Python in HPC

Joseph John

National Computational Infrastructure, Australia

Outline

We will cover:

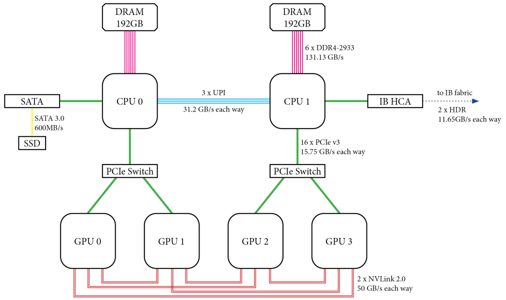
- Limitation of Python
- Numba
- CuPy
- Dask

Cluster Support

- **project**: vp91
- **modules**: python3/3.9.2 cuda/12.0.0
- **venv**: dask-python3.9-venv
- **repo**: https://github.com/nci900/AAPP-Parallel-Python

Why do we need Parallelism?

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

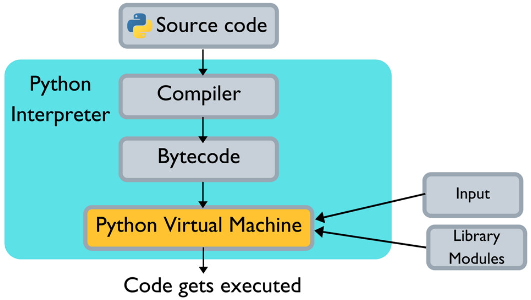


Limitation of

python™

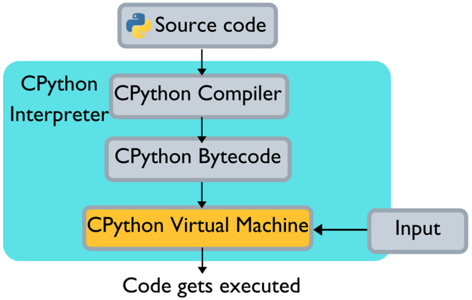
Python under the hood

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

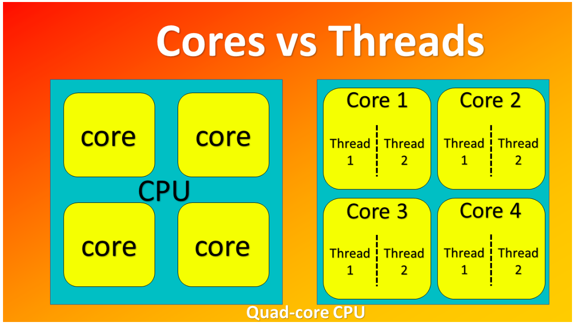


Python under the hood

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

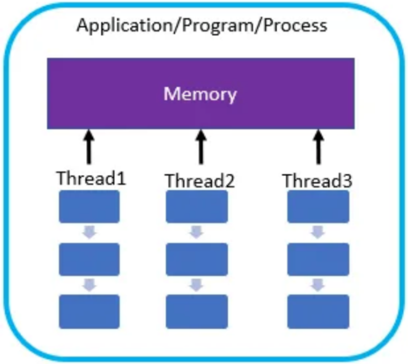


Threads


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

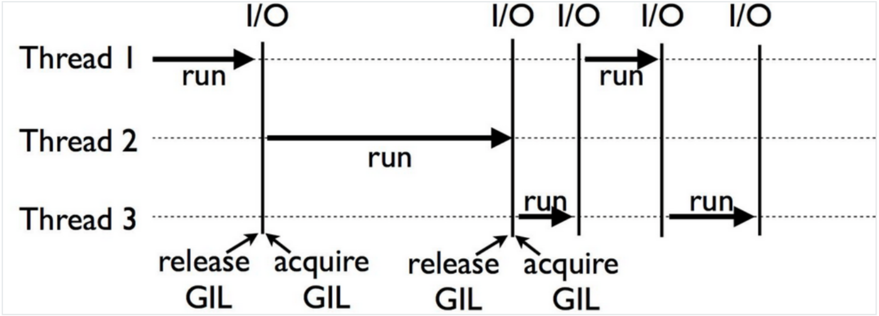
Python's `threading` Module

- 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 can access Python interpreter at any given time.
- Only one thread to execute the Bytecode at any given time.



Why do we need GIL?

- Python interpreter is not thread safe.
- Variables are managed a by a refcount.
- Without GIL there can be race conditions.

```
>>> import sys
>>> a = []
>>> b = a
>>> sys.getrefcount(a)
3
```

Race Condition

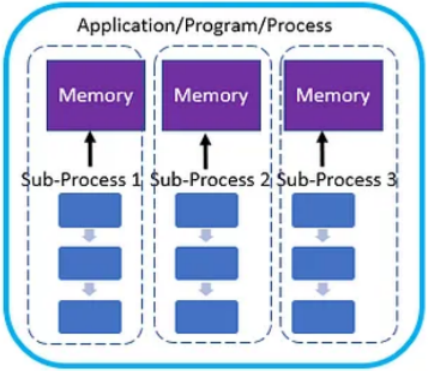
- **Critical section**: Section of code that accesses shared resources (variables or data structures).
- When multiple threads tries to write to a resource in a critical section it can result in unintended values.
- This is called race condition.
- Reading concurrently does not result in a race condition.

Mutex

- We prevent race condition using **mutex**.
- Mutex is a synchronization primitive that grants exclusive access to the shared resource to only one thread.
- Only one thread can acquire a mutex at a time.
- Only the thread that acquired the mutex can enter critical section.
- The thread that have access to the mutex should release it (after the critical section), for other threads to acquire it.

Python's `multiprocessing` Module

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



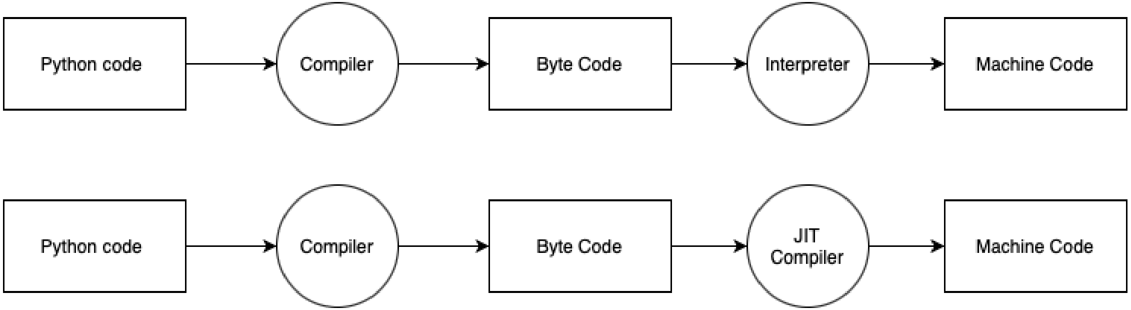
- Each process can run on different CPU cores.
- GIL is not involved.
- Resource intensive when compared to threads.

Numba

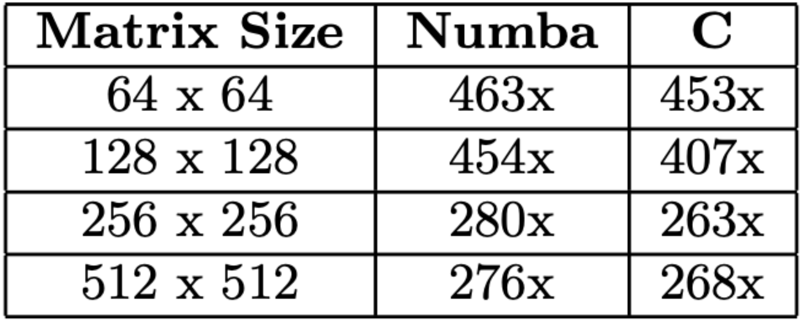
Background

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()
```

JIT

```python
@jit
def f(x, y):
    return x + y
```

- Decorating the function with @jit will mark a function for optimization by Numba’s JIT compiler
- The compilation will be deferred until the first function execution
- Different function invocation will result in different compilation

JIT

```python
@jit(int32(int32, int32))
def f(x, y):
    return x + y
```

- We can tell numba to generated code only for one set of arguments

JIT

```
@jit
def f(x, y):
    return x + y
```

- Decorating the function with @jit will mark a function for optimization by Numba's JIT compiler
- The compilation will be deferred until the first function execution
- Different function invocation will result in different compilation

JIT

```
@jit
def square(x):
    return x ** 2

@jit
def hypot(x, y):
    return math.sqrt(square(x) + square(y))
```

- One compiled function can call another compiled function.

nopython Mode

```
x = np.arange(100).reshape(10, 10)

@jit(nopython=True)
def with_numba(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
        return a + trace
```

- @jit decorator fundamentally operates in two compilation modes, nopython mode and object mode.
- *nopython* compilation mode compile the decorated without the involvement of the Python interpreter.

nogil Mode

```
@jit(nogil=True)
def f(x, y):
    return x + y
```

- Release GIL.
- Runs concurrently with other threads executing Python or Numba code.
- Takes advantage of multi-core systems.

cache Mode

```
@jit(cache=True)
def f(x, y):
    return x + y
```

- The chances are you call the same function again and again with the same argument type.
- Numba can cache the compiled code.

Automatic Parallelization

```
@jit(nopython=True, parallel=True)
def f(x, y):
    return x + y
```

- This feature only works on CPUs.

ufunc and Numba

```python
@vectorize([float64(float64, float64)])
def sinacosb_vect(a, b):
    return math.sin(a) * math.cos(b)
```

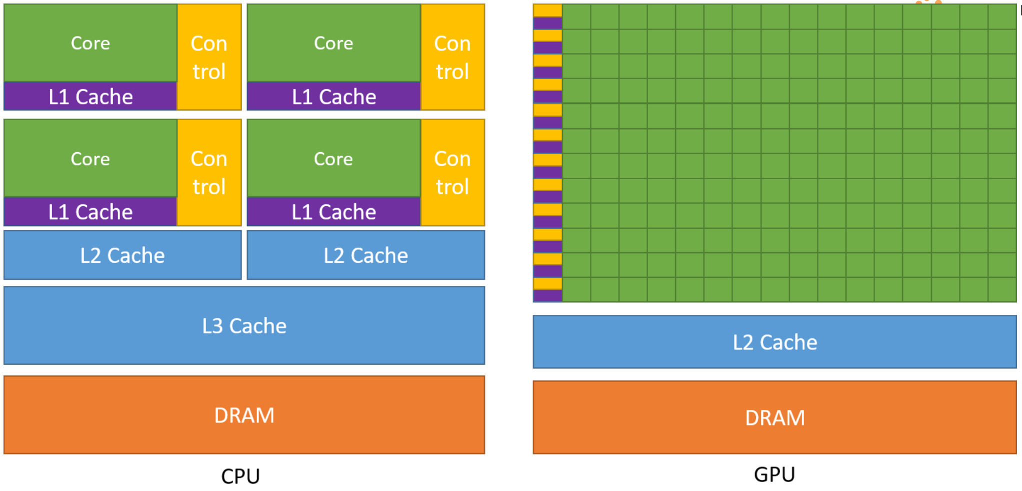
- Creating a ufunc that operates on a ndarray of a particular type is not straight forward.
- Numba makes this process easy.

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

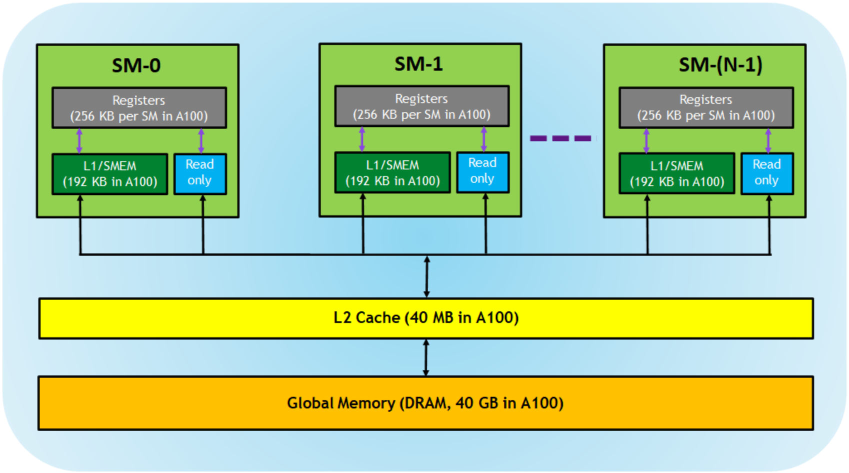
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

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/

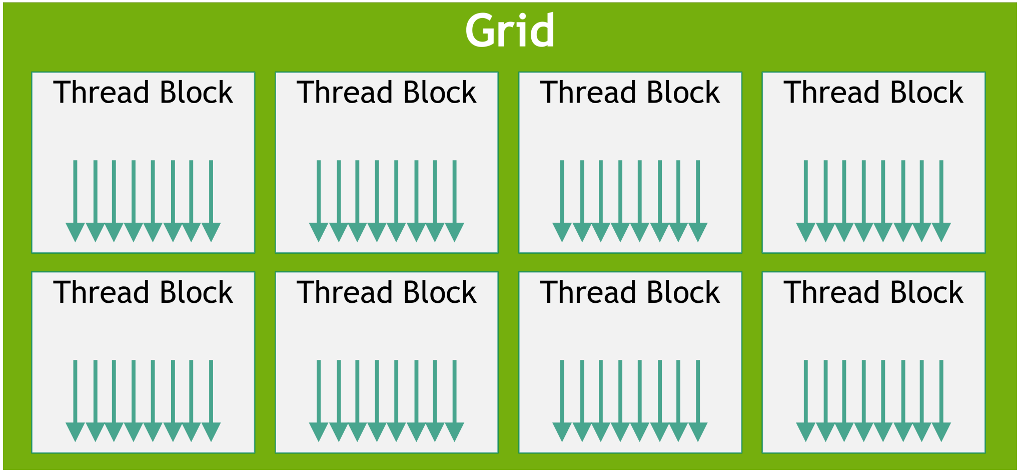
GPU Workflow

- 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 and Numba

```python
@vectorize(['int64(int64, int64)'], target='cuda')
def add_ufunc(x, y):
    return x + y
```

CUDA and Numba

Numba automates the following:

- Allcated GPU memory.
- Copy data to the GPU memory.
- Executed the CUDA kernel with the *correct kernel dimensions given the input sizes*.
- Copy data to the hist memory.
- Return the result as a NumPy array.

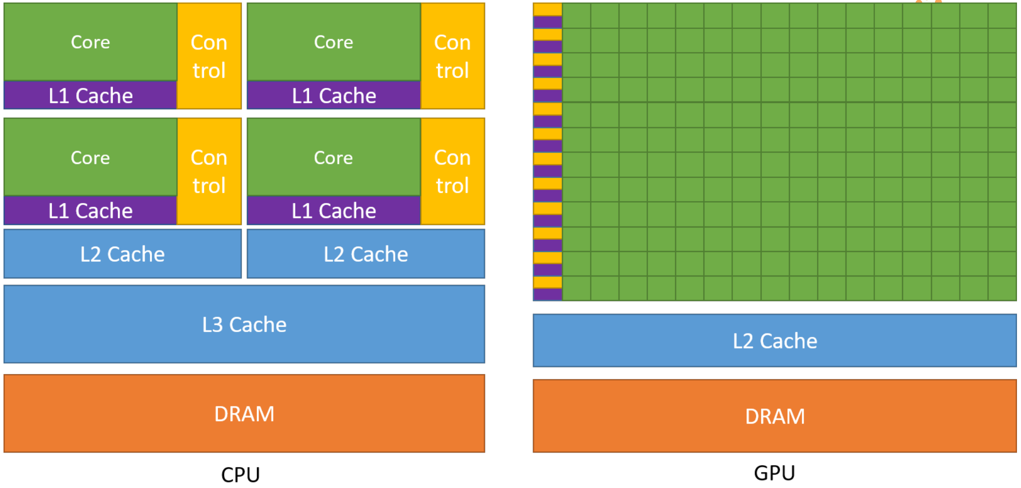
CuPy

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

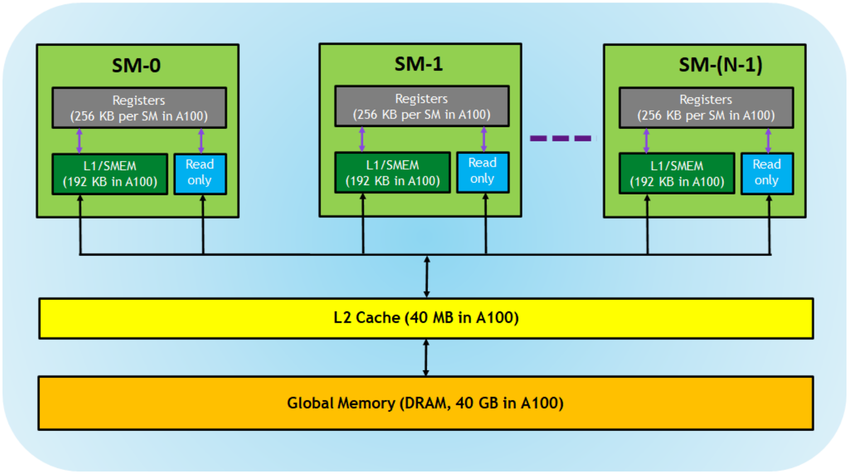
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

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/

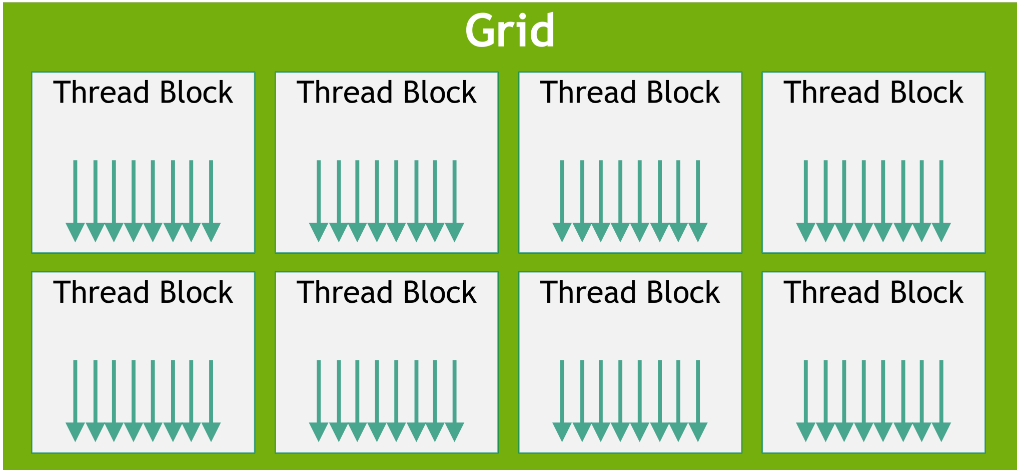
GPU Workflow

- 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

Current Device

```
cp.cuda.runtime.getDevice()
```

- CuPy has a concept of a current device.
- Default GPU device on which on which all operation of related to CuPy takes place.
- Unless specifically mentioned, all operation taskes place in this default device.

Allocate GPU memory

```
x_gpu = cp.array([1, 2, 3])
```

- cupy.array() allocates the data in the GPU memory.
- If no device is specified the memory gets allocated in the current device.

Switch GPU

```
with cp.cuda.Device(1):
    x_on_gpu1 = cp.array([1, 2, 3, 4, 5])
```

- We can use the device context manage to switch between the devices.

H2D data movement

```
x_cpu = np.array([1, 2, 3])
x_gpu_0 = cp.asarray(x_cpu)
```

- In CuPY the memory allocation and data movement can be done in a single operation.

D2D data movement

```
with cp.cuda.Device(1):
    x_gpu_1 = cp.asarray(x_gpu_0)
```

- D2D transfer is faster than H2D transfer.

D2H data movement

```
with cp.cuda.Device(0):
    x_cpu = cp.asnumpy(x_gpu_0)

with cp.cuda.Device(1):
    x_cpu = x_gpu_1.get()
```

- There are two ways to fetch the data from GPU:
  - cupy.ndarray.get()
  - cupy.asnumpy()

Device agnostic codes

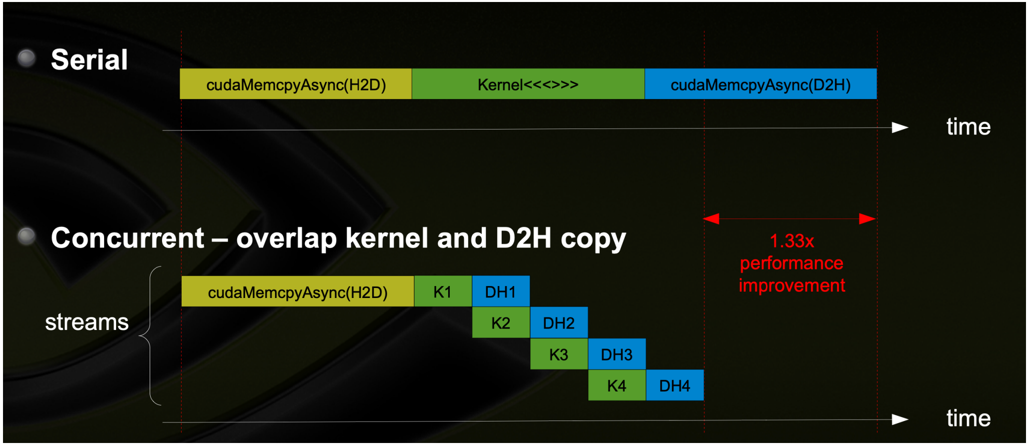
```
def log_array(x):
    xp = cp.get_array_module(x)
    xp.log1p(xp.exp(-abs(x)))

log_array(x_cpu)
log_array(x_gpu)
```

- We can make function calls to a data, without the knowledge of where they reside.

CUDA Kernels

- Elementwise Kernel.
- Reduction Kernel.
- Raw kernel.

CUDA Streams



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

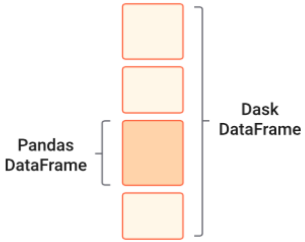
Dask

- 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

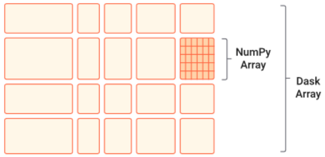


- One Dask DataFrame is comprised of many in-memory pandas DataFrames separated along the index
- One operation on a Dask DataFrame triggers many pandas operations on the constituent pandas DataFrames
- These operations are mindful of potential parallelism and memory constraints

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

Compute Vs Persist

- Dask executes the computations transformation to the distributed data.
- Compute: Converts it to a local object.
- Persist: The object remains distributed.