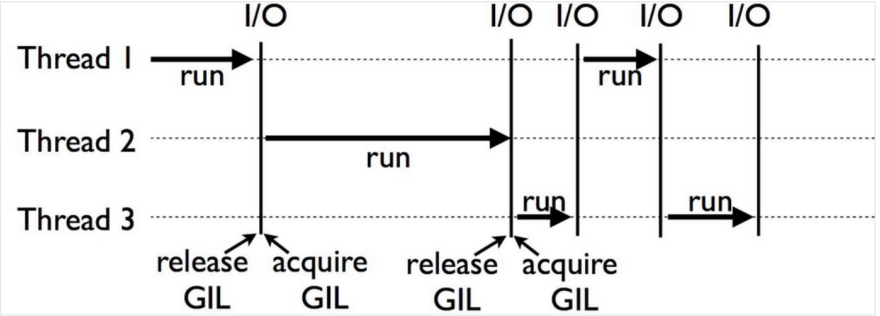
# Python's `threading` Module

- Memory is shared between multiple threads within a process
- Single instruction, multiple data (SIMD)



# Global Interpreter Lock (GIL)?

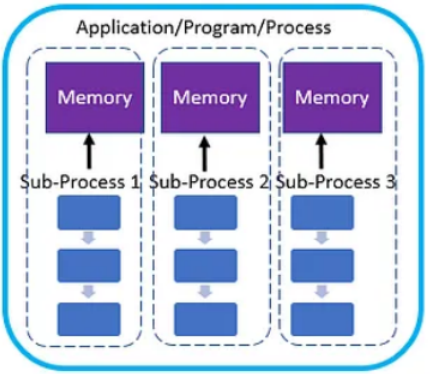
- Prevent multiple native threads from causing unwanted interactions
- Only one thread to execute the Bytecode at any given time

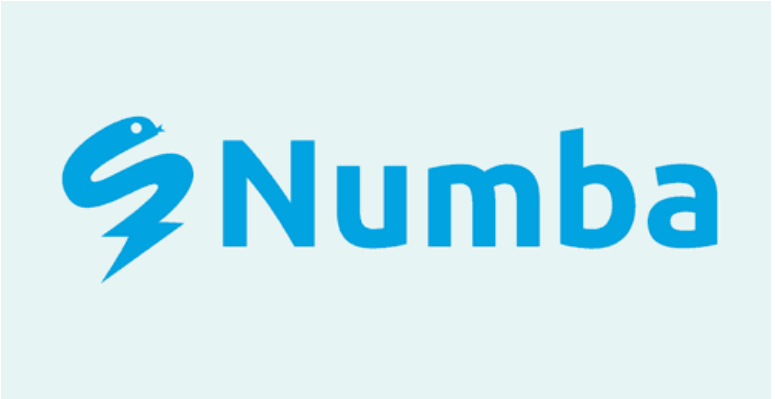


# Python's `multiprocessing` Module

- Spawn multiple native sub-processes within a program
- Each sub-process is allocated its own memory

- Each process can run on different CPU cores
- GIL is not involved
- Resource intensive when compared to threads

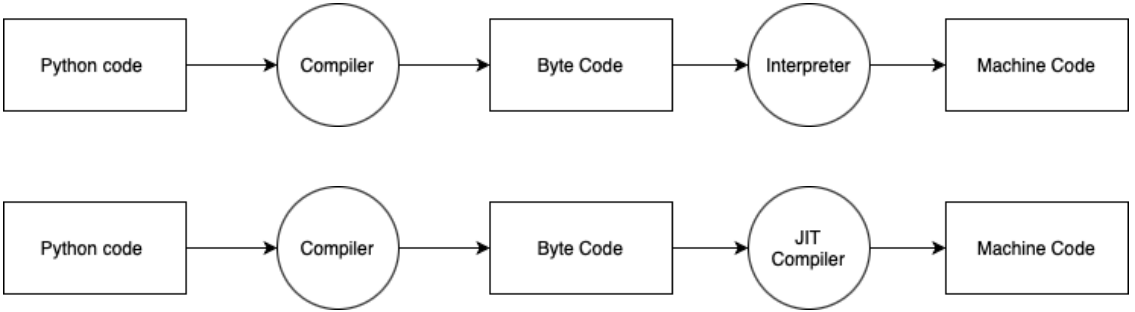



Numba

# Background

1. Optimized the inefficient use-cases of Numpy
2. Multi-dimensional array (ndarray)
3. Custom Python C extensions not required

# JIT


Python code
Compiler
Byte Code
Interpreter
Machine Code
Python code
Compiler
Byte Code
JIT
Compiler
Machine Code

# Performance

| Matrix Size | Numba | C |
|---|---|---|
| 64 x 64 | 463x | 453x |
| 128 x 128 | 454x | 407x |
| 256 x 256 | 280x | 263x |
| 512 x 512 | 276x | 268x |

Lam, Siu Kwan et al. “Numba: a LLVM-based Python JIT compiler.” LLVM ’15 (2015).

# Decorators

```
def uppercase_decorator(function):
  def wrapper():
      func = function()
      make_uppercase = func.upper()
      return make_uppercase

  return wrapper
```

# Decorators

```
def say_hi():
    return 'hello there'

decorate = uppercase_decorator(say_hi)
decorate()
```

```
@uppercase_decorator
def say_hi():
    return 'hello there'

say_hi()
```

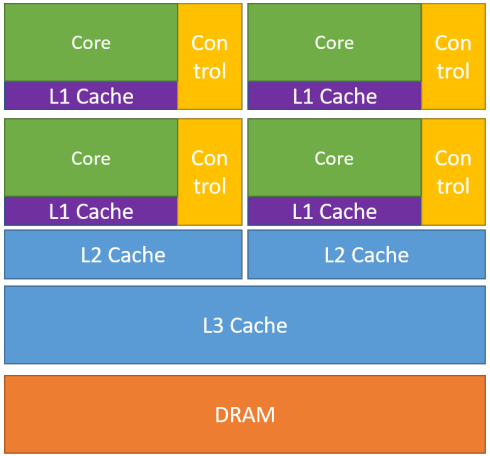
CuPy

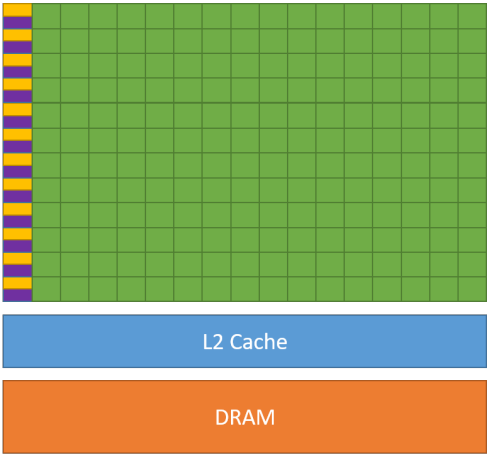
# CPU and GPU

1. CPU
   - Optimized to execute a code as fast as possible
   - Executes a few tens of threads in parallel
   - Transistors are give proportional importance control flow, computation and data caching
2. GPU
   - Optimized to execute a code as parallel as possible
   - Executes a few thousand of threads in parallel
   - Transistors are disproportional favour computation

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

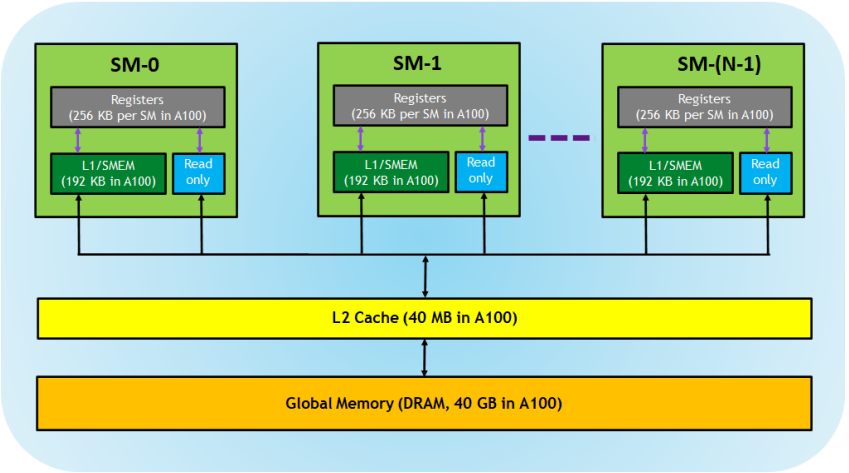
# CPU and GPU




https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

# GPU Architecture




https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

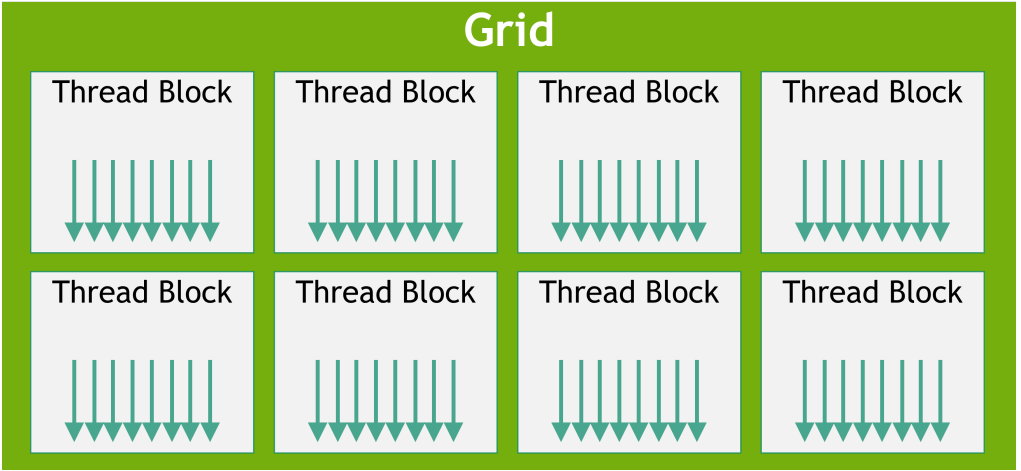
# GPU Workflow

- Allocate memory in GPU memory
- Move data from main memory to GPU memory
- Launch GPU kernel
- Move data back to main memory

# Example Program

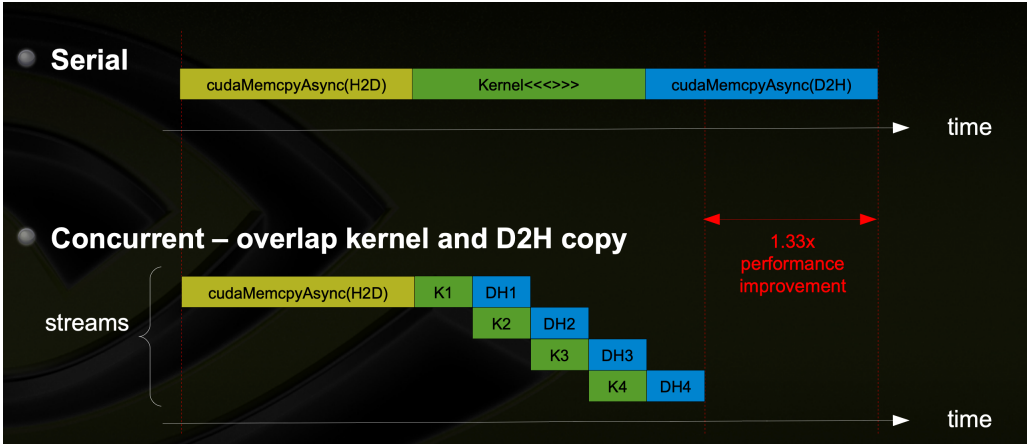
```
// Allocate Memory
cudaMalloc(&d_x, N*sizeof(float));

// Move data to GPU
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);

// Launch kernel
increment<<<(N+511)/512, 512>>>(N, d_x);

// Move data back
cudaMemcpy(x, d_x, N*sizeof(float), cudaMemcpyDeviceToHost)
```

# Blocks and Threads




https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

# CUDA Streams




https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf


dask

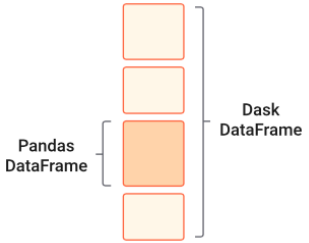
# Dask

- Parallel and distributed computing library for python
- Dask scale up to your full laptop capacity and out to a cloud cluster
- Multi-core and distributed+parallel execution on larger-than-memory datasets

# Dask Collection

- **High-level collections**: Mimic NumPy, lists, and pandas but can operate in parallel on datasets that don't fit into memory
  - Array
  - DataFrame
  - Bag
- **Low-level collections**: Give finer control to build custom parallel and distributed computations.
  - Delayed
  - Futures

# Dask Dataframe

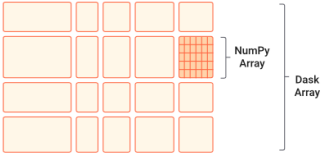



- One Dask DataFrame is comprised of many in-memory pandas DataFrames separated along the index
- One operation on a Dask DataFrame triggers many pandas operations on the constituent pandas DataFrames
- These operations are mindful of potential parallelism and memory constraints

# Lazy Evaluation

- Dask constructs the logic (called task graph) of your computation immediately
- Evaluates them only when necessary

# Dask Arrays




- Dask Array implements a subset of the NumPy ndarray interface using **blocked** algorithms
- Large array is cut into many small arrays
- Large computations are performed by combining many smaller computations

# Dask Delayed Decorator

- A Block of code can have operations that can happen in parallel
- Normally in python this would happen sequentially or the user will identify the parallel section and write parallel codes
- The Dask **delayed** function decorates your functions so that they operate lazily
- Dask will defer execution of the function, placing the function and its arguments into a task graph
- Dask will then identify opportunities for parallelism in the task graph
- The Dask schedulers will exploit this parallelism, generally improving performance

# Dask Future

- We can submit individual functions for evaluation
- The call returns immediately, giving one or more future
  - whose status begins as “pending”
  - later becomes “finished”
- There is no **blocking** of the local Python session.
- Difference between futures and delayed
  - delayed is lazy (it just constructs a graph)
  - futures are eager
- With futures, as soon as the inputs are available and there is compute available, the computation starts