Vectorization

Lecturer: Bo Li TA: Chen Gao

2026-03-03

Table of contents

1 Vectorization

In Python, vectorization (primarily through NumPy) speeds up computation by using fast low-level code to process large amounts of data. Pandas is also built on top of NumPy and provides similar performance benefits.

Let's start by importing the necessary libraries:

```
from time import time
import numpy as np
```

For example, consider adding an integer to every element in a Python list:

```
l = list(range(100_000_000))

start = time()
for i in range(len(l)):
    l[i] += 17
loop_time = time() - start
print(f"Elapsed: {loop_time:.2f} seconds")
```

```
Elapsed: 3.54 seconds
```

In CPython, a list of integers is actually a sequence of pointers to Python objects. From CPython's perspective, these are generic Python objects, not raw machine integers. Adding an integer to each list element involves:

- Looking up the type of each item.
- Looking up the addition function for that type.
- Calling that function with two Python objects.
- Converting both objects to machine-level integers.
- Performing the integer addition.
- Wrapping the result back into a Python object.

Even advancing to the next element in a Python list has overhead: iterator lookups, indexing operations, and object-to-machine-integer conversions. That is a lot of work.

By contrast, a NumPy integer array is essentially a contiguous array of machine integers. Adding a constant to every element therefore requires very few CPU instructions, and after setup, it behaves similarly to equivalent C code.

Here is the NumPy version:

```
l = np.array(range(100_000_000), dtype=np.uint64)

start = time()
l += 17
vec_time = time() - start
print(f"Elapsed: {vec_time:.2f} seconds")
print(f"Speedup: {loop_time / vec_time:.2f}x")
```

```
Elapsed: 0.06 seconds
Speedup: 55.88x
```

As expected, NumPy is much faster.

1.1 Limitations of Vectorization

Vectorization can significantly improve performance, but it is not perfect. Here are three common limitations.

1.1.1 1. Unnecessary Large Memory Allocations

Suppose you want to compute the mean distance of array elements from zero. One approach is to take the absolute value of each element and then compute the mean:

```
def mean_distance_from_zero(arr):
    return np.abs(arr).mean()
```

This works, and both operations are vectorized and fast. However, it allocates a new intermediate array for absolute values. If the original array is 1000MB, you may allocate another 1000MB. You can overwrite the original array to save memory, but you may need to keep it for other reasons.

1.1.2 2. Only Supported Operations Are Fast

Vectorization requires low-level machine code to drive the loop and perform operations. As soon as you fall back to Python loops, performance drops. For the same `mean_distance_from_zero()` example, a more memory-efficient implementation is:

```
def mean_distance_from_zero(arr):
    total = 0
    for i in range(len(arr)):
        total += abs(arr[i])
    return total / len(arr)
```

But this returns to Python-level looping, so it is slower. This is one reason NumPy implements so many operations internally: whenever execution falls back to Python, speed suffers. In compiled languages, compilers may also apply cross-operation optimizations that are hard to achieve with separate vectorized calls.

1.1.3 3. Vectorization Mainly Helps Uniform Batch Operations

Sometimes code is slow because it applies the same operation to many items of the same type. In that case, vectorization works very well.

In other cases, performance bottlenecks do not match this pattern. Then vectorization is less helpful, and you need other optimization techniques.