Computing for Economists

Parallelization & Big Data

Toren Fronsdal and Ruby Zhang

Harvard University

Table of Contents

Introduction

Introduction

In this lecture we will examine two closely-related topics:

- Writing **parallelized** code
- Working with **big data**

Parallelization:

⇒ Essential for some computationally intensive tasks, even with small datasets

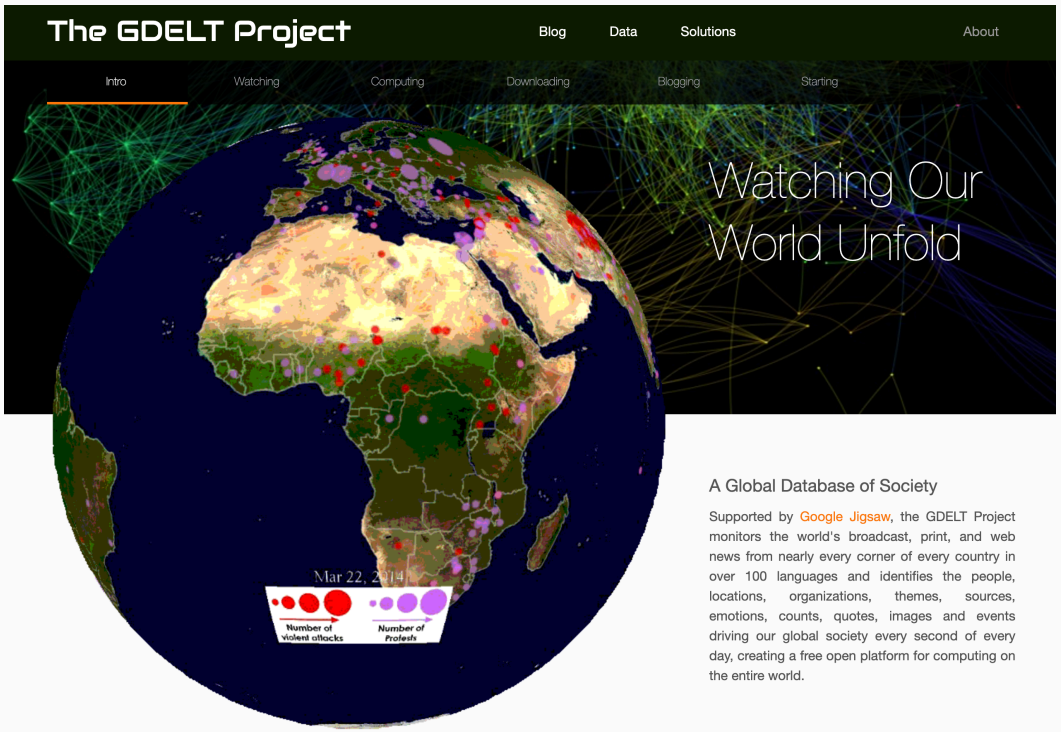
Working with Big Data:

⇒ Efficient storage and processing when working with data on a large scale

Motivating Example

The **GDELT 2.0 Event Database** captures information from the world's news media, including print, broadcast, and web.

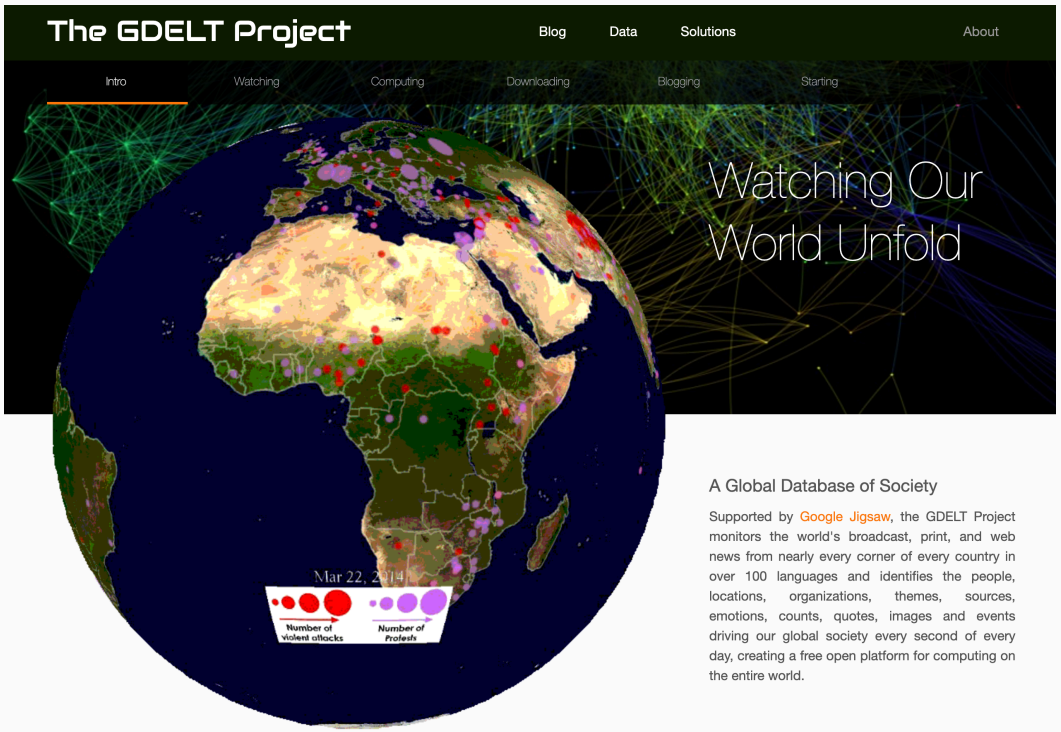
- Provides a global catalog of worldwide activities ("events") in over 300 categories from protests and military attacks to peace appeals and diplomatic exchanges



Motivating Example

The **GDELT 2.0 Event Database** captures information from the world's news media, including print, broadcast, and web.

- Provides a global catalog of worldwide activities ("events") in over 300 categories from protests and military attacks to peace appeals and diplomatic exchanges

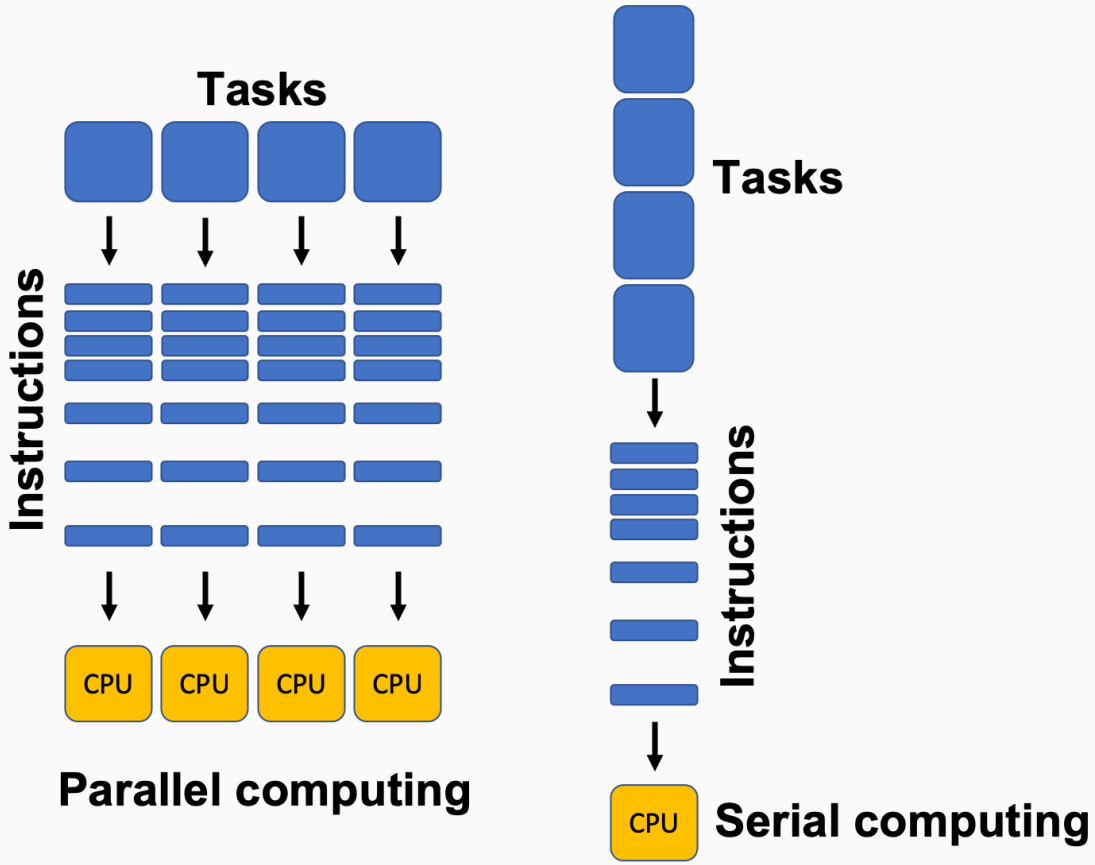


Number of rows	2,205,516,204
Total logical bytes	467.29 GB
Active logical bytes	467.29 GB
Long term logical bytes	0 B
Total physical bytes	129.7 GB
Active physical bytes	129.7 GB
Long term physical bytes	0 B
Time travel physical bytes	0 B

Parallelization

Parallelization

Parallelization: breaking down a task or a program into smaller, independent subtasks that can be executed simultaneously or in parallel by multiple processing units or threads.



Source: Python Numerical Methods

Embarassingly Parallel

Embarrassingly parallel programs refer to tasks or computations that can be easily divided into **independent**, **non-communicating** subtasks.

Examples:

- Monte Carlo integration
- Bootstrapping confidence intervals
- Grid search over hyperparameters

If serial steps in a program rely on the output of the previous step, then parallelization is not possible.

Example: Value Function Iteration

$$V(k) = \max_{k'} \{u(c) + \beta V(k')\}$$

$$c = k^{\alpha} + (1-\delta)k - k'$$

1. We have a grid of capital with 100 points, $k \in [k_1, k_2, \ldots, k_{100}]$.
2. We have a current guess $V^n(k)$.
3. We can send the problem:

$$\max_{k'} \{u(c) + \beta V^n(k')\}$$

$$c = k_1^{\alpha} + (1-\delta)k_1 - k'$$

to processor 1 to get $V^{n+1}(k_1)$.

4. We can send similar problem for each k to each processor.
5. When all processors are done, we gather the $V^{n+1}(k_1)$ back.

When to Not Parallelize

Amdahl's Law: the speedup of a program using multiple processors in parallel computing is limited by the *time needed for the sequential fraction of the program*.

Costs:

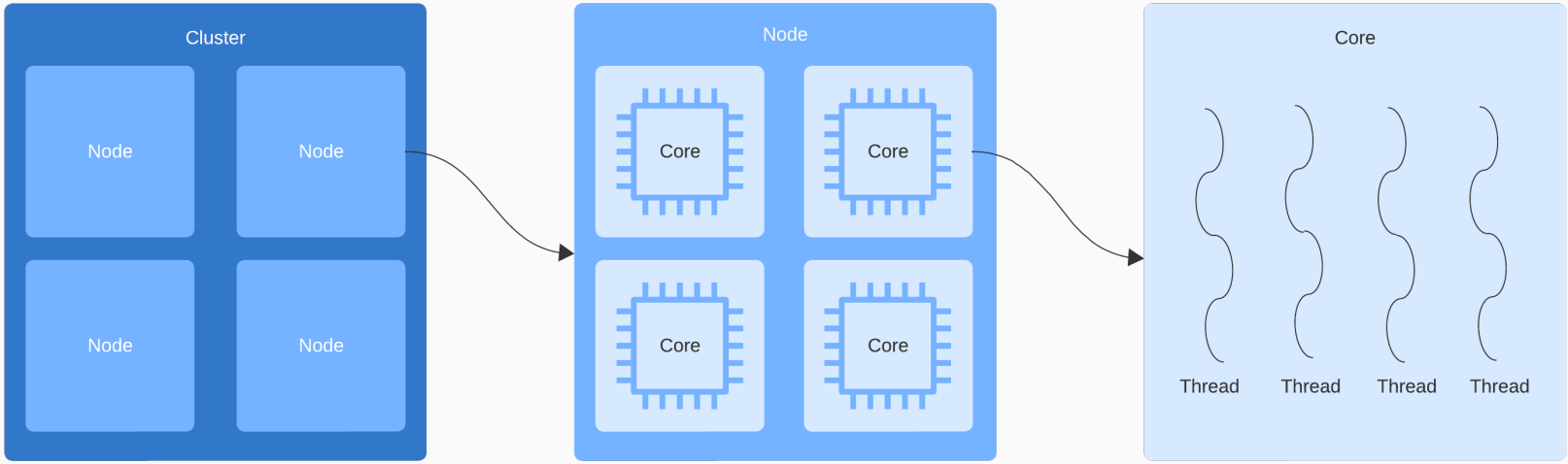
- Starting threads or processes/workers
- Transferring shared data to workers
- Synchronizing

$\Rightarrow$ **It is not always preferable to parallelize**:

- Small steps in the program are interspersed throughout tasks requiring communication
- A lot of communication needed between threads

For example, parallelizing an operation on each row of a dataset may slow down the execution if the data is not sufficiently large and the row operations are fast.

What to parallelize over?



- **Node**: individual computer or server within a cluster or network.
- **Core**: a physical processing unit within a CPU.
- **Thread**: the smallest unit of execution within a process.

Considerations: problem size, hardware architecture, communication overhead associated with parallelizing at different levels, memory constraints

Shared Memory vs. Distributed Memory

In **shared memory parallelization**, all processors have direct access to a common, global memory space.

- Each processor can read from and write to the same memory locations.
- Requires careful synchronization to avoid race conditions and ensure data consistency.
- Often within a single node, which can result in limited scalability.

In **distributed memory parallelization**, each processor has its own local memory, and there is no shared global memory.

- Processors communicate by passing messages between them.
- Scales well for large computational tasks distributed across multiple nodes.
- Requires explicit communication and synchronization, which can introduce complexity.

Parallelization Pitfalls

Race conditions occur when multiple threads or processes attempt to access shared data concurrently without proper synchronization. This can lead to unpredictable behavior and incorrect results.

Dead locks occur when A is waiting for B and B is waiting for A.

Identifying and managing **data dependencies** between parallel tasks can be challenging. Dependencies may introduce serialization points, limiting the benefits of parallelism.

Parallelization in Practice

Parallelization in Practice

Working on one computer

$\Rightarrow$ Parallelization across processors in Python

Distributed computing (e.g., computing clusters)

$\Rightarrow$ Parallelization across nodes in Linux clusters with Slurm

We will also see more options for distributed high-performance computing in the next section on working with big data.

Parallelization in Python

The *multiprocessing* module is a built-in Python library that provides support for creating and managing multiple processes, allowing concurrent execution of tasks on multi-core processors.

joblib offers tools for parallelizing Python functions using multiprocessing or threading.

dask provides a variety of tools for managing parallel computations across processes and across nodes (e.g. computing clusters).

Embarrassing Parallelization Across Processors in Python

multiprocessing.Pool:

```
from multiprocessing import Pool

def square(x):
  return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        ints = list(range(10))
        print(p.map(square, ints))
>> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

joblib.Parallel:

```
from joblib import Parallel, delayed

Parallel(n_jobs=2)(delayed(square)(i) for i in range(10))
>> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

multiprocessing

Process: represents an activity that is run in a separate process.

```
from multiprocessing import Process

def square(x):
  return x*x

def cube(x):
  return x*x*x

if __name__ == '__main__':
    # create processes and assign a function for each process
    process1 = Process(target=square, args=(2,))
    process2 = Process(target=cube, args=(2,))

    # start both processes
    process1.start()
    process2.start()

    # block the main program until these processes are finished
    process1.join()
    process2.join()
```

Multi-Processors and Memory

By default, separate processors do not share memory. Data can be stored in a shared memory variable using *`Value`* or *`Array`*.

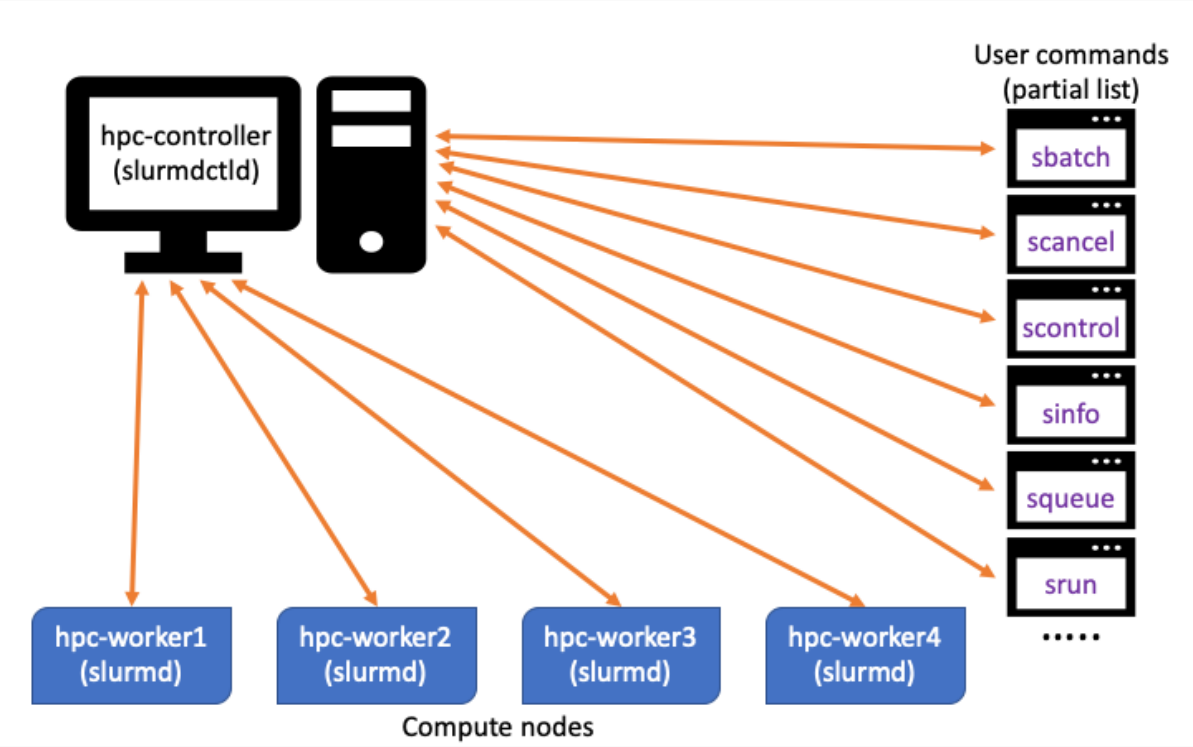
```
from multiprocessing import Process, current_process

def increment(x):
    for _ in range(3):
        x += 1
        print(f"Process {current_process().name}: {x}")

if __name__ == "__main__":
    x = 0
    # create two processes, each incrementing the variable
    process1 = Process(target=increment, args=(x,))
    process2 = Process(target=increment, args=(x,))
    # start both processes
    process1.start()
    process2.start()
    # wait for both processes to finish
    process1.join()
    process2.join()
    # print the final value of the variable
    print("Final value of the variable:", x)
>> Process Process-1: 1
>> Process Process-1: 2
>> Process Process-2: 1
>> Process Process-1: 3
>> Process Process-2: 2
>> Process Process-2: 3
>> Final value of the variable: 0
```

Not sharing memory

```
from multiprocessing import Process, current_process, Value

def increment(x):
    for _ in range(3):
        x.value += 1
        print(f"Process {current_process().name}: {x.value}")

if __name__ == "__main__":
    x = Value('i', 0)
    # create two processes, each incrementing the variable
    process1 = Process(target=increment, args=(x,))
    process2 = Process(target=increment, args=(x,))
    # start both processes
    process1.start()
    process2.start()
    # wait for both processes to finish
    process1.join()
    process2.join()
    # print the final value of the variable
    print("Final value of the variable:", x.value)
>> Process Process-1: 1
>> Process Process-1: 2
>> Process Process-2: 3
>> Process Process-1: 4
>> Process Process-2: 5
>> Process Process-2: 6
>> Final value of the variable: 6
```

Sharing memory

Parallelization with Cluster Computing



Slurm

Slurm is a common workload manager and job scheduler for cluster computing.

- The Harvard FASRC cluster uses Slurm to manage jobs
- Typically interact via the command line
- A number of ways to use Slurm for parallel computations

Slurm

Most common approach is to use a batch script to define a job and submit the job with the command *sbatch*. For example, we can create the shell script *runscript.sh*:

```
#!/bin/bash
#SBATCH -c 1                # Cores requested (equivalent --cpus-per-task)
#SBATCH -N 1                # Nodes requested
#SBATCH -t 0-00:10          # Max time requested in D-HH:MM
#SBATCH -p shared           # Partition to submit to
#SBATCH --mem=500M          # Memory pool for all cores
#SBATCH -o outfile_%j.out   # Send stdout to outfile (%j is replaced with the job ID)
#SBATCH -e errfile_%j.err   # Send stderr to errfile

# Load software
module load Anaconda/5.0.1-fasrc01

# Run python script
python script.py
```

and submit the job via the command line *sbatch runscript.sh*.

- *!/bin/bash* allows the script to be run as a bash script
- *#SBATCH* lines set parameters for the Slurm scheduler

Parallelization with Slurm - Requesting Resources

Suppose we write a program ***parallel.py*** containing a portion of code that parallelizes over the maximum number of cores available.

```
from joblib import Parallel, delayed
...
n_cores = int(os.environ['SLURM_CPUS_PER_TASK'])
Parallel(n_jobs=n_cores)(delayed(function)(i) for i in range(n))
```

We can then write a batch script with a command that contains the number of cores to make available:

```
#!/bin/bash
#SBATCH -c 16                # Cores requested (equivalent --cpus-per-task)
#SBATCH -N 1                 # Nodes requested
#SBATCH -t 0-10:00           # Max time requested in D-HH:MM
#SBATCH -p shared            # Partition to submit to
#SBATCH --mem=2000M          # Memory pool for all cores
#SBATCH -o outfile_%j.out    # Send stdout to outfile (%j is replaced with the job ID)
#SBATCH -e errfile_%j.err    # Send stderr to errfile

module load Anaconda/5.0.1-fasrc01
python parallel.py
```

Parallelization with Slurm - Using Job Arrays

Another approach is to use **job arrays** to parallelize by having Slurm run the same task on different nodes simultaneously (with potentially different inputs).

Suppose you have one data file for each year (*`data2010.csv`*, *`data2011.csv`*, etc.).

The Python script *`data_processing.py`* processes one input data file.

```
import sys
import pandas as pd

# sys.argv contains command line arguments passed into the script (argv[0] is the script name)
year = sys.argv[1]
df = pd.read_csv(f"../data/raw/data{year}.csv")
...
df_processed.to_csv(f"../data/processed/data{year}.csv")
```

Parallelization with Slurm - Using Job Arrays

The *--array* SBATCH command provides a way to submit similar jobs quickly and easily.

To process our datasets in parallel, we can specify the array with the command *--array=2010-2024*.

Each task has a unique environmental variable *$SLURM_ARRAY_TASK_ID*.

```
#!/bin/bash
#SBATCH -c 16               # Cores requested (equivalent --cpus-per-task)
#SBATCH -N 4                # Nodes requested
#SBATCH --array=2010-2024   # Create an array job with task IDs ranging from 2010 to 2024
#SBATCH -t 0-10:00          # Max time requested in D-HH:MM
#SBATCH -p shared           # Partition to submit to
#SBATCH --mem=2000M         # Memory pool for all cores
#SBATCH -o outfile_%j.out   # Send stdout to outfile (%j is replaced with the job ID)
#SBATCH -e errfile_%j.err   # Send stderr to errfile

module load Anaconda/5.0.1-fasrc01
python data_processing.py $SLURM_ARRAY_TASK_ID
```

Large-Scale Data Manipulation

Goals

- Understand key concepts and strategies for working with big data.

  $\Rightarrow$ Enduring approaches

- Provide a high-level overview of a smattering of tools for working with large data sets.

  $\Rightarrow$ Rapidly-evolving ecosystem

Working with Big Data

Data needs to be...

- Stored and accessed
  - $\Rightarrow$ Databases, distributed file systems, and data serialization
- Analyzed
  - $\Rightarrow$ Programming systems / models of computation

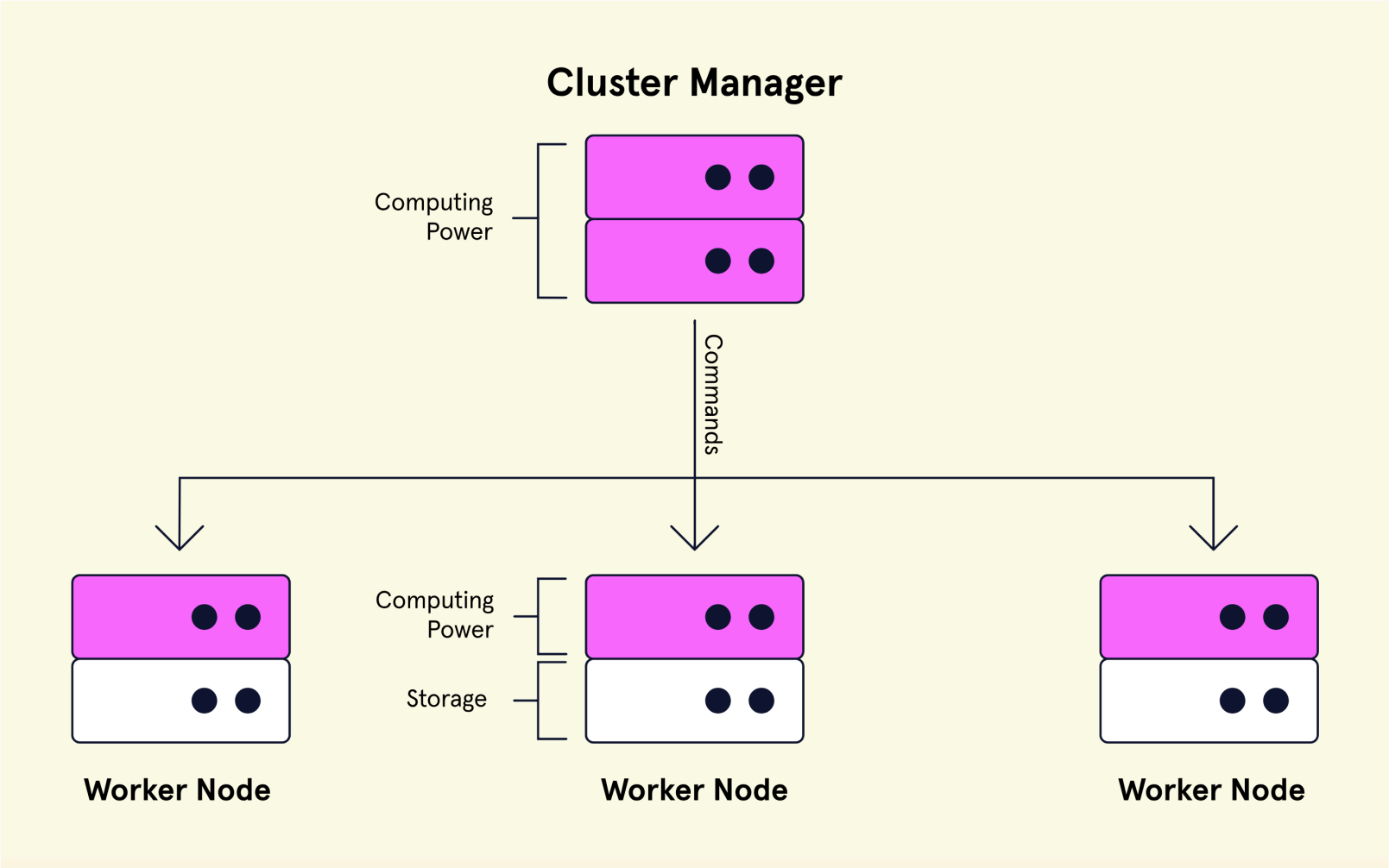
Data Storage and Databases

Economists primarily work with tabular data. This structured form of data is most commonly held in a **relational database management system (RDBMS)**.

More generally, structured and unstructured data can be held in **distributed file systems**, like the Apache Hadoop Distributed File System (HDFS).

Unstructured data (e.g., images) can be housed in **cloud object stores** like AWS S3, Azure Blob Storage, and Google Cloud Storage.

Distributed File System



Source: CodeAcademy

SQL and NoSQL

SQL (Structured Query Language) and NoSQL (Not Only SQL) are two distinct approaches to database management.

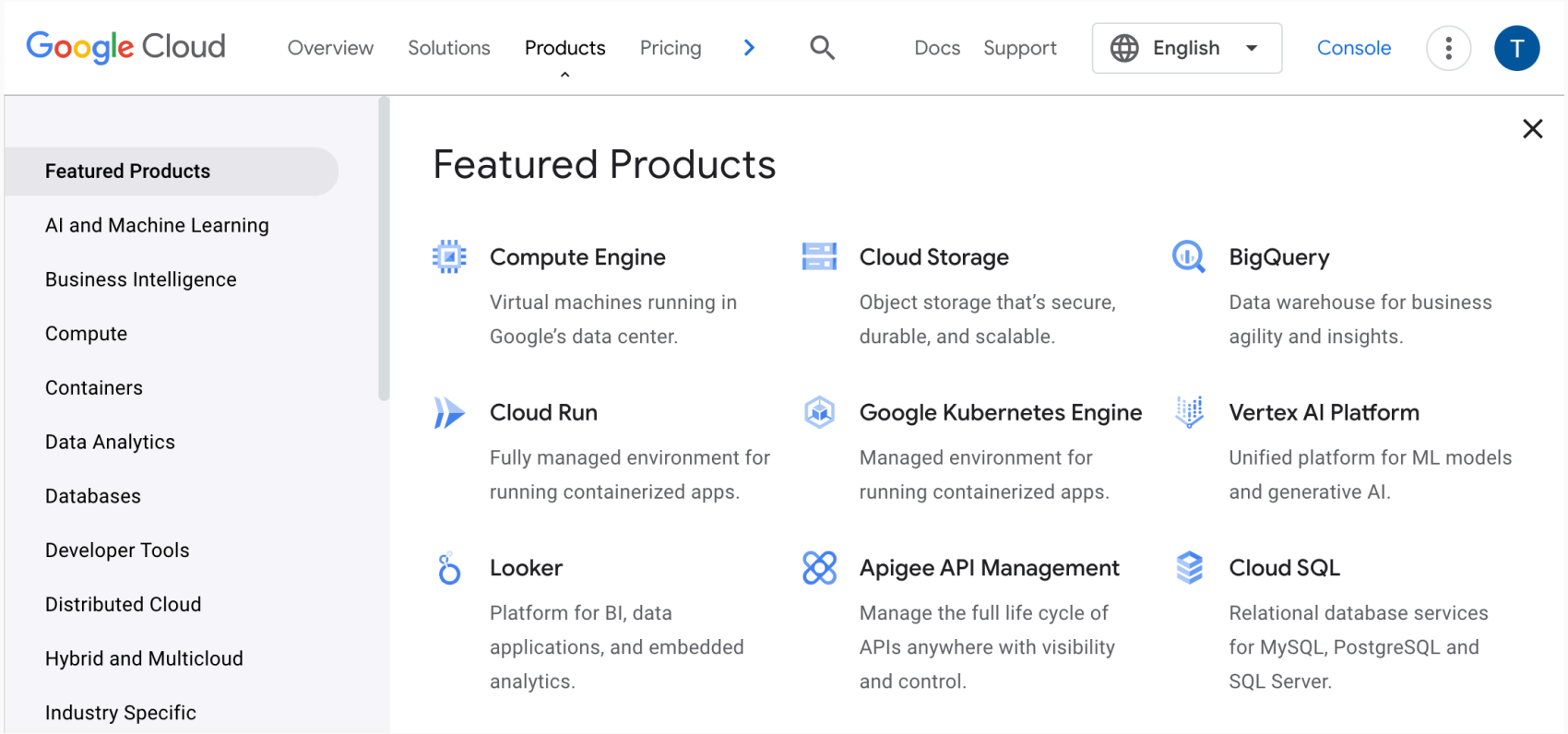
SQL

- Relational database
- Structured schema
- Standard for tabular data
- Ex: MySQL, SQL Server, PostgreSQL

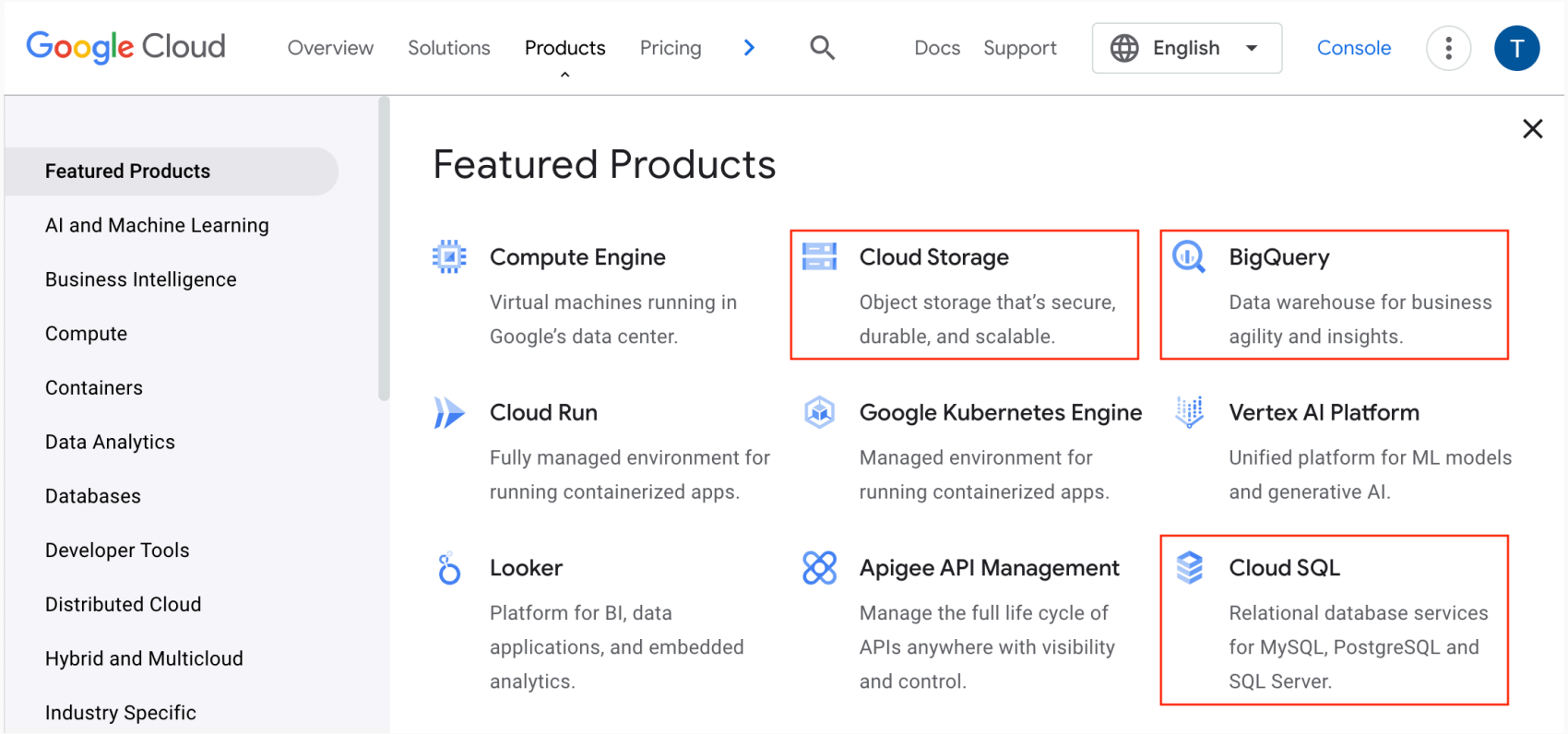
NoSQL

- Encompass various data models, including document-oriented, key-value, column-family, and graph databases
- Schema-less or schema-flexible
- Ex: Amazon DynamoDB, Google Bigtable, Cassandra

Data in the Cloud Example: Google Cloud

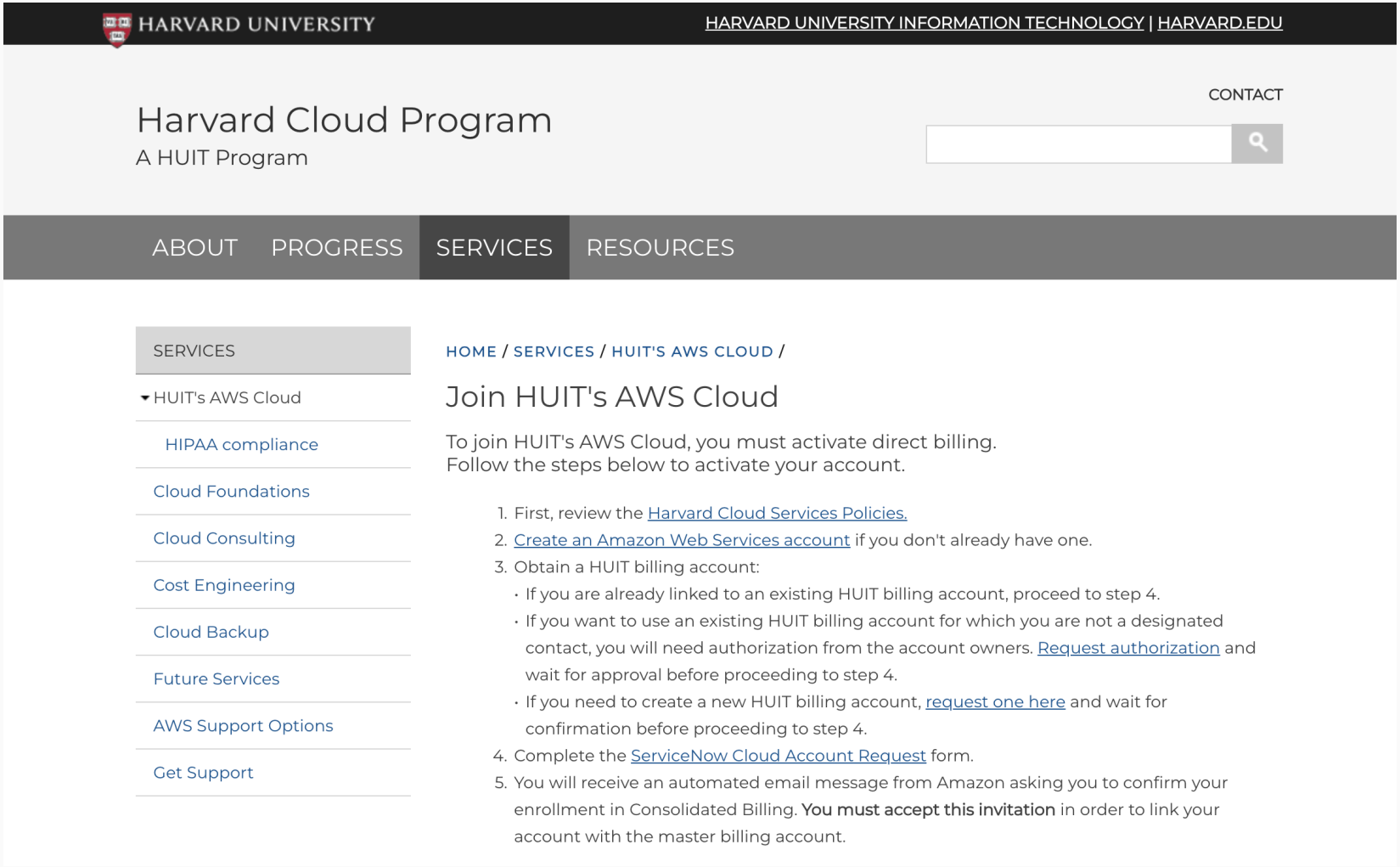


Data in the Cloud Example: Google Cloud



Storage and Compute Options at Harvard

Storage and Compute Options at Harvard



HARVARD UNIVERSITY

HARVARD UNIVERSITY INFORMATION TECHNOLOGY | HARVARD.EDU

Harvard Cloud Program

A HUIT Program

CONTACT

ABOUT PROGRESS SERVICES RESOURCES

SERVICES

- HUIT's AWS Cloud
  - HIPAA compliance
- Cloud Foundations
- Cloud Consulting
- Cost Engineering
- Cloud Backup
- Future Services
- AWS Support Options
- Get Support

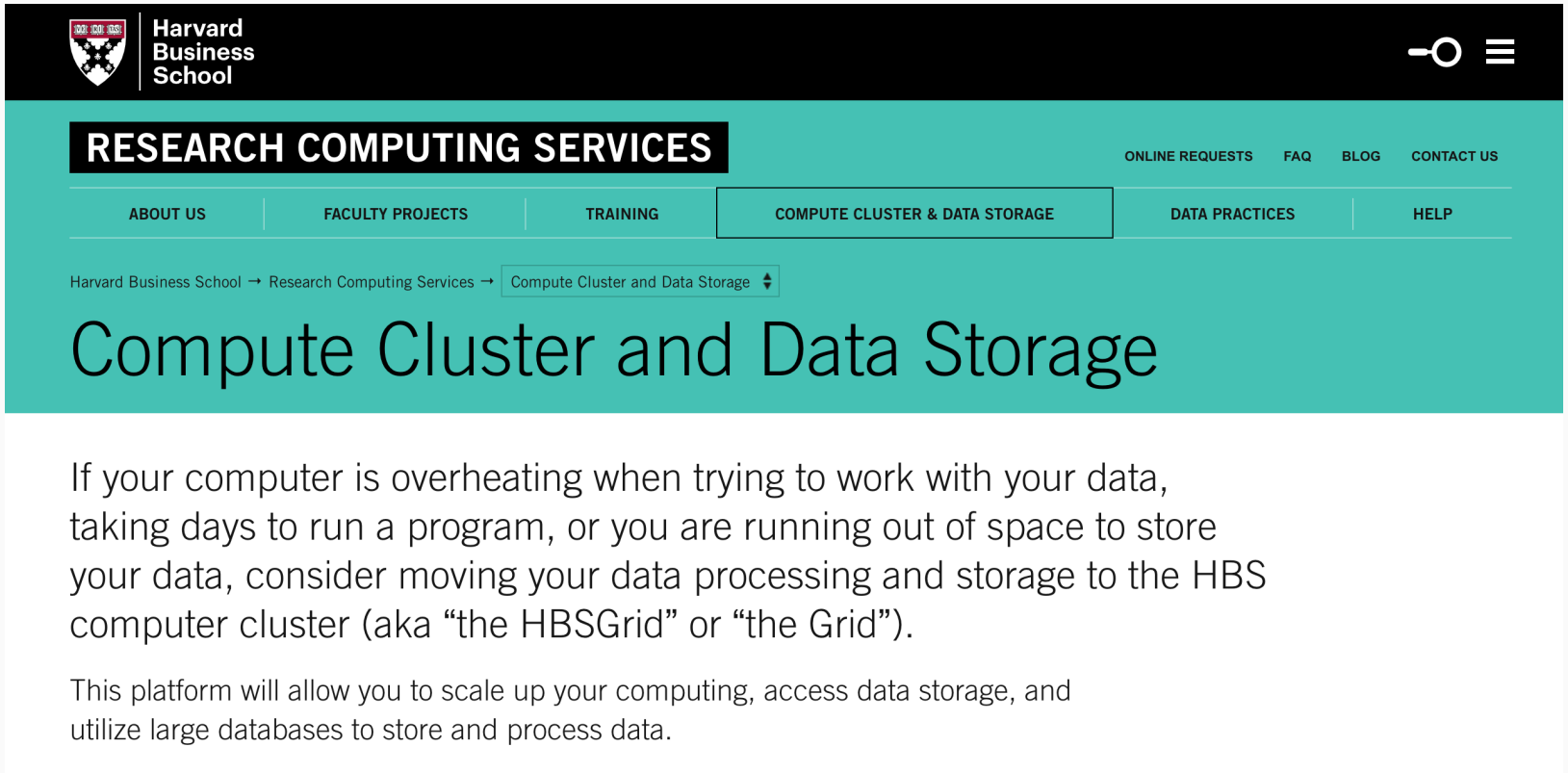
HOME / SERVICES / HUIT'S AWS CLOUD /

Join HUIT's AWS Cloud

To join HUIT's AWS Cloud, you must activate direct billing.
Follow the steps below to activate your account.

1. First, review the Harvard Cloud Services Policies.
2. Create an Amazon Web Services account if you don't already have one.
3. Obtain a HUIT billing account:
   - If you are already linked to an existing HUIT billing account, proceed to step 4.
   - If you want to use an existing HUIT billing account for which you are not a designated contact, you will need authorization from the account owners. Request authorization and wait for approval before proceeding to step 4.
   - If you need to create a new HUIT billing account, request one here and wait for confirmation before proceeding to step 4.
4. Complete the ServiceNow Cloud Account Request form.
5. You will receive an automated email message from Amazon asking you to confirm your enrollment in Consolidated Billing. **You must accept this invitation** in order to link your account with the master billing account.

Storage and Compute Options at Harvard



Data Serialization

Data serialization is the process of converting data structures, such as objects or data in memory, into a format that can be easily stored.

Choice of data serialization depends on efficiency, readability, ease of use, and interoperability.

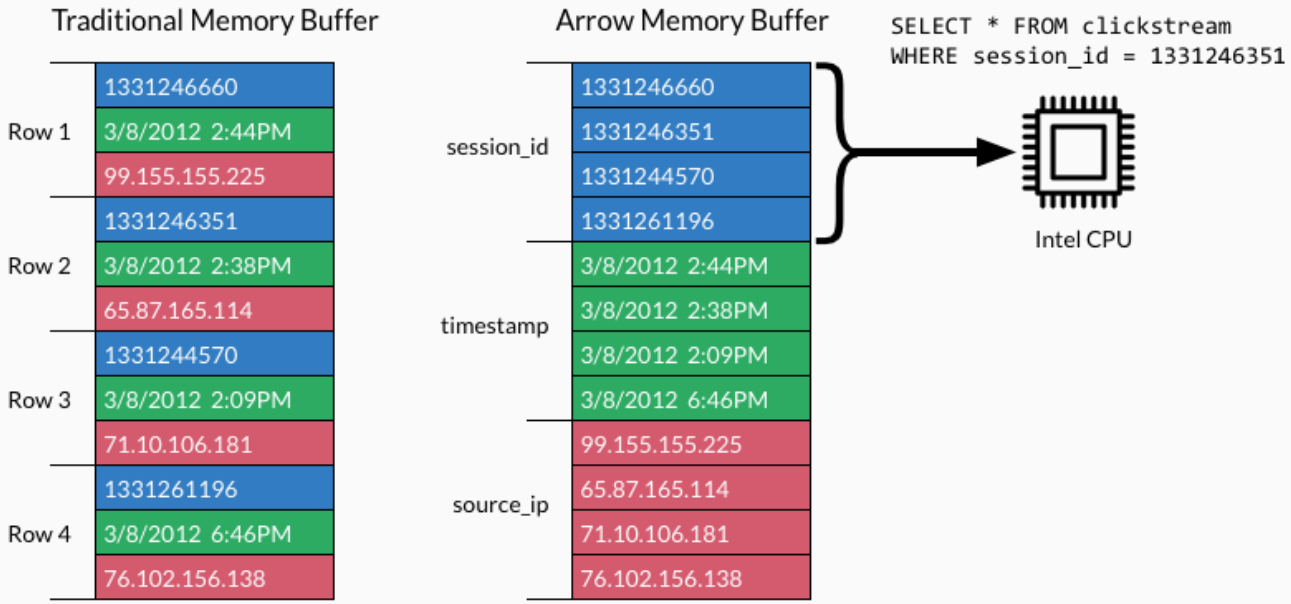
- CSV (Comma-Separated Values)
- JSON
- Apache Parquet
- Apache Feather
- Protocol Buffers (protobuf)

Comparison of Serialization Formats

	CSV	Parquet	Feather
Human-readable	Yes	No	No
Read/write efficiency	Moderate	High, but less efficient than Feather	High
Space efficiency	Low	High	High, but less efficient than Parquet
Type handling	No notion of types $\Rightarrow$ some data types will require parsing when loading	Wide support of types	Wide support of types
Data orientation	Row-oriented $\Rightarrow$ suboptimal if only need to load certain columns	Column-oriented $\Rightarrow$ better for static data than data streams	Column-oriented $\Rightarrow$ better for static data than data streams

Row vs. Columnar Formats

	session_id	timestamp	source_ip
Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138



Source: Apache Arrow

Working with Big Data

Just as you may run into trouble with data storage and reading/writing large data sets, you may also face limitations of the standard Python data science toolkit (Pandas and NumPy) when analyzing the data.

Problems:

- Data can't fit in memory
- Data operations become prohibitively slow

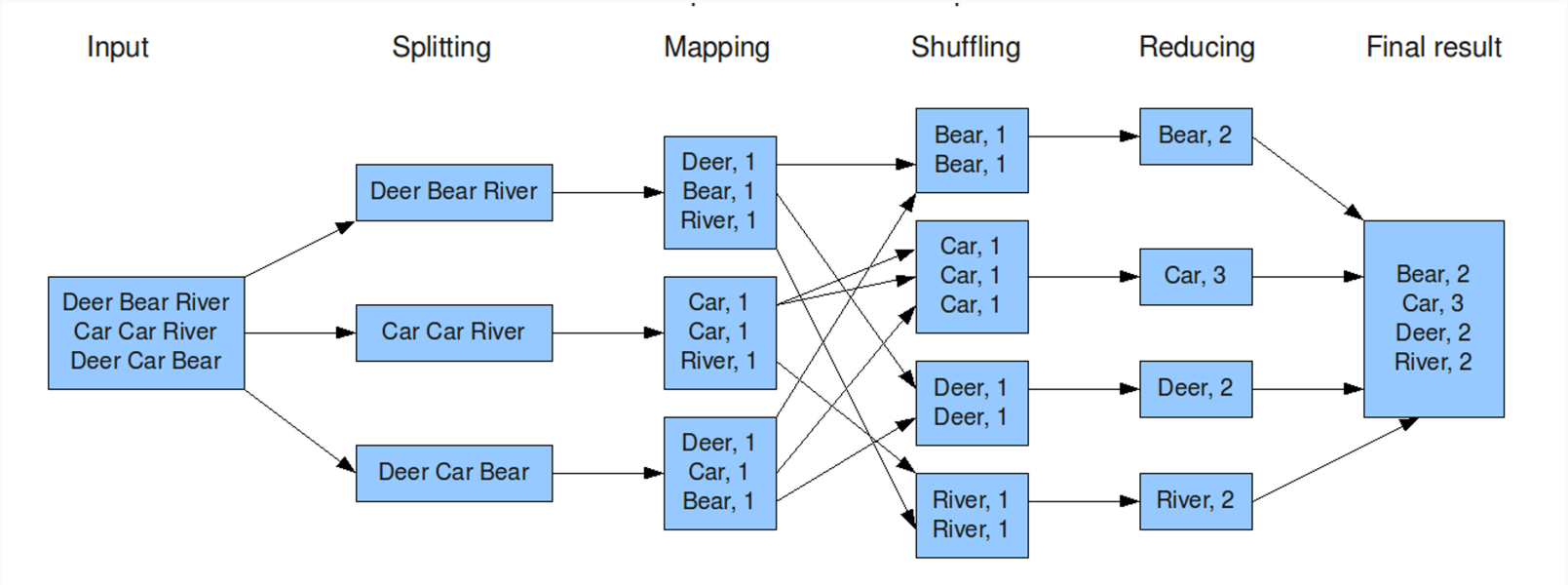
Solutions:

1. Expanding available compute, e.g., with high-performance computing
2. Using more efficient implementations of standard libraries

MapReduce

A programming model introduced by Google to address the challenge of processing vast amounts of data efficiently and in a distributed manner.

Key idea is simple: map data into a collection of <key, value> pairs and reduce over all pairs with the same key.



Apache Spark

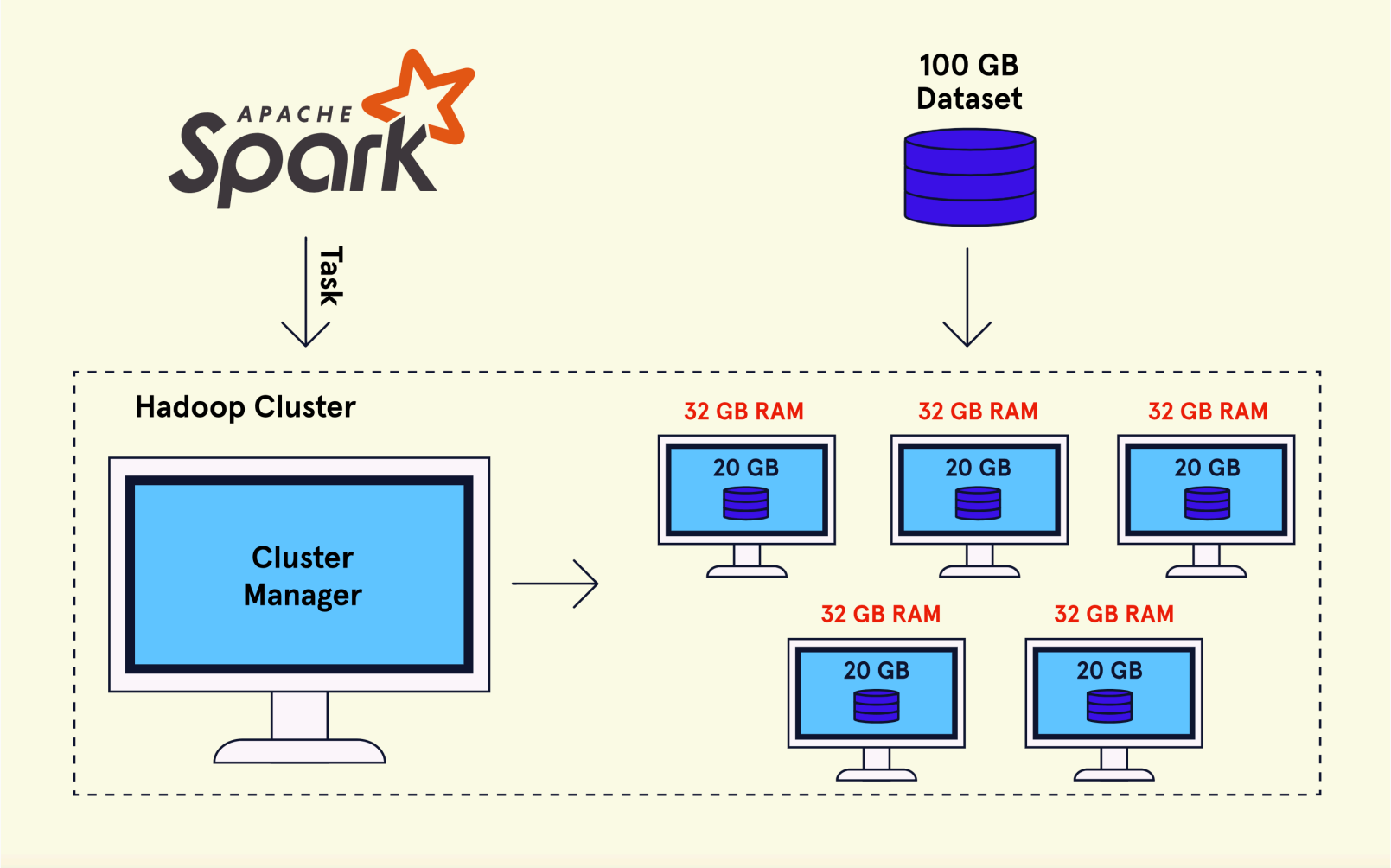
Apache Spark is a framework for distributed data processing and computation that draws from the MapReduce computation model.

PySpark is the Python API for Apache Spark.

- PySpark DataFrames allow for analyzing data with Python or SQL
- Spark's Pandas API allows for using Pandas syntax while working with data distributed across multiple nodes

```
import pandas as pd
import pyspark.pandas as ps

# create a pandas DataFrame
pdf = pd.DataFrame(np.random.randn(6, 4), columns=list('ABCD'))

# convert it to a pandas-on-Spark DataFrame
psdf = ps.from_pandas(pdf)
```

Apache Spark



Source: CodeAcademy

Polars

Polars is a Python package for working with and querying DataFrames that is written in Rust, with **Apache Arrow** as its foundation.

It provides a more (memory and time) efficient approach to querying data through **multithreading** and **lazy execution**.

Relatively new and evolving package, but rapid adoption.

```python
import pandas as pd

df = pd.read_csv("data.csv")

df.query("price < 5")["products"]

df.groupby("market")["price"].agg("mean")

# eager eval means groupby is applied to full dataframe
(
    df
    .groupby("market")["price"]
    .agg("mean")
    .query("market.isin([1, 2])")
)
```

Pandas

```python
import polars as pl

df = pl.read_csv("data.csv")

df.filter(pl.col("price") < 5).select(pl.col("product"))

df.groupby("market").agg(pl.col("price").mean())

# lazy eval means groupby is NOT applied to full dataframe
(
    df
    .groupby("market")
    .agg(pl.col("price").mean())
    .filter(pl.col("market").is_in([1, 2]))
)
```

Polars

"I strongly feel that Arrow is a key technology for the next generation of data science tools." – Wes McKinney, creator of Pandas

Practical Advice for Working with Big Data in Python

1. Save data tables with an efficient data format:
    - **Parquet** or **Feather** provide significant performance enhancements compared to **CSVs**
2. Use efficient packages:
    - **PySpark** or **Dask** when working on **distributed systems** (cluster computing)
    - **Polars** when working with data stored on a **single machine** (one node)
3. Speed up code by changing the compilation process:[1]
    - Use **Numba** when performing numerical tasks with NumPy arrays
    - Write computationally-intensive tasks in **Cython**, allowing for compilation in C.
4. Move to the cloud, use a GPU.[1]

[1] See the "Extras" section at the end of this lecture for more details.

Extras

Code Compilation and Interpretation

- **Compiled languages** translate the entire (human-readable) source code of a program into (machine-readable) binary code before execution.
  - Typically faster and more efficient code
  - Examples: C, C++, Rust, Julia
- **Interpreted languages** translate and execute source code line by line, typically on-the-fly during runtime.
  - Typically slower performance from interpretation overhead
  - More flexible and interactive, easier to debug
  - Examples: Python, R

Numba

The default implementation of Python is compiled into bytecode and then translated the bytecode into machine code at runtime by the Python Interpreter.

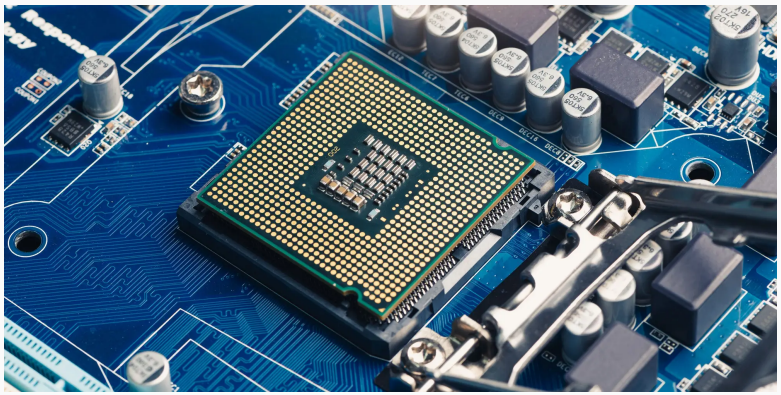
Numba is a **just-in-time compiler** that sidesteps interpreting bytecode by translating (a subset of) Python code directly into optimized machine code.

- Accelerates numerical and scientific computing with NumPy
- Acts on functions with the `@jit` decorator

```
from numba import @jit

@jit(nopython=True)
def loops(x):
    r = np.empty_like(x)
    n = len(x)
    for i in range(n):
        r[i] = np.cos(x[i]) ** 2 + np.sin(x[i]) ** 2
    return r
```

Without Numba (seconds)	With Numba (seconds)
25.2	0.670

CPUs vs. GPUs



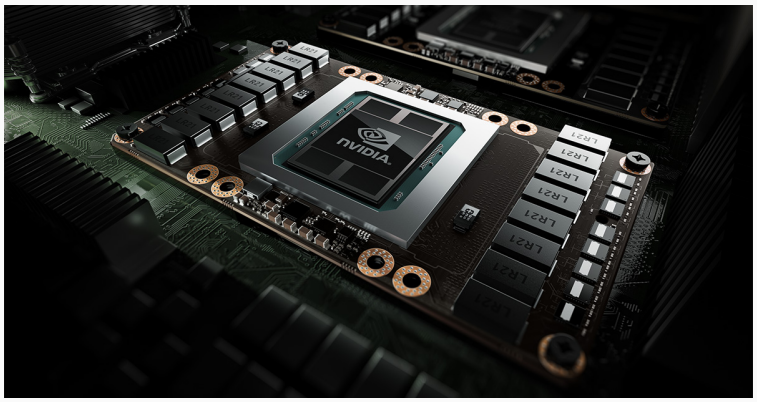
CPUs

General purpose processors

Low compute density

Low latency

Typically 4, 8, or 16 cores



GPUs

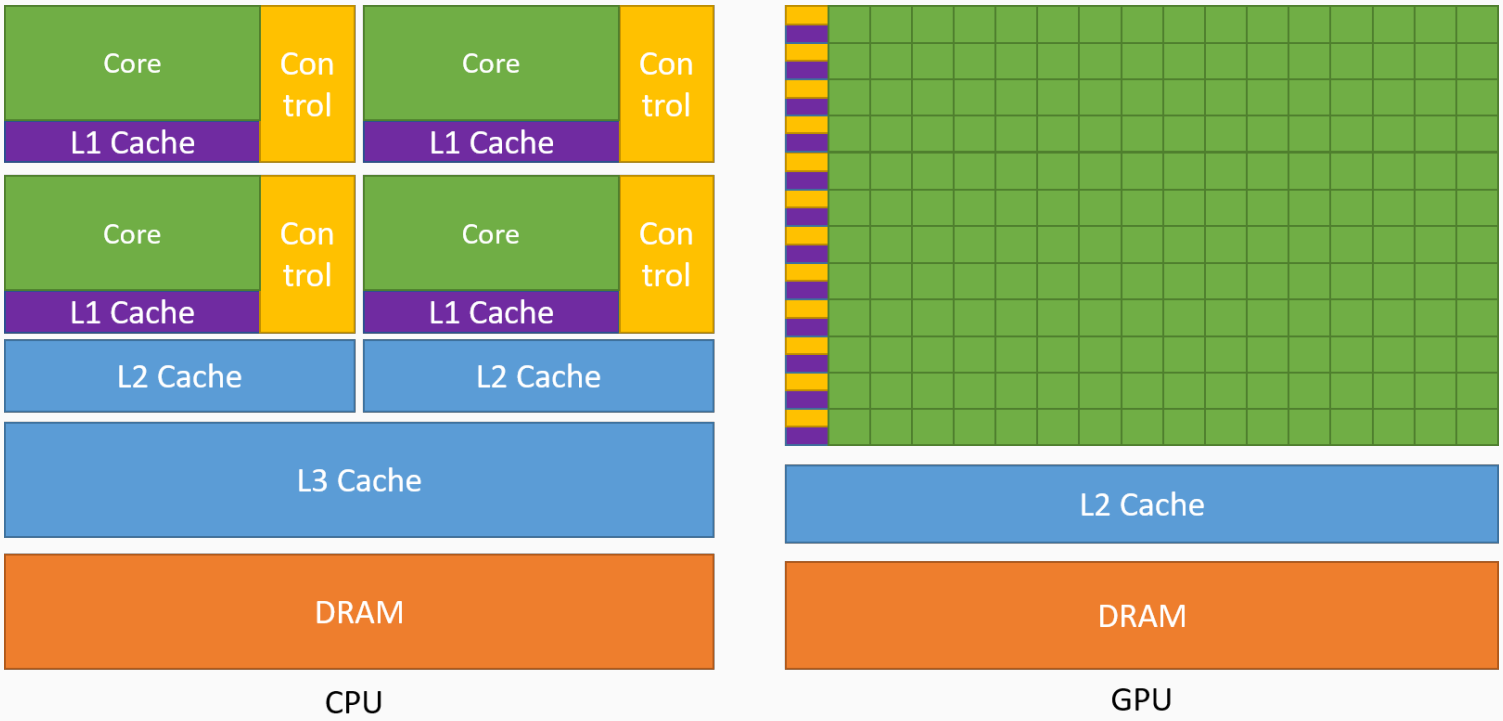
Specialized processors

High compute density

High throughput

Typically 100s-1,000s of cores

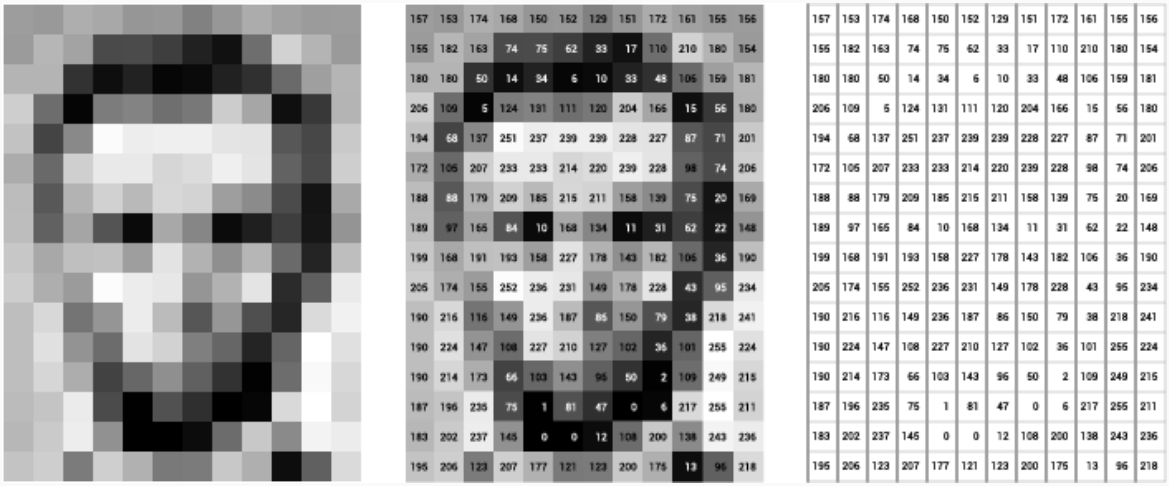
CPUs vs. GPUs



Source: CUDA C++ Programming Guide

GPUs

Graphics Processing Units (GPUs) were originally intended for graphics, primarily used in the gaming industry.



- Rendering pixels $\Longrightarrow$ millions of identical floating-point operations
  - $\Rightarrow$ Began to be leveraged for other highly parallelizable tasks
  - $\Rightarrow$ Unlocked revolution in deep learning

Use of GPUs

- Designed with a massively parallel architecture and used for tasks that can take advantage of this feature:
  - Graphics rendering (e.g., in the Apple Vision Pro)
  - Training neural networks (e.g., LLMs and self-driving cars)
  - Video editing (e.g., with Adobe Premiere Pro)
  - Cryptocurrency mining
  - Scientific and technical computing

CUDA

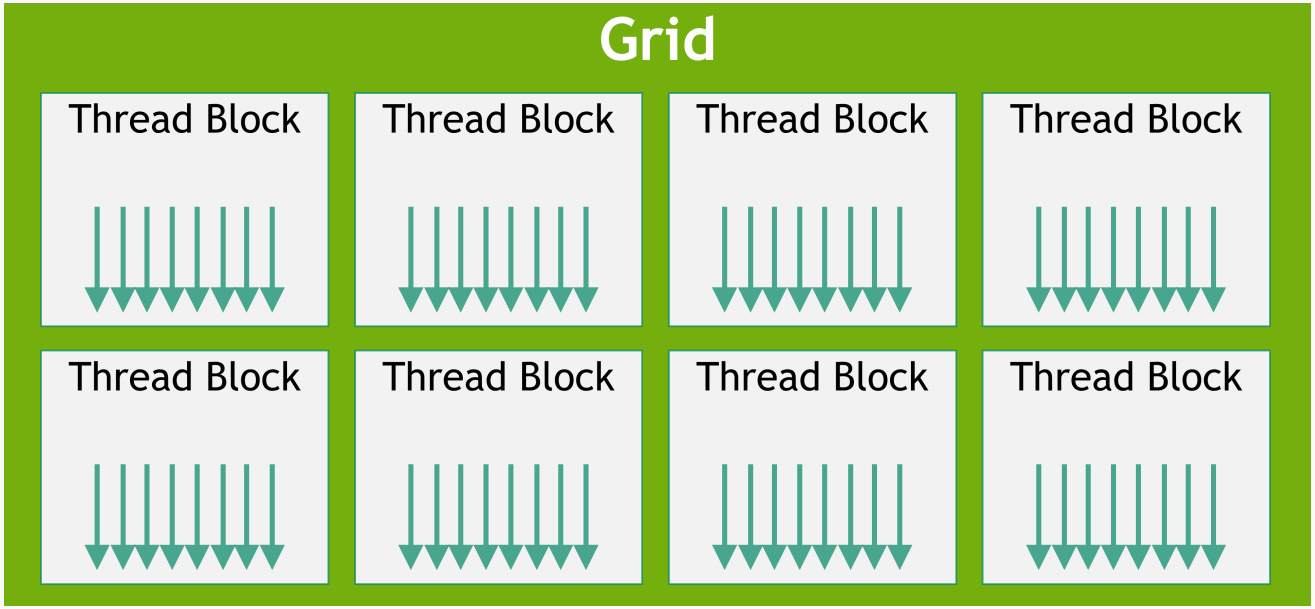
CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model for GPUs.

- Developed by Nvidia
- Allows developers to harness the computational power of GPUs for general-purpose computing
- Two key abstractions: thread hierarchy and memory hierarchy

Numba supports CUDA GPU programming:

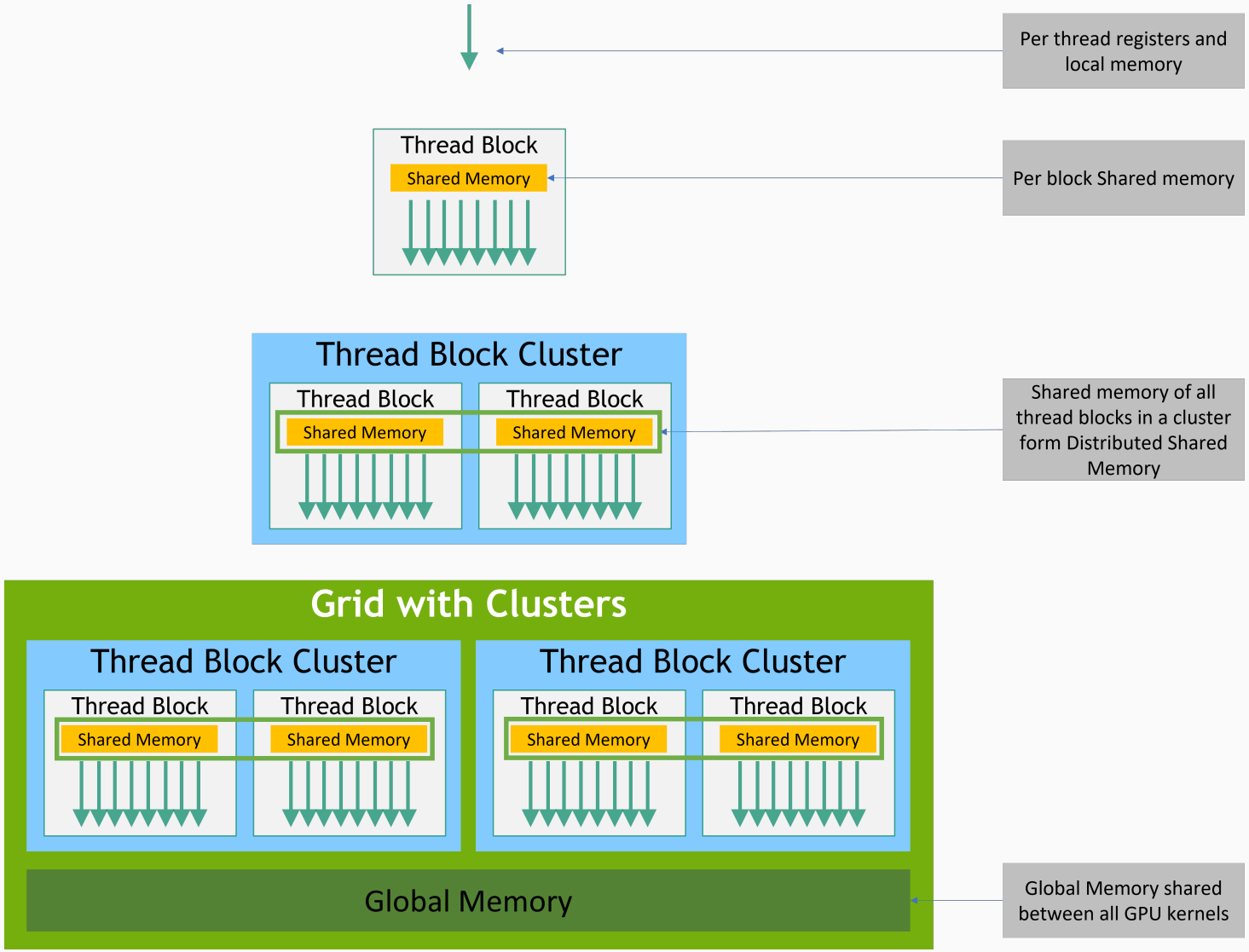
```
from numba import cuda

@cuda.jit(device=True)
def polar_to_cartesian(rho, theta):
    x = rho * math.cos(theta)
    y = rho * math.sin(theta)
    return x, y
```

Thread Hierarchy

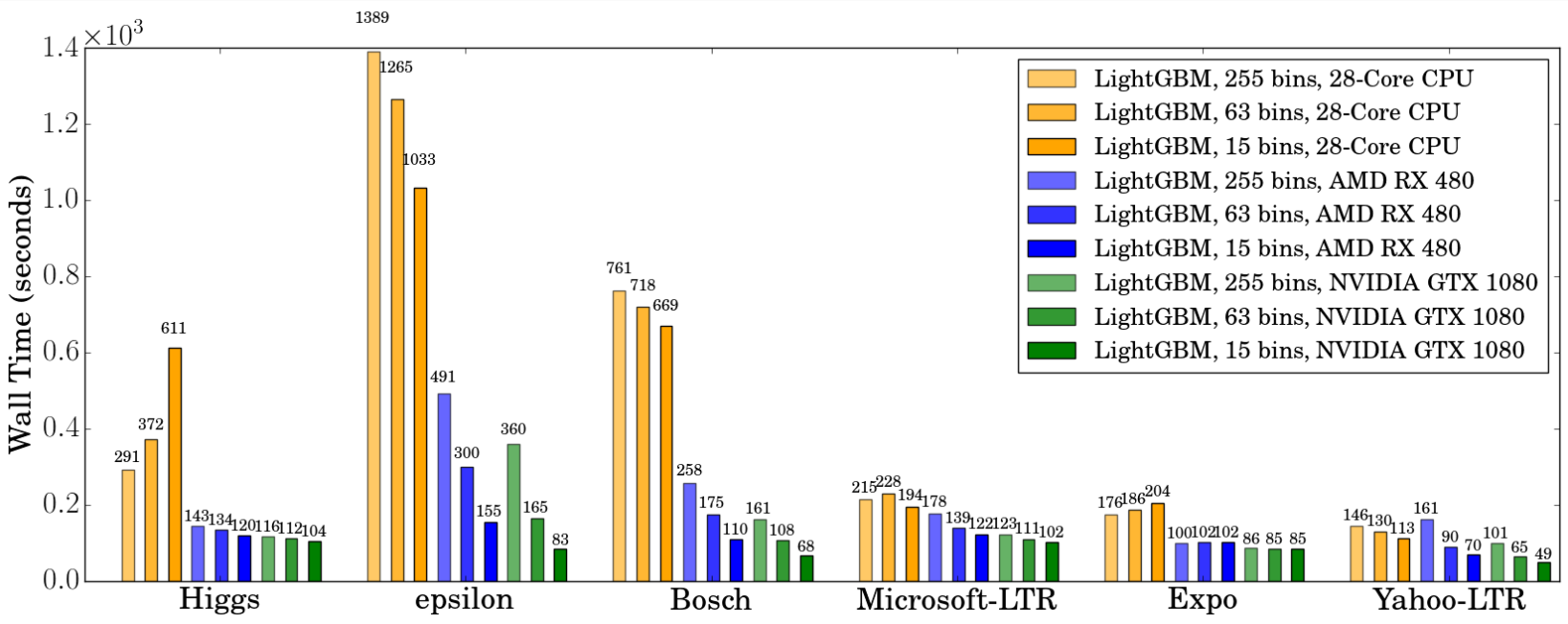


- **Thread**: basic unit of execution in CUDA
- **Block**: group of threads that can cooperate through shared memory and synchronization
- **Grid**: collection of blocks that execute independently (do not communicate)

Memory Hierarchy



Gradient Boosting Performance on CPUs and GPUs



Source: Zhang, Si, and Hsieh (2017)