Astroinformatics II (Semester 2 2024)

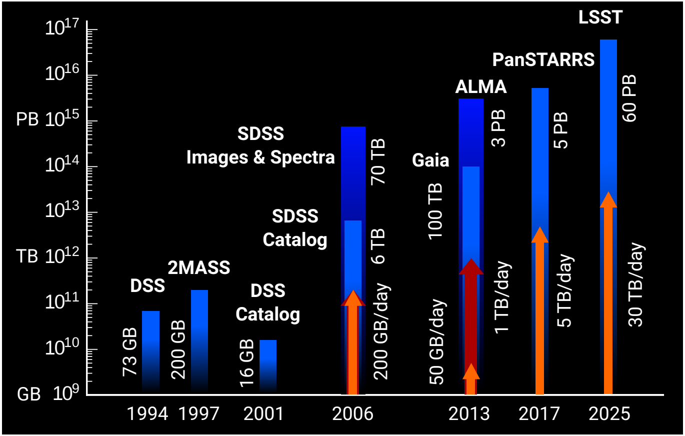
# Advanced Computer Infrastructure & Parallelization

**Nina Hernitschek**
Centro de Astronomía CITEVA
Universidad de Antofagasta

September 24, 2024

# Motivation

## Challenges with increasing data volume in astronomical surveys



# Motivation

## Challenges with increasing model/ algorithmic complexity

**Speed:** Nowadays computing facilities enable complex algorithms and models (such as neural networks). But increasing model training/ evaluation time limits productivity. We need a reasonable trade-off.

# Motivation

## Challenges with increasing model/ algorithmic complexity

**Speed:** Nowadays computing facilities enable complex algorithms and models (such as neural networks). But increasing model training/ evaluation time limits productivity. We need a reasonable trade-off.

validation error rate on 224 × 224 images from `https://image-net.org/`, benchmarked with `fb.resnet.torch` using four M40 GPUs

| Network | Top-1 error | Top-5 error | training time |
|---|---|---|---|
| ResNet-18 | 30.43% | 10.76% | 2.5 days |
| ResNet-50 | 24.01% | 7.02% | 5 days |
| ResNet-101 | 22.44% | 6.21% | 1 week |
| ResNet-152 | 22.16% | 6.16% | 1.5 weeks |

The Top-1 error indicates whether the top class (the one with the highest probability) matches the target label.
The Top-5 error indicates whether one of the top 5 predictions is matching the target label.

# Motivation

## Challenges with increasing model/ algorithmic complexity

**Energy Consumption:** this limits applications

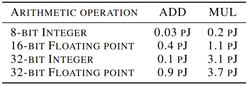
- on mobile devices: drains battery
- on data centers: increases the Total Cost of Ownership (TCO)

**example:** a single game of AlphaGo, involving 1920 CPUs and 280 GPUs, produces about a 3000 USD electricity bill

# Motivation

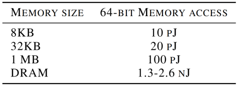
**analyzing** the problem:

larger models → more memory references → more energy



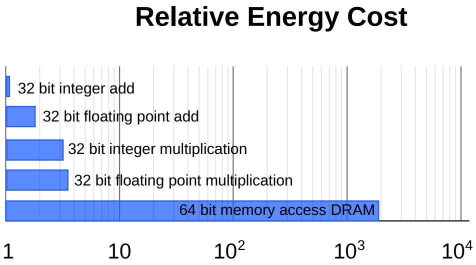
| ARITHMETIC OPERATION | ADD | MUL |
|---|---|---|
| 8-BIT INTEGER | 0.03 PJ | 0.2 PJ |
| 16-BIT FLOATING POINT | 0.4 PJ | 1.1 PJ |
| 32-BIT INTEGER | 0.1 PJ | 3.1 PJ |
| 32-BIT FLOATING POINT | 0.9 PJ | 3.7 PJ |

TABLE I: Approximate energy costs for different arithmetic operations in 45nm 0.9V.



| MEMORY SIZE | 64-BIT MEMORY ACCESS |
|---|---|
| 8KB | 10 PJ |
| 32KB | 20 PJ |
| 1 MB | 100 PJ |
| DRAM | 1.3-2.6 NJ |

TABLE II: Memory access energy expenditure (consumption) in 45nm 0.9V.



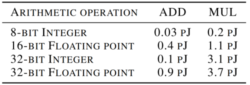
1 pJ (pico Joule) $= 10^{-12}$ J

tables from: M. Krouka (2021), Published in: 2021 IEEE 32nd Annual International Symposium on Personal, Indoor and Mobile Radio Communications

# Motivation

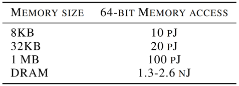
**analyzing** the problem:

larger models → more memory references → more energy



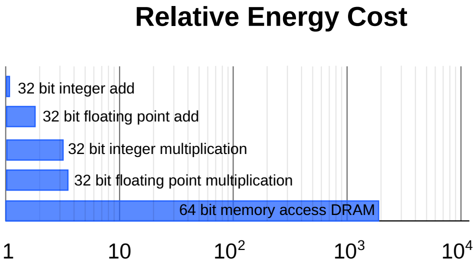
| Arithmetic operation | ADD | MUL |
|---|---|---|
| 8-bit Integer | 0.03 pJ | 0.2 pJ |
| 16-bit Floating point | 0.4 pJ | 1.1 pJ |
| 32-bit Integer | 0.1 pJ | 3.1 pJ |
| 32-bit Floating point | 0.9 pJ | 3.7 pJ |

TABLE I: Approximate energy costs for different arithmetic operations in 45nm 0.9V.



| Memory size | 64-bit Memory access |
|---|---|
| 8KB | 10 pJ |
| 32KB | 20 pJ |
| 1 MB | 100 pJ |
| DRAM | 1.3-2.6 nJ |

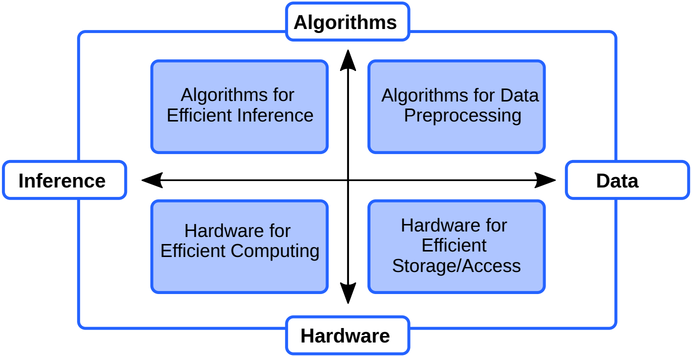
TABLE II: Memory access energy expenditure (consumption) in 45nm 0.9V.



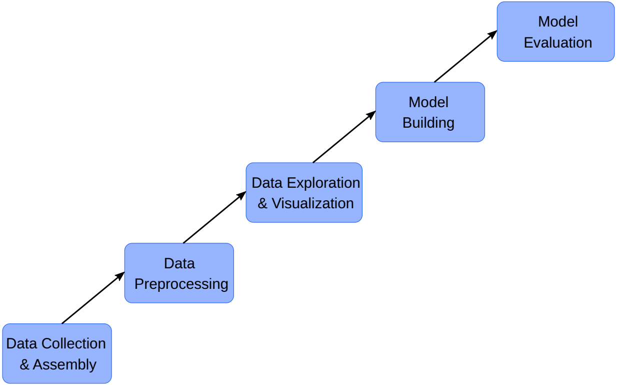
1 pJ (pico Joule) $= 10^{-12}$ J

tables from: M. Krouka (2021), Published in: 2021 IEEE 32nd Annual International Symposium on Personal, Indoor and Mobile Radio Communications

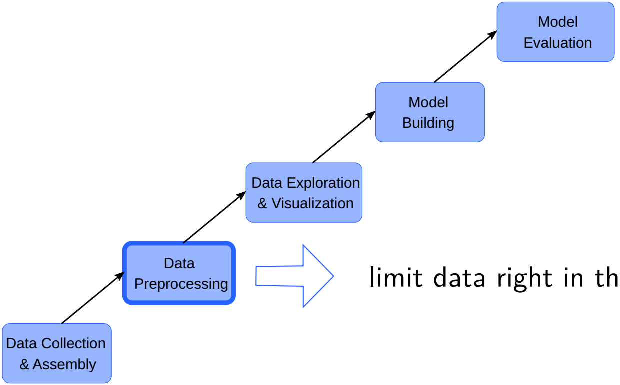
# Improving Efficiency


Algorithms
Algorithms for Efficient Inference
Algorithms for Data Preprocessing
Inference
Data
Hardware for Efficient Computing
Hardware for Efficient Storage/Access
Hardware

# Data Preprocessing



# Data Preprocessing



Data Collection & Assembly

Data Preprocessing

Data Exploration & Visualization

Model Building

Model Evaluation

limit data right in the beginning

# Data Preprocessing

Data Preprocessing transforms data so that it may be more easily processed.

Why is Data Preprocessing important?

# Data Preprocessing

Data Preprocessing transforms data so that it may be more easily processed.

Why is Data Preprocessing important?

The main agenda for a model to be accurate and precise in predictions is that the algorithm should be able to easily interpret the data's features.

Many real-world datasets are highly susceptible to be missing, inconsistent, and noisy due to their heterogeneous origin.

- Duplicate or missing values may give an incorrect view of the overall statistics of data.
- Outliers and inconsistent data points often tend to disturb the model's overall learning, leading to false predictions.

# Data Preprocessing

Data Preprocessing transforms data so that it may be easily processed by the machine learning algorithm.

Why is Data Preprocessing important?

The main agenda for a model to be accurate and precise in predictions is that the algorithm should be able to easily interpret the data's features.

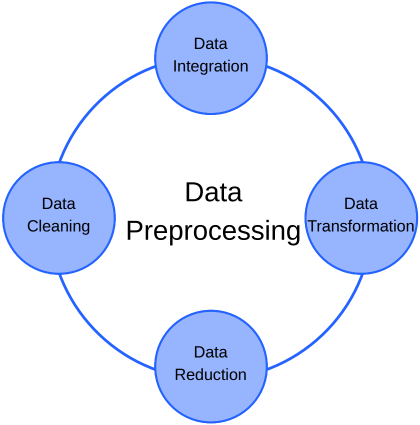
Many real-world datasets are highly susceptible to be missing, inconsistent, and noisy due to their heterogeneous origin.

- Duplicate or missing values may give an incorrect view of the overall statistics of data.
- Outliers and inconsistent data points often tend to disturb the model's overall learning, leading to false predictions.

⇨ Quality decisions must be based on quality data.

# Data Preprocessing

Data Preprocessing consists of four steps:



# Data Preprocessing

**Data Cleaning** as typically the first step of data processing involves:

**Filling missing values:**

- Ignore those tuples: This method should be considered when the dataset is huge and numerous missing values are present within a tuple. Caution: Can lead to an imbalanced data set!
- Fill in the missing values (imputation): Possible due to regression. Caution: Can lead to false predictions!

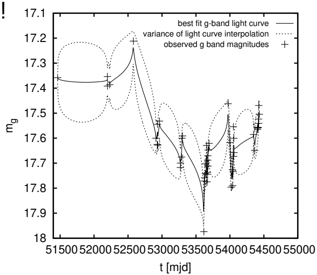
# Data Preprocessing

**Data Cleaning** as typically the first step of data processing involves:

**Filling missing values:**

- Ignore those tuples: This method should be considered when the dataset is huge and numerous missing values are present within a tuple. Caution: Can lead to an imbalanced data set!
- Fill in the missing values (imputation): Possible due to regression. Caution: Can lead to false predictions!

**Sometimes a better choice:**
Keeping missing values as they are.
Build your model in a way so it can deal with missing data.
E.g.: Interpolation, regression doesn't happen at the data level, but internally at the feature extractor or classifier level.



# Data Preprocessing

**Data Cleaning** as typically the first step of data processing involves:

**Dealing with noisy data:**
Techniques involve removing a random error or variance in a measured variable, using the following techniques:

- Measurement errors: If measurement errors are given (often for astronomical data), they can be incorporated into a fitting function.
- Binning: This technique works on data that can be sorted; all data in a bin are replaced by its mean, median or boundary values.
- Regression: This technique is generally used for prediction. It helps to smoothen data by fitting all the data points in a regression function.
- Clustering: Creation of groups/clusters from data having similar values. Values outside of the cluster can be treated as noisy data and can be removed.

# Data Preprocessing

**Data Cleaning** as typically the first step of data processing involves:

**Dealing with noisy data:**
Techniques involve removing a random error or variance in a measured variable, using the following techniques:

- Measurement errors: If measurement errors are given (often for astronomical data), they can be incorporated into a fitting function.
- Binning: This technique works on data that can be sorted; all data in a bin are replaced by its mean, median or boundary values.
- Regression: This technique is generally used for prediction. It helps to smoothen data by fitting all the data points in a regression function.
- Clustering: Creation of groups/clusters from data having similar values. Values outside of the cluster can be treated as noisy data and can be removed.

**Caution:** This noise could be where the interesting stuff is - rare events!

# Data Preprocessing

**Data Cleaning** as typically the first step of data processing involves:

**Removing outliers:**

- Statistical methods like $3\sigma$ clipping.
- Clustering techniques group together similar data points. The tuples that lie outside the cluster are outliers/inconsistent data.

# Data Preprocessing

**Data Cleaning** as typically the first step of data processing involves:

**Removing outliers:**

- Statistical methods like $3\sigma$ clipping.
- Clustering techniques group together similar data points. The tuples that lie outside the cluster are outliers/inconsistent data.

**Caution:** Again, outliers could be where the interesting stuff is - rare events!

**Sometimes a better choice:**

Keeping possible outliers as they are. Build your model in a way so it can deal with missing data. E.g.: building an outlier model that can account for complex outliers, without clipping too much data points.

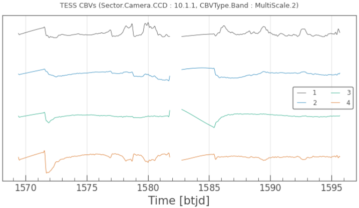
# Data Preprocessing

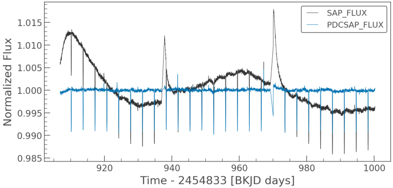
**Data Cleaning** as typically the first step of data processing involves:

**Removing trends:**

- General statistical methods like moving-average detrending.
- Specific models that make use of e.g. spacecraft telemetry or statistics of carefully selected subsamples like periodic variable stars.

Cotrending Basis Vectors (CBVs) are generated in the Kepler/K2/TESS pipeline to remove systematic trends in light curves.





# Data Preprocessing

**Data Integration** merges data from multiple sources.

With large astronomical surveys, follow-up surveys, multi-messenger astronomy... having access to multiple sources of data is nowadays very common in astronomy.

typical **issues** we encounter during data integration include:

- Schema integration and object matching: The data can be present in different formats, and attributes that might cause difficulty in data integration.
- Detection and resolution of data value conflicts. Example: How to deal with slightly different positions for the same astronomical source?

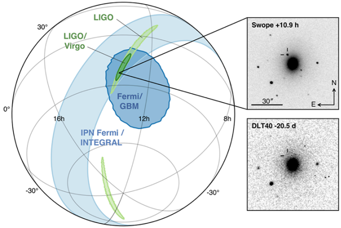


image credit: LIGO

# Data Preprocessing

**Data Transformation** changes the value, structure, or format of data using methods like the following ones:

**Generalization:**
The low-level data are converted to high-level information by using concept hierarchies. Example: observations are grouped to light curves or objects.

**Normalization:**
Numerical attributes are scaled to fit within a specified range. Normalization can be done in multiple ways, which are highlighted here:

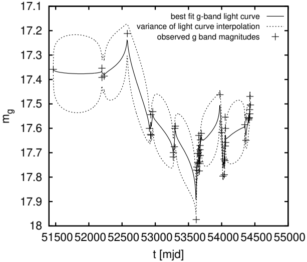
- min-max normalization
- Z-Score normalization
- decimal scaling normalization

# Data Preprocessing

**Data Transformation** changes the value, structure, or format of data using methods like the following ones:

Strategies especially common in time-domain astronomy are:

**Feature extraction:**
The algorithms used can be rather simple, as well as complex (machine-learning).



This light-curve fit leads to feature extraction: the features are parameters of a structure function model.

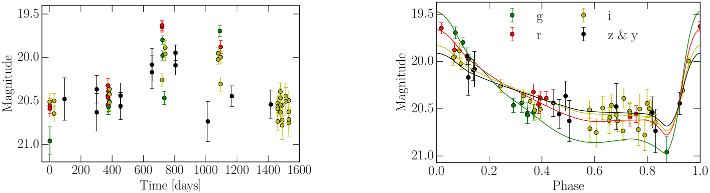
# Data Preprocessing

**Data Transformation** changes the value, structure, or format of data using methods like the following ones:

Strategies especially common in time-domain astronomy are:

**Phase-folding:** Phase-folding requires knowledge about the periodicity of a signal (such as a light curve). Period detection can involve algorithms such as Lomb-Scargle as well as complex machine-learning algorithms capable of dealing with data with noise and less-than-optimal cadence.
Caution: Previously measured periodicity information might have changed!

**example:** RR Lyrae period fitting with light curve templates from SDSS



# Data Preprocessing

**Data Reduction** obtains a reduced and better manageable data set that produces the same quality of analytical results. Common Data Reduction strategies include:

**Dimensionality reduction:**
Dimensionality reduction techniques are used to perform (abstract) feature extraction by using algorithms such as Principal Component Analysis.

**Numerosity reduction:**
The data can be represented as a model or equation like a regression model. This saves the burden of storing huge datasets instead of a model.

**Attribute subset selection:**
Selecting only specific features, attributes (table columns) is very important to reduce model sizes. Only attributes that add more value towards model training should be considered, and the rest all can be discarded for the problem at hand.

# Data Preprocessing

## Data Quality Assessment

A high-quality data set should fulfill the following conditions:

- data validity
- accuracy and reliability in terms of information
- consistency in all features
- completeness with no missing attribute values
- no redundancy

# Data Preprocessing

## Data Quality Assessment

A high-quality data set should fulfill the following conditions:

- data validity
- accuracy and reliability in terms of information
- consistency in all features
- completeness with no missing attribute values
- no redundancy

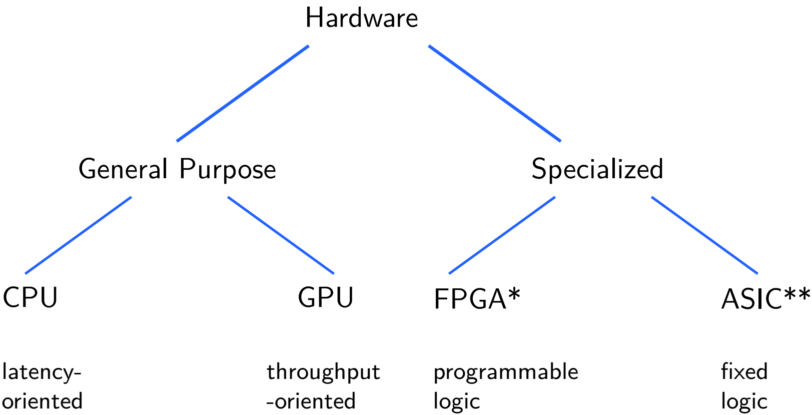
A common redundancy problem in astronomical data sets: multiple object IDs refer to the same astronomical object/ a light curve is split into multiple object IDs due to problems in light curve extraction.

# Data Preprocessing

**Best practices** for Data Preprocessing:

- The first step in Data Preprocessing is to understand your data. Just looking at your dataset can give you an intuition of what things you need to focus on.
- Use statistical methods or pre-built libraries that help you visualize the dataset and give a clear image of how your data looks in terms of class distribution.
- Summarize your data in terms of the number of duplicates, missing values, and outliers present in the data.
- Drop the fields you think have no use for the modeling or are closely related to other attributes. Dimensionality reduction is one of the very important aspects of Data Preprocessing.
- Do some feature engineering and figure out which attributes contribute most towards model training.

# Advanced Computing Infrastructure



Hardware

General Purpose

Specialized

CPU

GPU

FPGA*

ASIC**

latency-oriented

throughput-oriented

programmable logic

fixed logic

* Field Programmable Gate Arrays
** Application Specific Integrated Circuit

# Advanced Computing Infrastructure

High Performance Computing (HPC) is taking place in:

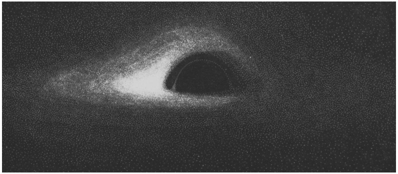
- international and national computing clusters
- university/ institute computing clusters
- even (to some extent) on your own gaming PC/ workstation

# Advanced Computing Infrastructure

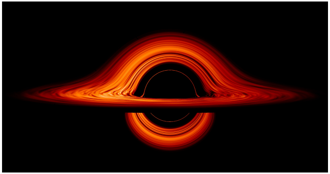
High Performance Computing (HPC) is taking place in:

- international and national computing clusters
- university/ institute computing clusters
- even (to some extent) on your own gaming PC/ workstation

opens up new possibilities in science and engineering, enabling researchers to tackle much larger and more complex challenges



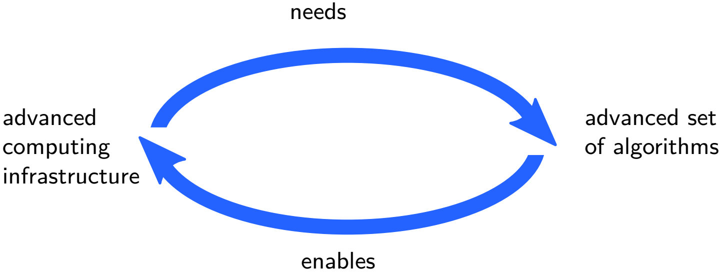
Visualization of a black hole, calculations done by using a computer, data points drawn by hand, J.-P. Luminet, (1979)



Visualization of a black hole by using a raytracing algorithm, NASA (2019)

# Advanced Computing Infrastructure

this advanced computing infrastructure makes programming techniques necessary that can access it



# Parallelization

**Generally:**
Nowadays computing infrastructure often allows for carrying out computations in parallel.

# Parallelization

**Generally:**
Nowadays computing infrastructure often allows for carrying out computations in parallel.

How we **implement** that exactly depends on the problem we want to solve.

# What can be parallelized?

In general, many things can be parallelized. It can be more or less obvious.

**example:**

```python
def vector_add(A, B, C, size):

    for item in range(0, size):
        C[item] = A[item] + B[item]

    return C
```

# What can be parallelized?

In general, many things can be parallelized. It can be more or less obvious.

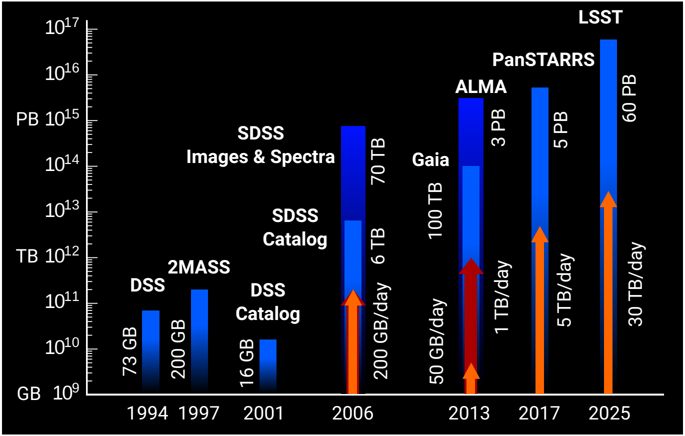
**example:**

```
def vector_add(A, B, C, size):

    for item in range(0, size):
        C[item] = A[item] + B[item]

    return C
```

In this code, each iteration of the `for` loop is **independent** from another. So, even if we reorder the iterations, or even compute each iteration in parallel or on a different device, we will still come up with the same output. Such algorithms are called **naturally parallel**, and they are the best candidates to be executed in parallel.

# What can be parallelized?

Common in astronomy: many files, many objects

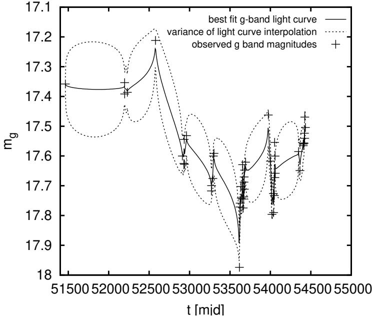


# What can be parallelized?

Common in astronomy: many files, many objects

We often apply the same analysis, like feature computation, on all objects from a data set independently.

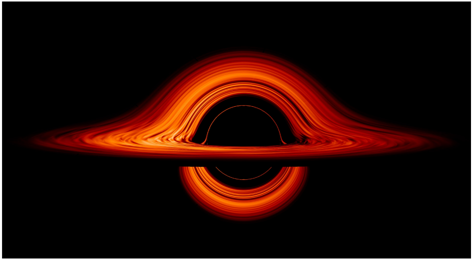
**example:** This type of light-curve fit was carried out for $\sim 10^9$ light curves from the Pan-STARRS1 $3\pi$ survey:



# What can be parallelized?

**example:** Also pixels in an image are (usually) independent of each other.

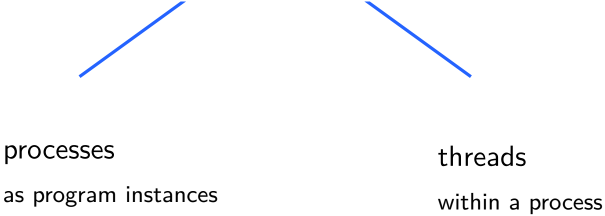
In this ray-traced image, pixels can be handled independently from each other.
When working with photographical images, in many cases, but not always, this can be done.



Visualization of a black hole by using a raytracing algorithm, NASA (2019)

# How to parallelize: Multiprocessing and Multithreading

There are generally two approaches:



processes

as program instances

threads

within a process

# Processes

A process represents an **independent instance of a running program**, created when executing a program.

# Processes

A process represents an **independent instance of a running program**, created when executing a program.

Each process operates **autonomously**, with its own memory space and resources.

Processes enable parallelism by leveraging multiple CPU cores for improved performance.

**Inter-process communication**, facilitated by mechanisms like pipes and queues, enhances collaboration between processes.

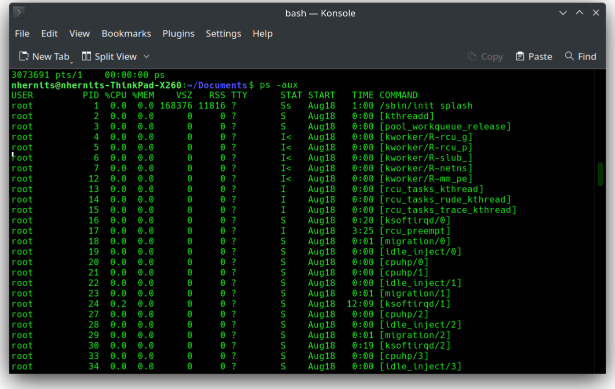
Processes provide **isolation**, ensuring that issues in one process don't affect others.

**Scalability** is achieved by distributing tasks across multiple processes, optimizing execution and enhancing overall performance.

# Processes

viewing processes under Linux:

The ps command displays information about running processes. You can use ps -aux to list all running processes along with information such as process ID (PID), terminal, CPU usage, and more.



```
3073691 pts/1    00:00:00 ps
nhernits@nhernits-ThinkPad-X260:~/Documents$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 168376 11816 ?        Ss   Aug18   1:00 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    Aug18   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    Aug18   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<   Aug18   0:00 [kworker/R-rcu_g]
root           5  0.0  0.0      0     0 ?        I<   Aug18   0:00 [kworker/R-rcu_p]
root           6  0.0  0.0      0     0 ?        I<   Aug18   0:00 [kworker/R-slub_]
root           7  0.0  0.0      0     0 ?        I<   Aug18   0:00 [kworker/R-netns]
root          12  0.0  0.0      0     0 ?        I<   Aug18   0:00 [kworker/R-mm_pe]
root          13  0.0  0.0      0     0 ?        I    Aug18   0:00 [rcu_tasks_kthread]
root          14  0.0  0.0      0     0 ?        I    Aug18   0:00 [rcu_tasks_rude_kthread]
root          15  0.0  0.0      0     0 ?        I    Aug18   0:00 [rcu_tasks_trace_kthread]
root          16  0.0  0.0      0     0 ?        S    Aug18   0:20 [ksoftirqd/0]
root          17  0.0  0.0      0     0 ?        I    Aug18   3:25 [rcu_preempt]
root          18  0.0  0.0      0     0 ?        S    Aug18   0:01 [migration/0]
root          19  0.0  0.0      0     0 ?        S    Aug18   0:00 [idle_inject/0]
root          20  0.0  0.0      0     0 ?        S    Aug18   0:00 [cpuhp/0]
root          21  0.0  0.0      0     0 ?        S    Aug18   0:00 [cpuhp/1]
root          22  0.0  0.0      0     0 ?        S    Aug18   0:00 [idle_inject/1]
root          23  0.0  0.0      0     0 ?        S    Aug18   0:01 [migration/1]
root          24  0.2  0.0      0     0 ?        S    Aug18  12:09 [ksoftirqd/1]
root          27  0.0  0.0      0     0 ?        S    Aug18   0:00 [cpuhp/2]
root          28  0.0  0.0      0     0 ?        S    Aug18   0:00 [idle_inject/2]
root          29  0.0  0.0      0     0 ?        S    Aug18   0:01 [migration/2]
root          30  0.0  0.0      0     0 ?        S    Aug18   0:19 [ksoftirqd/2]
root          33  0.0  0.0      0     0 ?        S    Aug18   0:00 [cpuhp/3]
root          34  0.0  0.0      0     0 ?        S    Aug18   0:00 [idle_inject/3]
```

# Processes

Process-based parallization (multiprocessing) can be carried out within Python using the `multiprocessing` library:

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

(we will see examples in this week's tutorial session)

# Multithreading

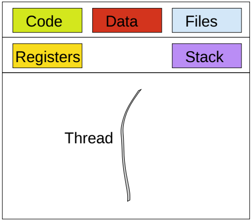
A **thread** is a compact unit of execution that symbolizes a separate control flow within a program.

# Multithreading

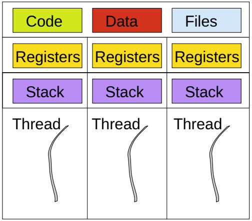
A **thread** is a compact unit of execution that symbolizes a separate control flow within a program.

Multithreading allows numerous threads to be executed simultaneously within a single process. It thus permits parallelism at the thread level within a process.

By enabling autonomous execution of program components, multithreading promotes efficient task management and boosts CPU utilization, enhancing responsiveness and performance.



Single-Threaded Process



Multi-Threaded Process

# Multithreading

## Advantages of Multithreading

**Improved Responsiveness:** Multithreading can improve how responsive a program is. It lets the program carry out laborious tasks in the background while being interactive and sensitive to user interaction.

**Efficient Resource Utilization:** A program can better utilize resources (CPU and memory) by running numerous threads concurrently, reducing idle time and maximizing resource utilization.

**Simplified Design and Modularity:** Multithreading can simplify program design by dividing complicated processes into smaller, more manageable threads. It encourages modularity, which makes it simpler to maintain and reason about the code.

**Shared Memory Access:** Direct access to shared memory by threads in the same process enables efficient data sharing and communication between them. This can be advantageous when threads must cooperate, exchange information, or work on a common data structure.

# Multithreading

## Disadvantages of Multithreading

**Synchronization and Race Conditions:** To coordinate access to shared resources, synchronization techniques are required by multithreading to avoid corrupted data and unpredictable behaviour. Synchronization might result in performance overhead and increases the complexity of the code.

**Increased Complexity and Debugging Difficulty:** Programs using many threads are typically more sophisticated than those with a single thread. Due to non-deterministic behaviour and probable race situations, debugging multithreaded programmes can also be more challenging.

**Potential for Deadlocks and Starvation:** In which threads cannot move forward because they are waiting for one another to release resources, they can result from improper synchronization or resource allocation. Similar to how some threads may run out of resources if resource allocation is not correctly controlled.

# Multithreading

## Disadvantages of Multithreading

**Global Interpreter Lock (GIL):** The Global Interpreter Lock (GIL) in Python prevents multithreaded programs from properly utilizing multiple CPU cores. One thread can only run Python bytecode simultaneously due to the GIL, which restricts the possible performance advantages of multithreading for CPU-bound operations. Multithreading can still be advantageous for I/O-bound or concurrent I/O and CPU-bound scenarios requiring external libraries or sub-processes.

# Multithreading

## − Disadvantages of Multithreading

**Global Interpreter Lock (GIL):** The Global Interpreter Lock (GIL) in Python prevents multithreaded programs from properly utilizing multiple CPU cores. One thread can only run Python bytecode simultaneously due to the GIL, which restricts the possible performance advantages of multithreading for CPU-bound operations. Multithreading can still be advantageous for I/O-bound or concurrent I/O and CPU-bound scenarios requiring external libraries or sub-processes.

⇨ Determining when and how to use multithreading successfully requires understanding its benefits and drawbacks.

# Implementing Multithreading in Python

Python provides a module that enables the construction and administration of threads, thus making multithreaded applications easier to implement.

**example:**

```
import threading

def task():
    print("Thread task executed")

# Create a thread
thread = threading.Thread(target=task)

# Start the thread
thread.start()

# Wait for the thread to complete
thread.join()

print("Thread execution completed")
```

We define a task function. We create a thread by instantiating the `Thread` class with the target argument set to the `task` function. The thread is started using the `start()` method, which initiates the execution of the task function in a separate thread. Finally, we use the `join()` method to wait for the thread to complete before moving forward with the main program.

# Implementing Multithreading in Python

The threading module provides various methods and attributes to **manage threads**. Some commonly used methods include:

`start()`: Initiates the execution of the thread's target function.

`join([timeout])`: Waits for the thread to complete execution. The optional timeout argument specifies the maximum time to wait for thread completion.

`is_alive()`: Returns True if the thread is executing.

`name`: A property that gets or sets the thread's name.

`daemon`: A Boolean property determining whether the thread is a daemon thread. Daemon threads are abruptly terminated when the main program exits.

# Implementing Multithreading in Python

The threading module provides various methods and attributes to **manage threads**. Some commonly used methods include:

`start()`: Initiates the execution of the thread's target function.

`join([timeout])`: Waits for the thread to complete execution. The optional timeout argument specifies the maximum time to wait for thread completion.

`is_alive()`: Returns True if the thread is executing.

`name`: A property that gets or sets the thread's name.

`daemon`: A Boolean property determining whether the thread is a daemon thread. Daemon threads are abruptly terminated when the main program exits.

These are only a few of the provided methods for thread management. To help manage shared resources and synchronize thread execution, the threading module provides extra features, including locks, semaphores, condition variables, and thread synchronization.

# Concurrency and Parallelism

Concurrency and parallelism are related concepts but have distinct differences:

**Concurrency** describes a system's capacity of executing many tasks so they can overlap and advance simultaneously. While tasks may not run simultaneously in a concurrent system, they can advance interleaved. Even when they run on a single processing unit, coordinating several tasks concurrently is the main goal.

**Parallelism** entails carrying out numerous tasks concurrently, each assigned to a different processing unit or core. A parallel system carries out tasks concurrently and in parallel. The emphasis is on breaking a difficulty into more manageable actions that it can carry out concurrently to produce quicker outcomes.

# Concurrency and Parallelism

Concurrency and parallelism are related concepts but have distinct differences:

**Concurrency** describes a system's capacity of executing many tasks so they can overlap and advance simultaneously. While tasks may not run simultaneously in a concurrent system, they can advance interleaved. Even when they run on a single processing unit, coordinating several tasks concurrently is the main goal.

**Parallelism** entails carrying out numerous tasks concurrently, each assigned to a different processing unit or core. A parallel system carries out tasks concurrently and in parallel. The emphasis is on breaking a difficulty into more manageable actions that it can carry out concurrently to produce quicker outcomes.

Using many processes running simultaneously with multiprocessing enables parallelism while enabling numerous threads within a single process enables concurrency via multithreading.

# Concurrency with Multithreading

The following **example** shows concurrency with multithreading:

```
import threading
import time

def task(name):
    print(f"Task {name} started")
    time.sleep(2)  # Simulating some time-consuming task
    print(f"Task {name} completed")

# Creating multiple threads
threads = []

for i in range(5):
    t = threading.Thread(target=task, args=(i,))
    threads.append(t)
    t.start()

# Waiting for all threads to complete
for t in threads:
    t.join()

print("All tasks completed")
```

We create five threads and assign each to execute the task function with a different name. Concurrency is enabled via multithreading by enabling numerous threads within a single process.
You'll observe that the tasks start and complete in an interleaved manner, indicating concurrent execution.

# Parallelism with Multiprocessing

The following **example** shows parallelism with multiprocessing:

```
import multiprocessing
import time

def task(name):
    print(f"Task {name} started")
    time.sleep(2)  # Simulating some time-consuming task
    print(f"Task {name} completed")

# Creating multiple processes
processes = []

for i in range(5):
    p = multiprocessing.Process(target=task, args=(i,))
    processes.append(p)
    p.start()

# Waiting for all processes to complete
for p in processes:
    p.join()

print("All tasks completed")
```

In this example, we define the same task function as before. However, instead of creating threads, we make five processes using multiprocessing. Process class. Each process is assigned to execute the task function with a different name. The processes are started and then joined to wait for their completion.

# Parallelism with Multiprocessing

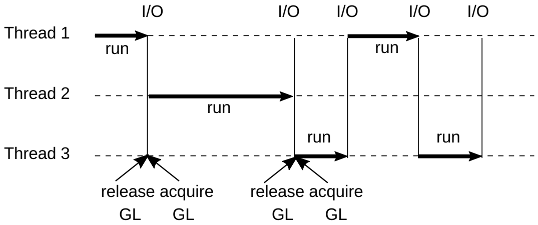
The following **example** shows parallelism with multiprocessing:

```python
import multiprocessing
import time

def task(name):
    print(f"Task {name} started")
    time.sleep(2)  # Simulating some time-consuming task
    print(f"Task {name} completed")

# Creating multiple processes
processes = []

for i in range(5):
    p = multiprocessing.Process(target=task, args=(i,))
    processes.append(p)
    p.start()

# Waiting for all processes to complete
for p in processes:
    p.join()

print("All tasks completed")
```

When you run this code, you'll see that the tasks are executed in parallel. Each process runs independently, utilizing separate CPU cores. As a result, the tasks may be completed in any order, and youll observe a significant reduction in the execution time compared to the multithreading example.

# The Global Interpreter Lock

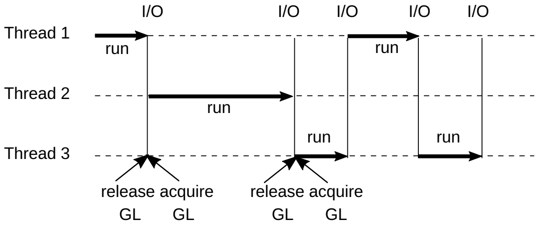
The Global Interpreter Lock (GIL) in CPython, the language's default implementation, ensures that only one thread at a time can execute Python bytecode. This means that even a Python program with several threads can only advance one thread at a time.



Python's GIL was created to make memory management easier and guard against concurrent object access. However, because only one thread can run Python bytecode, even on computers with many CPU cores, it also restricts the potential performance advantages of multithreading for CPU-bound operations.

# The Global Interpreter Lock

The Global Interpreter Lock (GIL) in CPython, the language's default implementation, ensures that only one thread at a time can execute Python bytecode. This means that even a Python program with several threads can only advance one thread at a time.



The GIL does not completely forbid or invalidate the use of multithreading for specific sorts of operations:

Multithreading in Python is better suited to I/O-bound activities, concurrent I/O jobs, and situations where threads must wait a long time for I/O operations to complete.

# The Global Interpreter Lock

For this reason, Python's `multiprocessing` module, which uses distinct processes rather than threads, is often advised as a way to get around the GIL's restrictions for CPU-bound workloads that can benefit from real parallelism over many CPU cores.

When considering whether to employ multithreading or consider alternate strategies like multiprocessing for obtaining the desired performance and concurrency in a Python program, it is essential to understand the impact of the GIL on multithreading in Python.

# The Global Interpreter Lock

For this reason, Python's `multiprocessing` module, which uses distinct processes rather than threads, is often advised as a way to get around the GIL's restrictions for CPU-bound workloads that can benefit from real parallelism over many CPU cores.

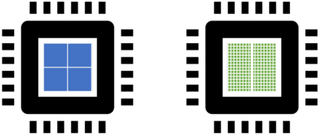
When considering whether to employ multithreading or consider alternate strategies like multiprocessing for obtaining the desired performance and concurrency in a Python program, it is essential to understand the impact of the GIL on multithreading in Python.

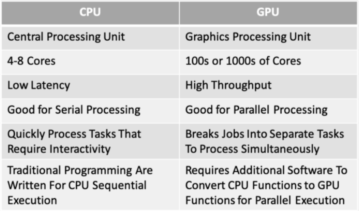
Not every Python implementation has a GIL. **Alternative Python implementations**, such as Jython and IronPython, do not include a GIL, enabling genuine thread parallelism. Additionally, there are circumstances where certain **extension modules**, like those written in C/C++, can release the GIL deliberately to boost concurrency.

# GPU-Accelerated Computing

In a computer, we typically find a Central Processing Unit (CPU), as well as a Graphics Processing Unit (GPU).

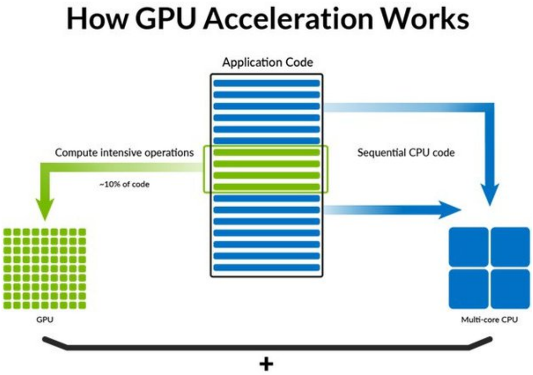
GPUs process data in order to render images on an output device, such as a screen. However, modern GPUs are general purpose computing devices that can be used to perform any kind of computation:
They are massively parallel devices that can execute thousands of threads at the same time.





| 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 |

# GPU-Accelerated Computing



# GPU-Accelerated Computing

**Tasks** suited to a GPU are such as:

- summarizing values in an array
- matrix multiplication, array operations
- image processing (images are arrays of pixels)
- machine learning which uses a combination of the above

# GPU-Accelerated Computing

**Tasks** suited to a GPU are such as:

- summarizing values in an array
- matrix multiplication, array operations
- image processing (images are arrays of pixels)
- machine learning which uses a combination of the above

**Implementation in Python:**

CuPy: A GPU array library that implements a subset of the NumPy and SciPy interfaces. It is a convenient tool for those familiar with NumPy to explore the power of GPUs, without the need to write code in a GPU programming language like CUDA and OpenCL.
For installation: see `https://docs.cupy.dev/en/stable/install.html`

# GPU-Accelerated Computing

**example:** Sorting an array

We will be creating a large array in Python, then sorting it with the `sort()` function of NumPy on the CPU.
We compute the time of the sorting using `timeit`.
Run the following in iPython:

```
import numpy as np

size = 8192 * 8192
array = np.random.random(size).astype(np.float32)

%timeit -n 1 -r 1 result = np.sort(array)
```

```
8.29 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

So it took 8.29 seconds to sort the array.

# GPU-Accelerated Computing

**example:** Sorting an array

Now let's perform the same sorting, but this time on a GPU. We will be using the `sort()` function from CuPy. Again, we compute the time of the sorting using the `%timeit` command.

```
import cupy as cp
import numpy as np

size = 8192 * 8192
array = np.random.random(size).astype(np.float32)

array_gpu = cp.asarray(array)
%timeit -n 1 -r 1 result_gpu = cp.sort(array_gpu)
```

```
36.6 ms   0 ns per loop (mean   std. dev. of 1 run, 1 loop each)
```

So it took 36.6 ms seconds to sort the array. That's much faster than using the CPU.

# GPU-Accelerated Computing

**example:** Sorting an array

From the results, we noticed that sorting the array with CuPy, i.e. using the GPU, is faster than with NumPy, using the CPU.

To quantify the speed, we will compute the speedup of using CuPy over NumPy. The speedup is defined as the ratio between the sequential (NumPy in our case) and parallel (CuPy in our case) execution times.

Note: Execution times must be in the same unit (milliseconds or seconds)

```
speedup = 8.29 / 0.0366
print(speedup)
```

We get a speedup by a factor of ∼226.

# GPU-Accelerated Computing: Convolution

**example:** Convolution in Python

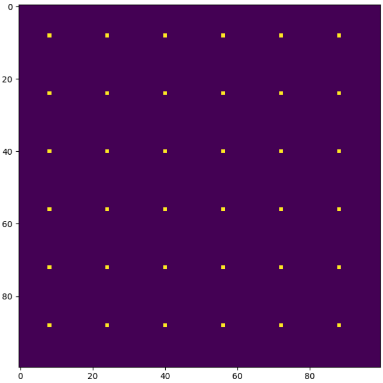
We start by generating an image using Python and NumPy.
The image is filled with zeros, except for isolated pixels with value one, on a regular grid. The plan is to convolve it with a Bicubic interpolation.
Then record the time it takes to execute this convolution on the host.

# GPU-Accelerated Computing: Convolution

First we construct the image:

```
import numpy as np

#Visualization
import pylab as pyl


# Construct an image with repeated delta functions
deltas = np.zeros((4096, 4096))
deltas[8::16,8::16] = 1



# Display the image
pyl.imshow(deltas[0:100, 0:100])
pyl.show()
```

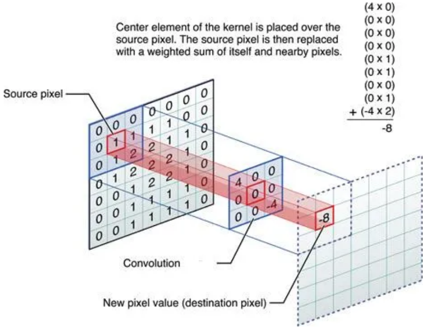
# GPU-Accelerated Computing: Convolution



# GPU-Accelerated Computing: Convolution

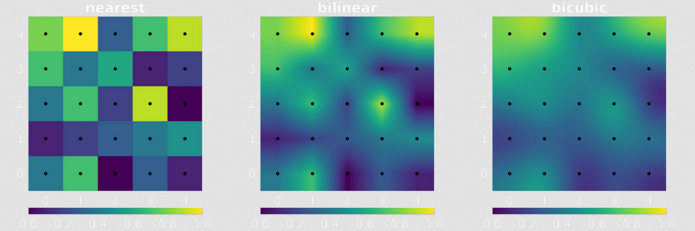
The computation we want to perform on this image is a convolution, first on the CPU then on the GPU to compare the results and execution times.

We can think of an image as a matrix of colour values, so when we convolve that image with a filter, we generate a new matrix with different color values.
The following illustrates the principle of convolution of an image with a kernel:


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution
New pixel value (destination pixel)

# GPU-Accelerated Computing: Convolution

In our example, we will convolve our image with a Bicubic interpolation:



The Bicubic interpolation method is known for its ideal combination of processing time and output quality.

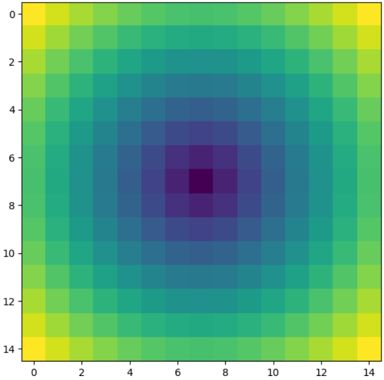
# GPU-Accelerated Computing: Convolution

## Convolution on a CPU: Scipy

Let us first construct the Bicubic, and then display it.

```
import numpy as np

x, y = np.meshgrid(np.linspace(-2, 2, 15), np.linspace(-2, 2, 15))
dst = np.sqrt(x*x + y*y)
sigma = 1
muu = 0.000
pyl.imshow(dst)
pyl.show()
```

# GPU-Accelerated Computing: Convolution



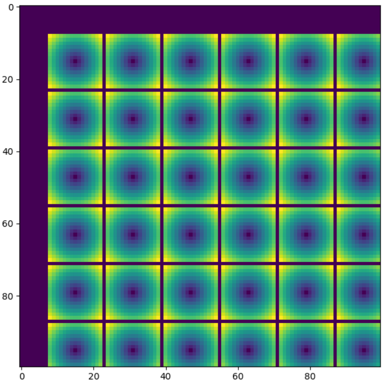
# GPU-Accelerated Computing: Convolution

We can now carry out the convolution on the CPU.

We don't have to write this convolution function ourselves, as it is very conveniently provided by SciPy.

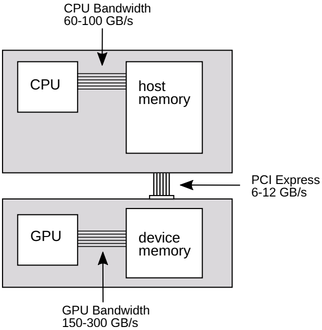
```python
from scipy.signal import convolve2d as convolve2d_cpu

convolved_image_using_CPU = convolve2d_cpu(deltas, dst)
pyl.imshow(convolved_image_using_CPU[0:100, 0:100])

pyl.show()
```

# GPU-Accelerated Computing: Convolution



# GPU-Accelerated Computing: Convolution

Although there is a physical connection between the CPU and the GPU, they do not share the same memory space. This image depicts the different components of CPU and GPU and how they are connected:



This means that an array created using NumPy is physically located in the main memory of the CPU, and thus visible to the CPU but not the GPU. It is not yet in GPU memory, so we need to copy the input image and the convolving function to the GPU, before we can execute any code on it.

# GPU-Accelerated Computing: Convolution

The arrays deltas and dst are in the CPU's RAM. Let's copy them to GPU memory using CuPy.

```
import cupy as cp

deltas_gpu = cp.asarray(deltas)
dst_gpu = cp.asarray(dst)
```

Execution of the convolution:

```
from cupyx.scipy.signal import convolve2d as convolve2d_gpu

convolved_image_using_GPU = convolve2d_gpu(deltas_gpu, dst_gpu)
```

# GPU-Accelerated Computing: NumPy on GPU

We saw above that we cannot execute routines from the `cupyx` library directly on NumPy arrays. In fact we need to first transfer the data from host to device memory.

This will lead to an error:

```
convolve2d_gpu(deltas, dst)
```

Vice versa, if we try to execute a regular SciPy routine (i.e. designed to run the CPU) on a CuPy array, we will also encounter an error.

```
convolve2d_cpu(deltas_gpu, dst_gpu)
```

# GPU-Accelerated Computing: NumPy on the GPU

The transfer function is mandatory for any CUDA program execution which is divided to 3 main steps:

- Copy the input data from host memory to device memory: Host-to-device transfer.
- Load the GPU program and execute.
- Copy the results from device memory to host memory: Device-to-host transfer.

```python
import cupy as cp

def transfer_compute_transferback():
    deltas_gpu = cp.asarray(deltas)
    dst_gpu = cp.asarray(dst)
    convolved_image_using_GPU = convolve2d_gpu(deltas_gpu, dst_gpu)
    convolved_image_using_GPU_copied_to_host = cp.asnumpy(convolved_image_using_GPU)
```

# Summary: Parallel Programming

We have seen various ways on how to speed up our programs:

Data Preprocessing is a step that is always recommended.

Further steps depend on the computing infrastructure available, and can include:

- Multiprocessing
- Multithreading
- usage of GPUs.

# An Outlook: C/C++

So far, we have worked with Python. In the next lecture, we will see how we can work with another programming language, C/C++.