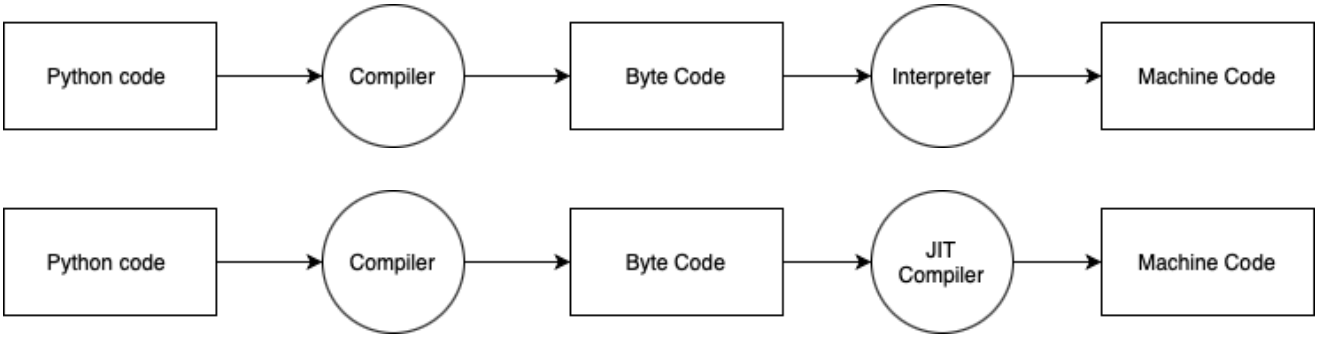
Background

- Numba was initially developed to optimize the inefficient use-cases of Numpy.
- Numpy uses multi-dimensional array (ndarray) object to store data.
- Python operators on ndarrays will trigger operations that are implemented in C and this is very efficient.
- Before Numba, NumPy users had to write Python C extensions to implement any custom computation in an efficient way.
- Numba is a **function-at-a-time** Just-in-Time **(JIT)** compiler for CPython.
- Numba lets users annotate a **compute-intensive** Python function for compilation without rewriting the code in a low-level language like C.

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

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

JIT Compiler



- Compiling happens at runtime (Just in time)
- Not ahead of time (AOT) like in C

How does Numba work

- The programmer adds a Numba decorator to the function.
- The decorator replaces the original Python function with a special object that just-in-time compiles the function when it is called the first time.

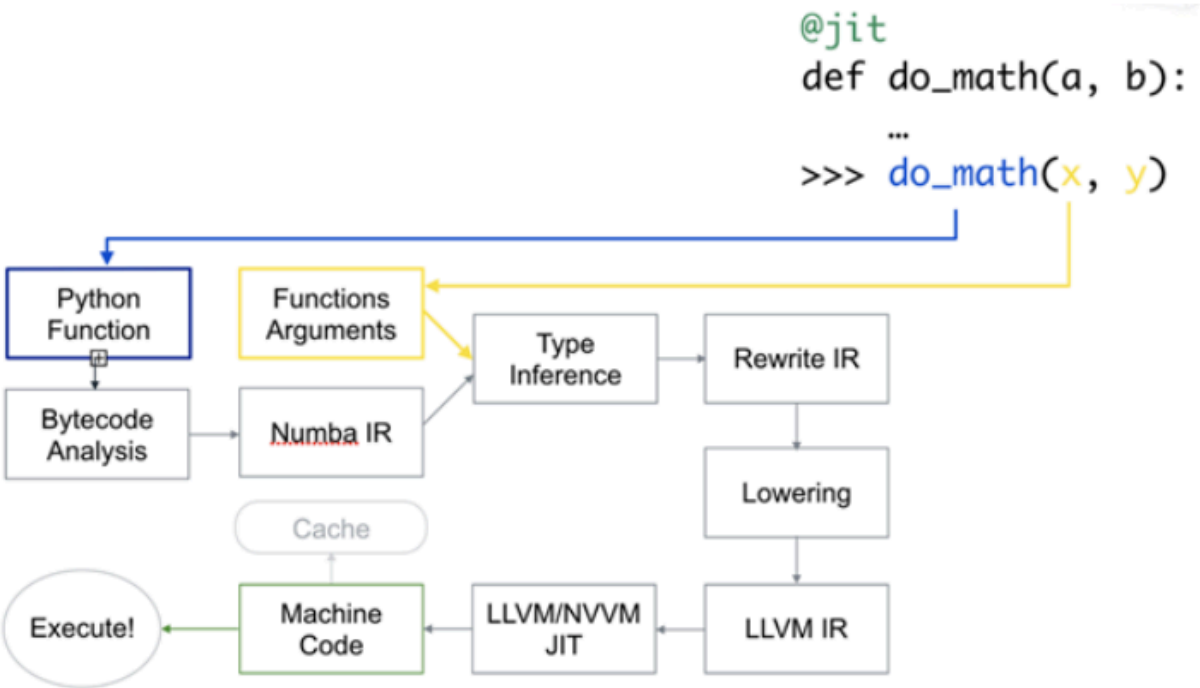
Import Required Libraries

This cell imports everything needed for the notebook:

- `numba` — the main Numba package for JIT compilation.
- `jit` — the core decorator that marks Python functions for JIT compilation.
- `int32`, `float64` — Numba type objects for explicit argument/return type declarations.
- `prange` — a parallel-capable `range` replacement that Numba can distribute across CPU threads when `parallel=True` is set.
- `vectorize` — a decorator for creating NumPy-style universal functions (ufuncs) from scalar Python functions.
- `cuda` — Numba's CUDA GPU backend (imported for completeness; used in GPU notebooks).
- `numpy as np` — for array creation and numerical operations.
- `math` — the standard Python math library; Numba natively supports its functions inside JIT-compiled code.

In [ ]:
```
import numba
from numba import jit, int32, prange, vectorize, float64, cuda
import numpy as np
import math
```

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



Basic @jit Decorator

The `@jit` decorator tells Numba to compile `f(x, y)` **the first time it is called**, not at decoration time. Numba inspects the actual argument types at that first call, generates optimized LLVM IR, and compiles it to native machine code. Subsequent calls with the same types reuse the compiled code instantly. This is the simplest Numba usage — add the decorator and leave the code unchanged.

In [ ]:
```
@jit
def f(x, y):
```

```
    return x + y
```

First Call — Triggers Compilation (Integer Types)

Calling `f(2, 3)` with two Python integers triggers JIT compilation for the `(int64, int64)` type signature. The compilation happens transparently; the result `5` is printed. This compiled version is cached internally so future calls with integer arguments incur no compilation overhead.

In [ ]:
```
print(f(2, 3)) # This generates one compiled code
```

Inspect the Generated LLVM IR

`f.inspect_llvm()` returns a dictionary mapping each compiled type signature to its **LLVM Intermediate Representation (IR)** — the low-level code Numba produces before passing it to the LLVM backend for final machine-code generation. Each key is a tuple of argument types (e.g., `(int64, int64)`), and each value is the full LLVM IR string. This is a powerful debugging tool for understanding exactly what Numba compiled and for diagnosing performance issues.

In [ ]:
```
# we can see what LLVM is doing with the function
for k, v in f.inspect_llvm().items():
    print(k, v)
```

Second Call — New Compilation for String Types

Calling `f('2', '3')` with strings causes Numba to generate a **second, separate** compiled version for the `(unicode_type, unicode_type)` signature. Numba now holds two specializations: one for integers, one for strings. This **type specialization** is central to Numba's JIT model — each unique combination of argument types gets its own optimized compiled version.

In [ ]:
```
print(f('2', '3')) # This generates another compiled code
```

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

Cell Explanation: Declaring a Fixed Type Signature

Passing `int32(int32, int32)` directly to `@jit` specifies that `f_1` should be compiled **only** for 32-bit integer inputs returning a 32-bit integer. This is called **eager compilation** — the function is compiled at decoration time rather than lazily on the first call. Fixing the signature prevents Numba from generating multiple specializations and also serves as explicit documentation of the function's intended types.

In [ ]:
```
@jit(int32(int32, int32))
def f_1(x, y):
    return x + y
```

Valid Call with Fixed Signature

`f_1(2, 3)` succeeds because both arguments are compatible with `int32`. Numba converts the Python integers to native 32-bit integers and executes the pre-compiled function, returning `5`.

In [ ]:
```
print(f_1(2, 3))
```

Invalid Call — Type Mismatch Error

Because `f_1` was eagerly compiled for `int32` arguments only, passing strings `'2'` and `'3'` raises a `TypeError`. Unlike plain `@jit`, a fixed type signature removes Numba's ability to create a new

specialization for a different type or fall back to object mode. This strict type enforcement is useful for catching type bugs early in development.

In [ ]:
```
print(f_1('2', '3')) # generates error
```

JIT-Compiled Functions Calling Each Other

This demonstrates **function call inlining** in Numba. Both `square` and `hypot` are JIT-decorated. When `hypot` is compiled, Numba recognizes that `square` is also a JIT function and can inline its compiled body directly into `hypot`'s machine code, eliminating all Python function-call overhead. This produces highly efficient native code for nested computation hierarchies — comparable to what a C compiler would generate.

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

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

In [ ]:
```
print(hypot(4, 3))
```

Performance

Pure Python / NumPy Baseline Function

`without_numba` computes the **matrix trace with tanh** then broadcasts it back onto the array — using a plain Python `for` loop over the diagonal.

Key points:

- The explicit Python loop is slow because every iteration carries Python interpreter overhead (type checking, bytecode dispatch, reference counting, etc.).
- Calling `np.tanh` on a **scalar** `a[i, i]` is less efficient than calling it on a full array, because it incurs NumPy's per-call overhead each iteration.

This serves as the **performance baseline** against which the Numba version will be compared.

In [ ]:
```
def without_numba(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace
```

Numba-Compiled Equivalent with nopython=True

`with_numba` contains **identical logic** to `without_numba`, but `@jit(nopython=True)` instructs Numba to compile it entirely without the Python interpreter.

Key points:

- **`nopython=True`** is the recommended mode — it forces Numba to generate fully native code. If any operation cannot be compiled natively, Numba raises an error immediately rather than silently falling back to slow object mode.
- Numba excels at explicit Python `for` loops, replacing Python iteration with a tight native loop.
- `np.tanh` and NumPy broadcasting (`a + trace`) are both fully supported inside nopython mode.

- The equivalent shorthand decorator is `@njit` (nopython JIT).

In [ ]:
```
@jit(nopython=True) # Set "nopython" mode for best performance, equivalent to @njit
def with_numba(a): # Function is compiled to machine code when called the first time
    trace = 0.0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting
```

Create a Small 10x10 Test Array

`np.arange(100).reshape(10, 10)` creates a 1-D array of integers 0–99 and reshapes it into a 10×10 matrix. This small array is used for the first timing comparison so results appear quickly — though the small size means compilation overhead will dominate the Numba timing.

In [ ]:
```
x = np.arange(100).reshape(10, 10)
```

Time the Pure Python Version (Small Array)

The `%%time` cell magic measures **wall-clock time** and CPU time for the entire cell. Running `without_numba(x)` on the 10×10 array gives a baseline timing. For small arrays, the computation is fast

In [ ]:
```
%%time
print(without_numba(x))
```

Time the Numba Version (Small Array — Includes Compilation Cost)

The **first call** to `with_numba(x)` includes Numba's JIT compilation time, which can take hundreds of milliseconds. For a small array, this means the Numba version appears *slower* than pure Python on this first call. This is expected and normal — the compilation is a **one-time cost**. Subsequent calls (as seen with the large array below) will be dramatically faster.

In [ ]:
```
%%time
print(with_numba(x))
```

Create a Large 1000x1000 Test Array

`np.arange(1000000).reshape(1000, 1000)` creates a 1,000,000-element array reshaped into a 1000×1000 matrix. With 1000 diagonal elements to iterate over, the Python interpreter overhead in the loop becomes a significant fraction of total runtime — making this the meaningful performance comparison.

In [ ]:
```
x = np.arange(1000000).reshape(1000, 1000)
```

Time the Pure Python Version (Large Array)

Running `without_numba(x)` on the 1000×1000 array is noticeably slower than the small-array case. The Python interpreter must now execute 1000 loop iterations, with all associated overhead per step. This timing illustrates the **scalability problem** of pure Python loops on large data.

In [ ]:
```
%%time
print(without_numba(x))
```

Time the Numba Version (Large Array — No Compilation Overhead)

Since `with_numba` was already compiled in Cell 27, this call reuses the cached native code with **zero compilation overhead**. The speedup vs. Cell 29 illustrates Numba's core value proposition: the same

Python code, running at near-C speed. The performance gap grows with array size because computation time increases while the fixed compilation cost stays zero.

In [ ]:
```
%%time
print(with_numba(x))
```

Compilation options

nopython

- nopython mode
  - compilation without using Python C API (faster)
  - Entirely bypass the Python interpretter.
- object mode
  - compilation using Python C API (slower).
- If nopython mode fail numba will automatically fallback to object mode.
- so its a good practice to compile everything in nopython mode

Redefining f with Explicit nopython=True

This redefines `f` using `@jit(nopython=True)` — the same as `@njit`. If Numba encounters any Python object it cannot translate (arbitrary dictionaries, unsupported builtins, etc.), it raises a `TypingError` immediately at the first call rather than silently degrading to slow object mode. **This is the recommended practice** for production Numba code: fail loudly on compilation errors, never silently accept worse performance.

In [ ]:
```
# nopython=True will force numba not to fallback to the object mode.
# This will force an error if type inference does not work

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

nogil

Releasing the GIL with nogil=True

`nogil=True` instructs Numba to **release the Python Global Interpreter Lock (GIL)** while the compiled function runs. Normally the GIL prevents true multi-threading in CPython, but Numba-compiled code operates entirely on native types and memory without touching Python objects, so releasing the lock is safe.

This enables true parallelism when the same function is called from multiple Python threads simultaneously — something impossible with standard Python code. The caveat is important: releasing the GIL while threads share mutable data (like NumPy arrays) requires careful synchronization to avoid race conditions and data corruption.

In [ ]:
```
# A numba compiled a code operates only on the native types.
# So it is not necessary to hold the GIL

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

```
# beware: This can cause syncronization issues
```

cache

Caching Compiled Code to Disk

`cache=True` tells Numba to **save the compiled machine code to disk** (in a `__pycache__` subdirectory) after the first compilation. On subsequent runs of the script or notebook, Numba loads the cached binary and skips recompilation entirely — eliminating startup latency. This is especially valuable for scripts or services that are run repeatedly with the same argument types.

In [ ]:
```
# The chances are you call the same function again and again with the same argument type
# So you can cached the compiled code.

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

Automatic parallelization

Enabling Automatic Parallelization

`parallel=True` activates Numba's **automatic parallelization pass**. Numba analyses the function body and identifies loops and array operations that can safely execute in parallel across multiple CPU cores. Under the hood it replaces eligible loops with a parallel thread pool (using Intel TBB or a similar backend). No changes to the function code are required — Numba handles all threading automatically. Note: not all loops can be parallelized; Numba will only parallelize those it can prove are safe.

In [ ]:
```
#automatic parallelization

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

Reduction Loop Without Parallelism (Serial Baseline)

This function performs an **element-wise array reduction**: it multiplies `result1` by `tmp` a total of `n` times. Although `prange` is used (the parallel-capable range), `parallel=True` is **not** set on this function, so `prange` behaves identically to regular `range` — all iterations run sequentially on a single thread. This is the **serial baseline** used to measure the benefit of `parallel=True` in the next cell.

In [ ]:
```
@jit(nopython=True)
def reduction_without_parallel(n):
    shp = (13, 17)
    result1 = 2 * np.ones(shp, np.int_)
    tmp = 2 * np.ones_like(result1)

    for i in prange(n):
        result1 *= tmp

    return result1
```

Timing the Serial Reduction (n=10)

Times `reduction_without_parallel(10)` with a small iteration count. The first call also compiles the function. With only 10 iterations the computation is trivial, so the measured time mainly reflects compilation and function-call overhead.

In [ ]:
```
%%time
reduction_without_parallel(10)
```

Parallel Reduction with prange

This function is identical to `reduction_without_parallel` except `parallel=True` is added to the decorator. Now Numba treats `prange` as a **parallel for-loop**, distributing iterations across all available CPU cores. Numba automatically handles the **array reduction** of `result1` across threads safely, because it recognizes the `*=` pattern as a commutative reduction operation and inserts the necessary thread-safe accumulation.

In [ ]:
```
@jit(nopython=True, parallel=True)
def reduction_with_parallel(n):
    shp = (13, 17)
    result1 = 2 * np.ones(shp, np.int_)
    tmp = 2 * np.ones_like(result1)

    for i in prange(n):
        result1 *= tmp

    return result1
```

Timing the Parallel Reduction (n=10 — Expect Overhead)

With only 10 iterations, the parallel version will likely be **slower** than the serial version due to thread creation and synchronization overhead. Parallelism has fixed costs; it only pays off when the work per iteration is large enough to amortize those costs. This cell demonstrates that `parallel=True` is not universally beneficial — the workload must be substantial enough to justify the threading overhead.

In [ ]:
```
%%time
reduction_with_parallel(10)
```

Timing the Serial Reduction (n=10,000,000 — Meaningful Workload)

With 10 million iterations, the computation is expensive enough that wall-clock time is significant on a single CPU core. This is the meaningful baseline comparison: a heavyweight loop running purely sequentially.

In [ ]:
```
%%time
reduction_without_parallel(10000000)
```

Timing the Parallel Reduction (n=10,000,000 — Real Speedup)

With 10 million iterations, the parallel version distributes work across all available CPU cores and should show a clear speedup proportional to the number of cores (minus synchronization overhead). This cell demonstrates where `parallel=True` delivers real-world benefit: large, compute-heavy loops where the work-per-thread far outweighs threading costs.

In [ ]:
```
%%time
reduction_with_parallel(10000000)
```

ufunc

- In NumPy universal function (ufunc) is a function that operates on ndarrays in an element-by-element fashion

Built-in NumPy ufunc Example

This shows a built-in NumPy **universal function (ufunc)**: `np.add` applies element-wise addition to two lists, returning `[5, 7, 9, 11]`. NumPy ufuncs automatically handle broadcasting, type promotion, and arbitrary array shapes. They run in compiled C code and are very fast. The next cells motivate why we need `@vectorize` — to create **custom** ufuncs that are equally fast without writing C code.

In [ ]:
```
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = np.add(x, y)

print(z)
```

- Creating a ufunc that operates on a ndarray of a particular type is not straight forward

A Scalar Math Function Incompatible with Arrays

`sinacosb` computes `sin(a) × cos(b)` using Python's `math` module. The `math` functions operate **only on scalar values**, not on NumPy arrays. Calling `sinacosb(array, array)` will fail with a `TypeError`. Traditionally, making this work element-wise over arrays required writing a NumPy C extension — Numba's `@vectorize` decorator makes this trivial, as shown below.

In [ ]:
```
def sinacosb(a, b):
    return math.sin(a) * math.cos(b)
```

Create Large Float64 Arrays for Benchmarking

Creates two 1-D NumPy arrays of 10 million 64-bit floats:

- `a` — filled with 1.0 values (`np.ones`).
- `b = 2 * a` — filled with 2.0 values.

The `dtype=np.dtype('f8')` specifies 64-bit floating point (`f8` = 8 bytes = float64). These large arrays are used to benchmark the vectorized `sinacosb_vect` function at scale.

In [ ]:
```
n = 10000000
a = np.ones(n, dtype=np.dtype('f8'))
b = 2*a
```

Confirm the Scalar Function Fails on Arrays

Calling `sinacosb(a, b)` with NumPy arrays confirms the expected error: `math.sin` cannot handle an ndarray and raises a `TypeError`. This cell explicitly demonstrates the limitation that `@vectorize` in the next cell solves.

In [ ]:
```
sinacosb(a, b) # generates error
```

Creating a Custom ufunc with @vectorize

The `@vectorize([float64(float64, float64)])` decorator transforms `sinacosb_vect` into a proper NumPy **universal function**:

- The type list `[float64(float64, float64)]` specifies that the function takes two `float64` scalars and returns a `float64` scalar.
- Numba generates the looping infrastructure automatically, applying the scalar function to every element of the input arrays with the inner loop compiled to native machine code.
- The resulting `sinacosb_vect` behaves exactly like any built-in NumPy ufunc: it supports broadcasting, works on arrays of any shape, and runs at compiled speed.

In [ ]:
```
#Numba makes this process easy

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

Calling the Vectorized Custom ufunc

`sinacosb_vect(a, b)` now works correctly on the 10-million-element arrays, returning a new array where element `i` equals `sin(a[i]) × cos(b[i])`. The entire computation runs in compiled native code — much faster than a Python-level loop, and without any manual C extension code.

In [ ]:
```
sinacosb_vect(a, b)
```