



## GPU acceleration with Python

Slide: [14/10 presentation](#)

Demo Numba: [Google Colaboratory](#)

Demo Dask & RAPIDS: [Google Colaboratory](#)

Boosting Python program's speed with GPU-Accelerated libraries, including Numba, Dask, and RAPIDS.

Table of content:

- [1. Challenges](#)
- [2. Solution: Numba for CUDA/GPU-acceleration](#)
- [3. Introduction: Parallel computing with GPU](#)
  - [3.1. GPU \(Graphic Processing Unit\)](#)
  - [3.2. How GPU works](#)
- [4. Introduction: CUDA](#)
- [5. Python parallel programming with CUDA/GPU](#)
  - [5.1. CUDA-supported GPU: Components & terms](#)
  - [5.2. CUDA/GPU-accelerated Python libraries](#)
- [6. Numba for Python](#)
  - [6.1. Terminology](#)
  - [6.2. Installation](#)
  - [6.3. Supported GPUs](#)
  - [6.4. Supported GPU memories](#)
  - [6.5. Demo](#)
  - [6.6. Performance comparison](#)
- [7. Dask & RAPIDS \(cudf, cuml\) libraries](#)
  - [7.1. Dask: Parallel task scheduler for Python](#)
  - [7.2. RAPIDS AI](#)
  - [7.3. Dask versus RAPIDS AI: Difference and Comparisons](#)
  - [7.4. Demo](#)
  - [7.5. Performance comparison](#)

### 1. Challenges

Runtime performance is always a critical problem for all Python programmers. Python is a high level programming language that is very slow compared to lower level language such as C/C++. Therefore, the computation speed of Python, especially for several mathematical tasks such as matrix multiplication, sorting, matrix decomposition, usually take time. To tackle this problem, using GPU is a great solution.

In their daily task, Python programmers may need to implement from scratch GPU-accelerated functions (which may not be in the scope of ML/DL) to speedup their program. However, building fast/speedup programs is usually the task for C/C++ programmers, and a pure Python programmer may find it hard and troublesome to do such task using Python.

In summary, there are 2 critical problems:

- Python is slow and Python programmers, in some cases, may need to find ways to speedup their program, one of the way is using GPU.
- Program speedup in Python is hard, since it's usually solved using C/C++.

## 2. Solution: Numba for CUDA/GPU-acceleration

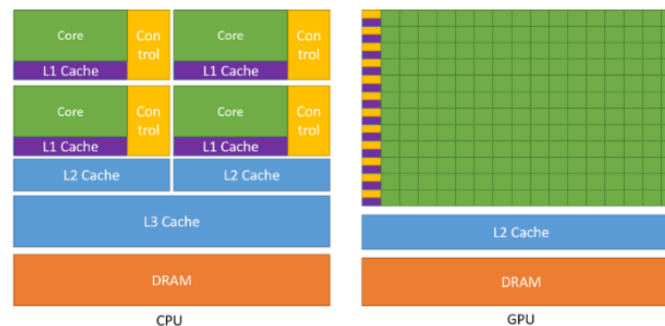
To tackle such challenges mentioned above, one can use GPU to speed-up their Python functions. Normally, when it comes to ML/DL tasks, many of the ML/DL libraries such as TensorFlow, Pytorch, etc., contain GPU-acceleration functions and params.

For tasks that need to build and implement Python programs from scratch, there are also several libraries that support GPU speed up, like Cython or Numba, which can help us to write fast code without going further than their Python knowledge.

## 3. Introduction: Parallel computing with GPU

### 3.1. GPU (Graphic Processing Unit)

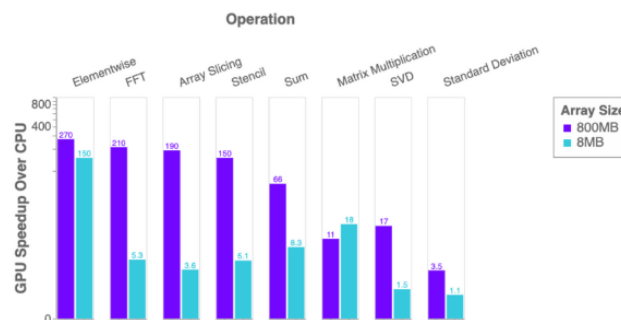
A hardware inside your computer that is initially created to handle computer graphic and video rendering. Basically, it tries to optimize the process by using parallel computing, the term in which your program is divided into multiple parts and the GPU handle all of the parts simultaneously.



Structure of GPU compares to CPU

While CPU only contains several cores, GPU contains from hundred to thousand cores, which are used to handle the task simultaneously. When applying GPU acceleration to Python, it is estimated to be 10-1000 times faster than pure Python code.

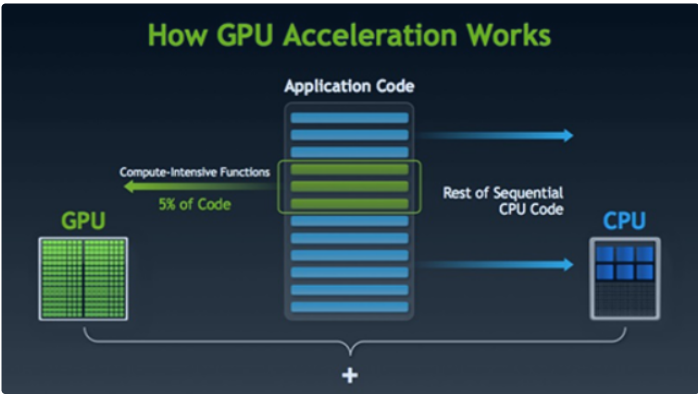
Nowadays, GPU has become one of the most important computation technology and has wide application. Over time, GPU became more flexible and programmable, enhancing their capabilities. This allowed graphics programmers to create more interesting visual effects and realistic scenes with advanced lighting and shadowing techniques. Other developers also began to tap the power of GPUs to dramatically accelerate additional workloads in high performance computing (HPC), deep learning.



GPU & CPU comparisons for some (mathematical) task

### 3.2. How GPU works

Basically, inside your code, together with sequential part, which must be executed one by one, there are several parts that can be parallelly computed. This is how GPU get into work to speedup your program.



How GPU Acceleration works

Comparison between CPU and GPU:

CPU	GPU
Central Processing Unit	Graphics Processing Unit
4-8 Cores	100s or 1000s of Cores
Low Latency	High Throughput
Good for Serial Processing	Good for Parallel Processing
Quickly Process Tasks That Require Interactivity	Breaks Jobs Into Separate Tasks To Process Simultaneously
Traditional Programming Are Written For CPU Sequential Execution	Requires Additional Software To Convert CPU Functions to GPU Functions for Parallel Execution

Characteristics of CPU versus GPU

#### 4. Introduction: CUDA

CUDA (2006 by Nvidia): general purpose **parallel computing platform** and **programming model** that **leverages the parallel compute engine in NVIDIA GPUs** to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C++ as a high-level programming language. CUDA is designed to support various languages and application programming interfaces.

As multicore CPUs and manycore GPUs are more and more popular nowadays, the problem is to develop an application software to scale up its parallelism to leverage the increasing number of processor cores (with widely varying numbers of cores).

CUDA programming model directs this problem by enabling the programmer to be easier to keep track in building and designing code that applies parallelism. Users are also using CUDA with familiar programming language such as C/C++.

CUDA has 3 key abstractions, which are a hierarchy of **thread groups**, **shared memories**, and **barrier synchronization** that are simply **exposed to the programmer as a minimal set of language extensions**. Generally, it enables programmers to breakdown their code into sub-parts and compute those sub-parts parallelly.



**Note:** A GPU is built around an array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

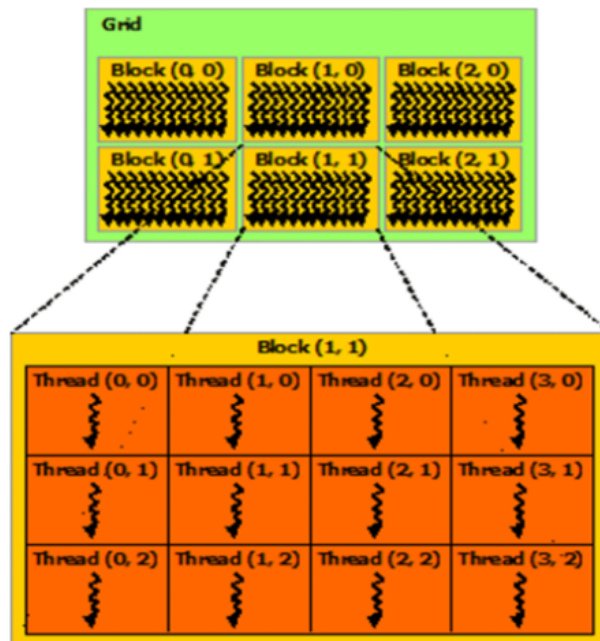
(4 SMs can parallelize into 4 lines of blocks and execute them simultaneously, meanwhile 2 SMs can only parallelize into 2 lines of blocks)

## 5. Python parallel programming with CUDA/GPU

First, let's understand several important components before we get our hands dirty!

### 5.1. CUDA-supported GPU: Components & terms

The components inside a CUDA-supported GPU are divided into 3 layers: Grids -> Blocks -> Threads.



Components of a CUDA-supported GPU





Inside a grid, there are blocks, and inside a block there are threads. The maximum number of threads per block is 1024, and the maximum number blocks per grid is 65535 ( $2^{16} - 1$ ) blocks in a single 1-dimensional grid.

The grid can be 1-dimensional, 2-dimensional, or 3-dimensional and is initialized based on the shape of the input data (for example, if the data is a 2D matrix, then we need to initialize 2-dimensional grid)

A GPU-accelerated function to be executed is called a kernel.

## 5.2. CUDA/GPU-accelerated Python libraries

There are several libraries support CUDA/GPU acceleration as follow:

<p><b>Compilation with Numba</b></p> <p>A Python to LLVM compiler</p> <p>JIT compiles numeric Python code to C speeds</p> <p><i>We can have for loops again!</i></p>		<p><b>Parallelism with Dask</b></p> <p>A Dynamic task scheduler</p> <p>Runs Python task graphs on distributed hardware</p> <p><i>MPI. But easier and slower Spark. But more flexible and without the JVM!</i></p>	
<p><b>GPUs - RAPIDS, CuPy</b></p> <p>CUDA-backed GPU libraries</p> <p>Like NumPy/Pandas/Scikit-Learn, but backed by CUDA code</p> <p><i>Python helped you to forget C Now you can forget CUDA too!</i></p>		<p><b>UCX</b></p> <p>High Performance Networking</p> <p>Provides interfaces and routing to high performance networking libraries like InfiniBand and NVLink</p> <p><i>Because once computation is fast we need to focus on everything else</i></p>	

## 6. Numba for Python

Using CUDA to speedup the program requires understanding of C/C++ and how CUDA works. However, many Python users who want to optimize their code & runtime may find it troublesome to learn and use C/C++ instead of their preferred language (Python).

Therefore, Numba is created as a solution direct to this problem. Numba is a friendly-to-use Python module for Python programmers to compile their code into CUDA and speedup their program. With Numba, Python users don't need to learn beyond their scope to apply CUDA and GPU-acceleration techniques into their daily tasks.

Numba can be applied for both CPU & GPU speed-up.

### 6.1. Terminology

There are several terms that we face a lot when we working with GPU-acceleration program, not only in Numba but also in TensorFlow or Pytorch.

Several important terms in the topic of CUDA programming are:

- *host*: the CPU
- *device*: the GPU
- *host memory*: the system main memory
- *device memory*: onboard memory on a GPU card
- *kernels*: a GPU function launched by the host and executed on the device
- *device function*: a GPU function executed on the device which can only be called from the device (i.e. from a kernel or another device function)

### 6.2. Installation

Simply type the command:

```
1 $ conda install cudatoolkit
```

or:

```
1 $ pip install cuda-python
```

### 6.3. Supported GPUs

Numba supports CUDA-enabled GPUs with Compute Capability 3.5 or greater. Support for devices with Compute Capability less than 5.3 is deprecated, and will be removed in a future Numba release.

Devices with Compute Capability 5.3 or greater include (but are not limited to):

- Embedded platforms: NVIDIA Jetson Nano, TX1, TX2, Xavier NX, AGX Xavier, AGX Orin.
- Desktop / Server GPUs: All GPUs with Pascal microarchitecture or later. E.g. GTX 10 / 16 series, RTX 20 / 30 series, Quadro P / V / RTX series, RTX A series, H100.
- Laptop GPUs: All GPUs with Pascal microarchitecture or later. E.g. MX series, Quadro P / T series (mobile), RTX 20 / 30 series (mobile), RTX A series (mobile).

## 6.4. Supported GPU memories

Numba supports 3 kinds of GPU memory:

- Global device memory (the large, relatively slow off-chip memory that's connected to the GPU itself).
- On-chip shared memory.
- Local memory.

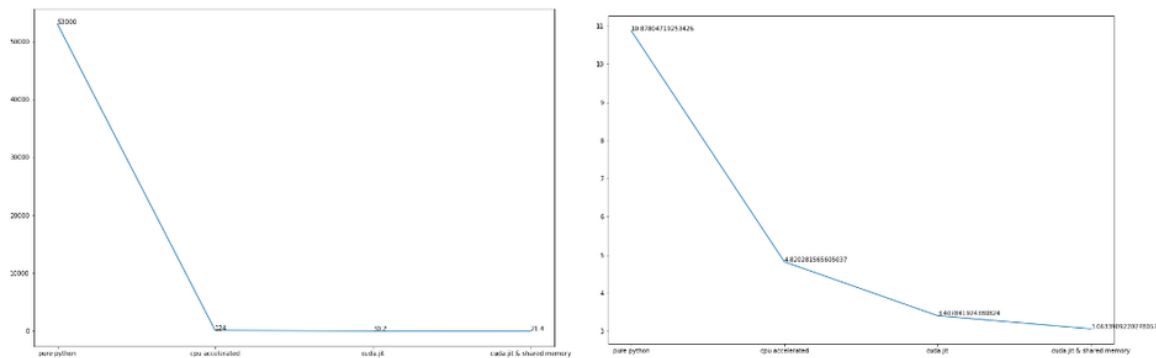
Initialization and assignment with **local memory** is faster than **device memory**.

## 6.5. Demo

Please go to [this Colab Notebook](#) for the demo code and instruction.

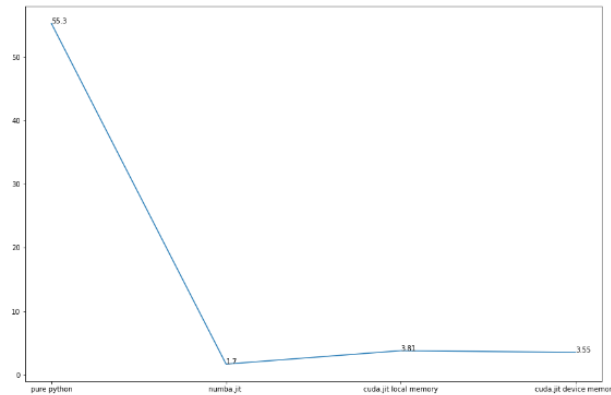
## 6.6. Performance comparison

Matrix multiplication:



Performance comparison between different implementations. The right image is just the log of the left image (for better visualization)

Quick-sort:



The performance of each implementation for quick-sort is:

- Pure Python: **55.3 ms**
- Python with CPU-accelerated: **1.7 ms**
- Cuda.jit using local memory with copy\_to\_host: **3.93 ms**
- Cuda.jit using device memory with copy\_to\_host: **3.55 ms**
- Cuda.jit using local memory: **331 µs**
- Cuda.jit using device memory: **83.8 µs**

## 7. Dask & RAPIDS (cudf, cuml) libraries

**Rising trends** for Python modules:

- Similar syntax & APIs for all Python modules.
- Programmers can change and work with different data-type from different libraries.
- Example: matmul with a numpy matrix and a pytorch tensor.

-> Dask & RAPIDS were built on top of Numpy, Pandas, Sklearn to speed-up the Python program by enabling parallelization.

### 7.1. Dask: Parallel task scheduler for Python

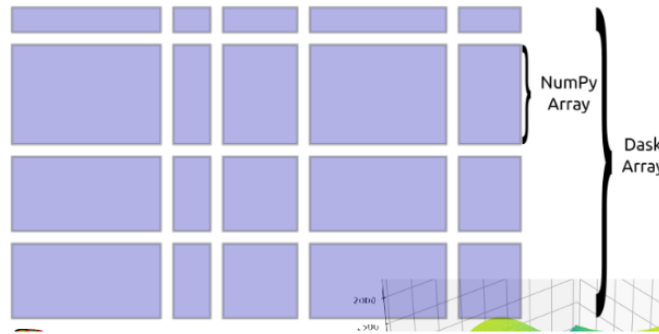
Dask enables parallel computing in Python.

- Focuses on **scaling out** and **deployability**.
- Can scale out to thousand-node clusters.
- Easy to install and use.
- Easy to deploy on Hadoop/Spark, Kubernetes, etc

Several important features of Dask is as follow:

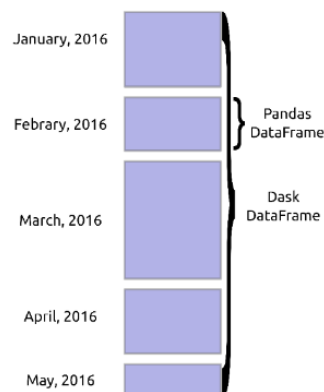
Parallel NumPy: Same API as Numpy. One Dask array is built from multiple Numpy arrays.

```
1 import dask.array as da
2 x = da.from_hdf5(...)
3 a = x + x.T - x.mean(axis = 0)
```



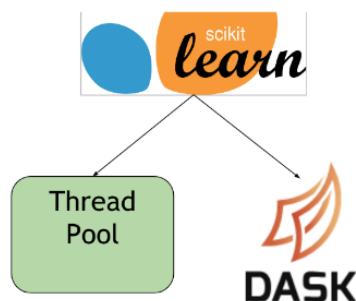
Parallel Pandas: Same API as Pandas. One Dask dataframe is built from multiple Pandas dataframe..

```
1 import dask.dataframe as dd
2 df = dd.read_csv()
3 df.groupby('name').balance.max()
```



Parallel Sklearn: Same API as Sklearn. Same code, just wrap with a decorator. Allowing scaling onto clusters. Available for most Sklearn where joblib is used.

```
1 from sklearn.externals import joblib
2 with joblib.parallel_backend('dask'):
3     estimator = RandomForest()
4     estimator.fit(X, y)
```



## 7.2. RAPIDS AI

RAPIDS is the GPU-accelerated to handle dataframe process and basic ML computation. RAPIDS has 2 popular libraries to handle GPU-acceleration:

- cuDF: similar to Pandas but can handle bigger datasets and run faster.



- cuML: similar to sklearn, with GPU accelerated.

Using cuDF and cuML can boost your training speed many times. However, such libraries still contain several drawbacks:

- Lack of necessary functions: Pandas and sklearn still have more tools, meanwhile cuDF does not have them and is still currently updating.
- Installing RAPIDS: If we want to install RAPIDS into our local computers, it is very likely that we will face version conflict (or other types of errors). Meanwhile, installing RAPIDS into Google Colabs or Kaggle also quite complicated and requires lots of code to make it work.

Example code of installing RAPIDS in Google Colab:

```

1 %%capture
2 # This get the RAPIDS-Colab install files and test check your GPU. Run this and the next cell only.
3 # Please read the output of this cell. If your Colab Instance is not RAPIDS compatible, it will warn you and gi
4 !git clone <https://github.com/rapidsai/rapidsai-csp-utils.git>
5 !python rapidsai-csp-utils/colab/env-check.py
6
7 # This will update the Colab environment and restart the kernel. Don't run the next cell until you see the sess
8 !bash rapidsai-csp-utils/colab/update_gcc.sh
9 import os
10 os._exit(00)
11
12 # This will install CondaColab. This will restart your kernel one last time. Run this cell by itself and only
13 import condacolab
14 condacolab.install()
15
16 # Import it again after restart kernel
17 import condacolab
18 condacolab.check()
19
20 # Installing RAPIDS is now 'python rapidsai-csp-utils/colab/install_rapids.py <release> <packages>'
21 # The <release> options are 'stable' and 'nightly'. Leaving it blank or adding any other words will default to
22 !python rapidsai-csp-utils/colab/install_rapids.py stable
23 import os
24 os.environ['NUMBAPRO_NVVM'] = '/usr/local/cuda/nvvm/lib64/libnvvm.so'
25 os.environ['NUMBAPRO_LIBDEVICE'] = '/usr/local/cuda/nvvm/libdevice/'
26 os.environ['CONDA_PREFIX'] = '/usr/local'
27
28 ## Done installation
29

```

Example usage:

```

1 from cuml import KMeans
2 kmeans = KMeans(n_clusters=num_clusters, init='k-means++', max_iter=5000, n_init=1, random_state=42)
3 kmeans.fit(X_vec)
4

```

As you may see, the code for cuDF and cuML is exactly the same as when you use Pandas or sklearn.

### 7.3. Dask versus RAPIDS AI: Difference and Comparisons

We can see through above, Dask and RAPIDS look like they all work on the same task. Then, what are their differences?

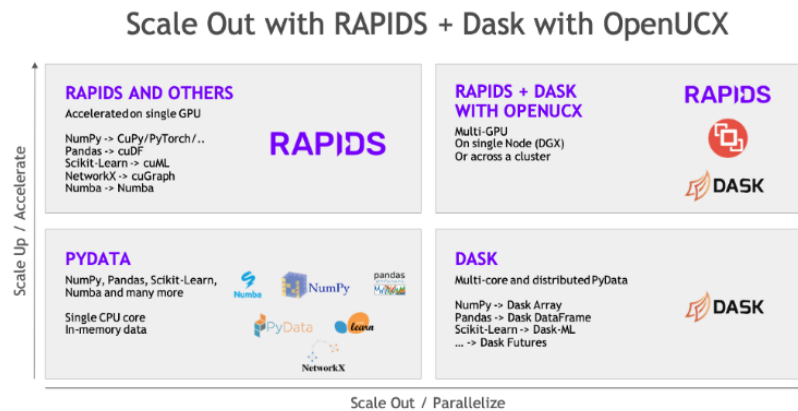
Dask:

- Dask allows you to scale out your workload onto multiple processors and multiple machines.
- Dask creates an abstract representation of your code and then distributing that onto a cluster.

RAPIDS:

- RAPIDS allows you to scale up your workload by reimplementing low-level parts of common open source APIs.
- Run RAPIDS on NVIDIA GPUs giving you faster execution.

We can utilize the advantages from both libraries:



## 7.4. Demo

Please go to [this Colab Notebook](#) for the demo code and instruction.

## 7.5. Performance comparison

Here are the results after performing data import, clustering, and K-selection method as below

Importing data (2m rows x 12 cols):

- Pandas: **18s**
- Dask (dask.dataframe): **88ms**
- RAPIDS (cudf): **78ms**

Perform clustering and K-selection (2m rows x 12 cols):

- Sklearn: **3min 24s**
- RAPIDS (cuml): **10.7s**