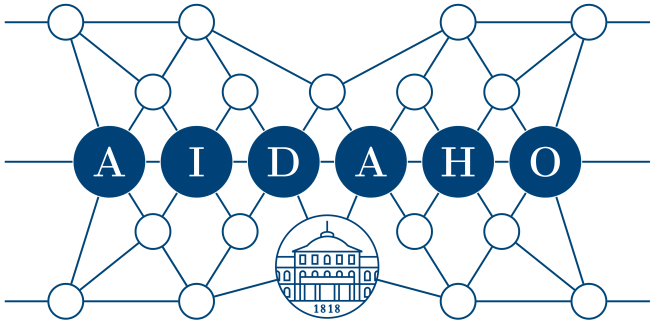
UNIVERSITY OF HOHENHEIM

Computational Science Hub

Kommunikations-, **I**nformations und **M**edienzentrum



AI & Data Science Certificate Hohenheim

Speed Up Your Code

Algorithm, Parallelization, GPUs ...

Dr. Johannes Bleher[1] Dr. Konstantin Kuck [2] Benedikt Otto [2]

[1]CSH [2]KIM

University of Hohenheim

.

Algorithmic Improvements

The naive method is not always the right way

The first step to optimizing your code is to choose the **right algorithm** for the problem.

Naive algorithms are quickly implemented but pose several problems:

- Low Performance
- Inefficient use of memory
- Numerical instability
- Not parallelizable

Example: Calculating the Variance

Scenario: We are analyzing a stream of values and want to gradually calculate their unbiased variance.

The variance can be calculated through its definition:

$$\bar{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad s_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x}_n)^2$$

However, using the definition of variance naively, we will run into several problems:

- **Performance** - For every added value, we need to re-calculate the mean and variance using the whole dataset
- **Memory** - We need to keep all the previous values in RAM.
- **Numerically unstable**

```python
def var_naive(stream: Generator[float]) -> float:
    s2, mean = 0.0, 0.0
    cached_values = []  # Remember the previous values.

    for val in stream:
        cached_values.append(val)
        count = len(cached_values)

        sum_values = 0.0
        for old_val in cached_values:
            sum_values += old_val
        mean = sum_values / count

        if count >= 2:
            sum_dev = 0.0
            for old_val in cached_values:
                sum_dev += (old_val - mean)**2
            s2 = sum_dev / (count - 1)

    return s2
```

Example: Welford's Algorithm

A better way to calculate the variance of a population is via the following recursion relations:

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n},$$
$$M_{2,n} = M_{2,n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n),$$
$$s_n^2 = \frac{M_{2,n}}{n-1}$$

Advantages:

- **Performance** – Updating the variance only requires a constant number of operations.
- **Memory-Efficient** – No need to keep old values in memory. (Online-Algorithm)
- **Numerically stable** for large values of n.
- **Parallelizable** ($\Rightarrow$ Chan's Algorithm)

```python
def var_welford(stream: Generator[float]) -> float:
    s2, mean, m2 = 0.0, 0.0, 0.0

    count = 0
    for val in stream:
        count += 1
        mean_new = mean + (val - mean) / count
        m2 = m2 + (val - mean) * (val - mean_new)

        mean = mean_new
        if count > 2:
            s2 = m2 / (count - 1)

    return s2
```

Benchmark

```
Microbenchmark (10.000 values, sigma=1.0, mu=1.0):
Numpy:          Variance: 1.0305436657528968
                Time: 0.561ms.
Naive method:   Variance: 1.0305436657529023
                Time: 3274.047ms.
Welford's:      Variance: 1.0305436657528928
                Time: 1.154ms.
```

```
Microbenchmark (10.000 values, sigma=1.0, mu=1e15):
Numpy:          Variance: 1.047598509850985
                Time: 0.515ms.
Naive method:   Variance: 2.028527852785284
                Time: 3145.399ms.
Welford's:      Variance: 1.0483548354835484
                Time: 1.112ms.
```

Parallelization

Why Parallelization?

Another approach to reduce the compute time of a problem is parallelization.

Mental model

- A problem is **job** to be done,
- and can be decomposed into **sub-tasks**:
- The CPU is the **workshop**,
- where each CPU core is a **worker**.

Basically, this idea builds on the multicore architecture of modern processors (CPU).

A Mental Model for CPU Parallelization

Key Implications

- Problem must be decomposable into **(independent) sub-tasks**
- Communication: Instructions and data need to be sent to workers, and workers' results need to be collected $\rightarrow$ Overhead
- Speed-up benefit is limited by the availability of shared resources

Typically, **some parts of the problem are inherently sequential** in real-world settings

When can parallelization be beneficial?

Parallelization Works Well When

- Problem can be decomposed into many similar, independent sub-tasks
- Each sub-task takes a meaningful compute time
- Workers rarely need to coordinate or wait

Parallelization Works Poorly When

- Sub-tasks depend on results from other sub-tasks
- Sub-tasks are computationally very small or uneven in size
- Workers constantly need shared resources ($\rightarrow$ Do not use start too many workers)

Bootstrap example: Setup

```
library(future)
library(future.apply)
library(microbenchmark)

# ignore this line for now
options(future.rng.onMisuse="ignore")

# Load dataset
Default <- ISLR::Default



# Parallel bootstrap using future.apply
# with base R resampling
plan(multicore, workers = 4)
set.seed(123)

# Number of bootstrap samples
N <- 2000
```

Bootstrap example: Bootstrap function

```r
# Bootstrap function
boot_fun <- function(i) {
  # Resample the data with replacement
  boot_idx <- sample(1:nrow(Default), replace = TRUE)
  boot_sample <- Default[boot_idx, ]

  # Estimate model based on bootstrap sample
  boot_glm <- glm(default ~ student + balance + income,
                  data = boot_sample, family = binomial)

  # return coefficients
  coef(boot_glm)
}
```

Bootstrap example: Wrapper functions for benchmark

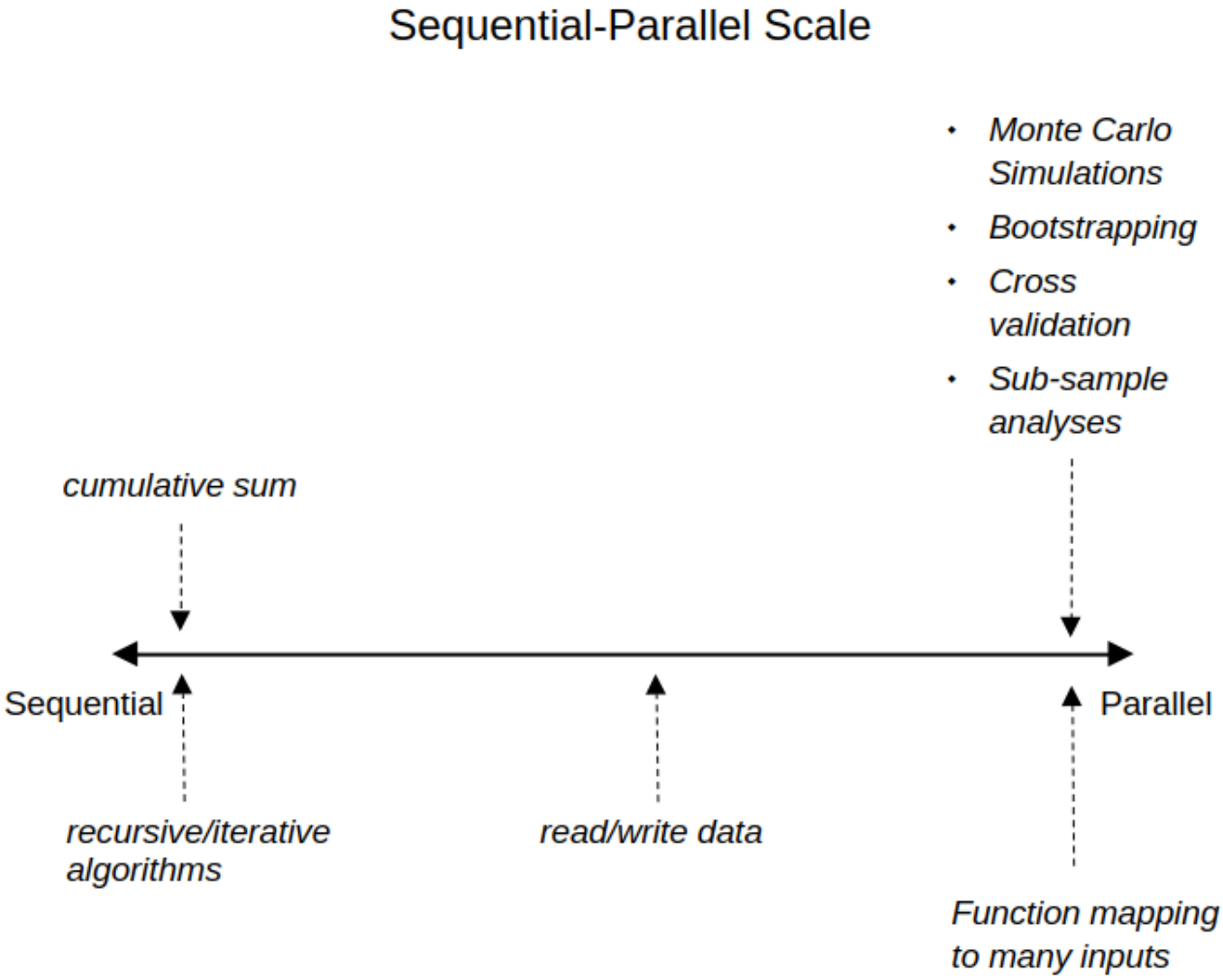
```
# Wrapper function for sequential bootstrap
boot_seq <- function() {
  lapply(1:N, boot_fun)
}

# Wrapper function for parallel bootstrap
boot_par <- function() {
  future_lapply(1:N, boot_fun)
}
```

Bootstrap example: Benchmark result

```
microbenchmark(
    sequential = boot_seq(),
    parallel   = boot_par(),
    times = 5L
)

plan(sequential)
```

```
Unit: seconds
       expr      min       lq     mean   median       uq      max neval
 sequential 66.03070 66.98823 72.52299 67.80510 69.54360 92.24731     5
   parallel 24.40395 24.44200 25.93027 24.55188 24.58525 31.66825     5
```

Parallelization potential of different methods/approaches



Parallelization potential: Checklist

1. Can the problem be decomposed into independent sub-tasks with no interaction?
   - absence of sequential dependence across parallelized sub-tasks
   - sub-tasks can be executed in any order
2. Is the computational cost per sub-tasks sufficiently large?
   - sub-tasks should have a runtime of more than a few seconds
   - very small tasks often run slower in parallel ($\rightarrow$ Overhead)
3. What is the dominant resource?
   - CPU-bound $\rightarrow$ good candidate
   - I/O-heavy (disk, network) $\rightarrow$ limited benefit
   - memory-bound $\rightarrow$ risk of slowdown

Parallelization potential: Checklist

1. Other—programming environment specific—approaches for efficient coding been used?

   In R, for instance:

   - avoidance of object-growing (pre-allocation, lists)
   - use of vectorization
   - functional programming
   - data management: `data.table`
   - just-in-time (JIT) compilation of functions
   - use of C++ for functions (`Rcpp`)

Some remarks

- Parallelization requires a profound understanding of the specific problem and the programming environment
- Algorithmic and other (programming environment specific) approaches for efficient coding should be considered before parallelization
- The specific implementation of parallelization of sub-tasks depends on the programming environment
- It should always be confirmed that code supposed to run in parallel consumes more than one CPU or core (Task Manager, `htop`)
- You need to account for the parallel setting in the context of random number generation

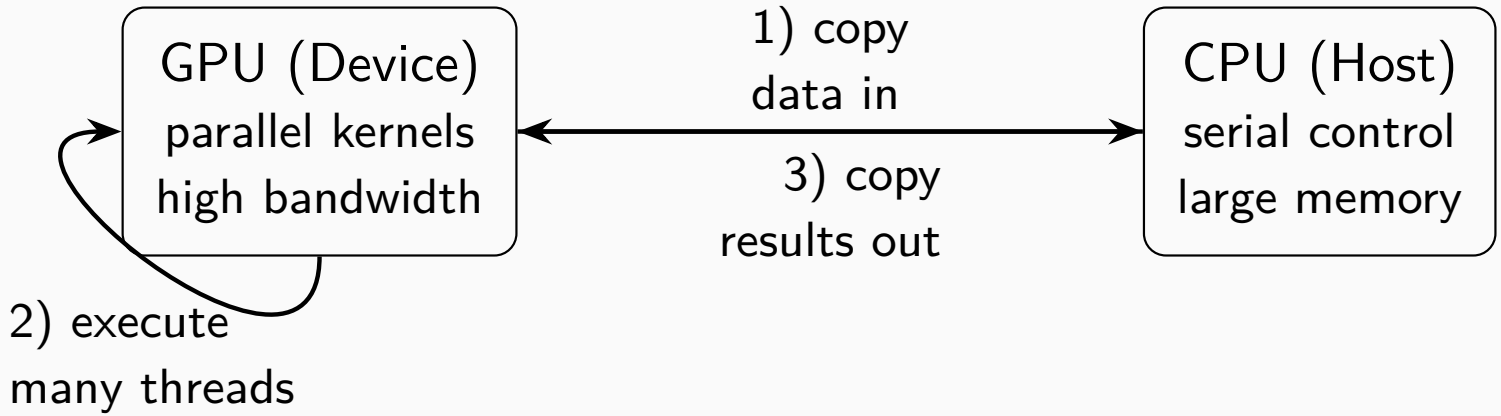
GPUs

Why GPUs?

- **Data parallelism** – same operation over many elements.
- CPUs are **latency-optimized** (few fast threads, large caches)
- GPUs are **throughput-optimized** (many threads, high memory bandwidth, latency hiding via parallelism).
- Practical compute acceleration by **offloading only compute-intensive jobs**, rest of application on CPU.

Working Module

Mental model



Rule of thumb: GPUs shine
when computation per byte moved is high
and work can be parallelized.

GPU Programming in Python: The Landscape

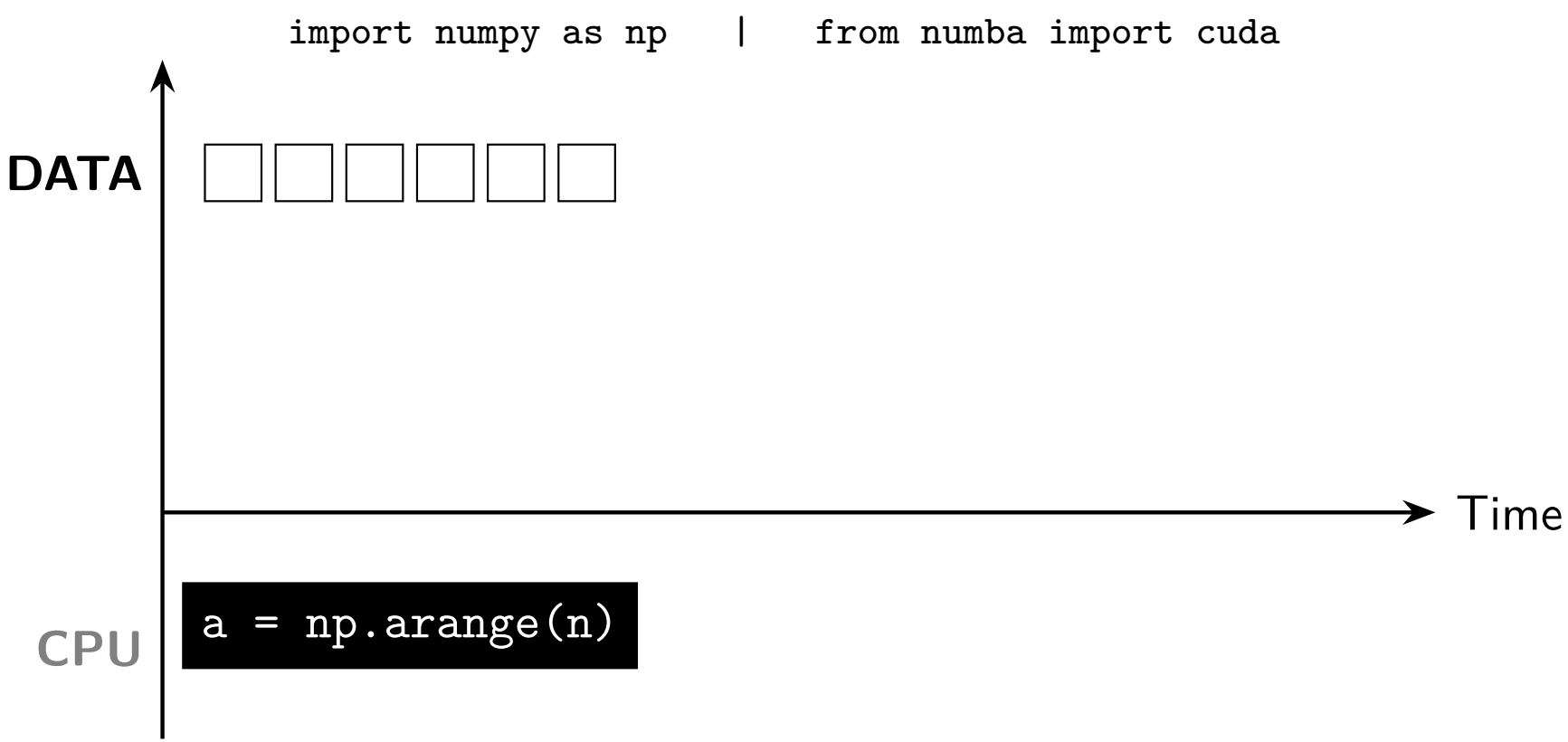
Two complementary ways to use GPUs

- **Write your own GPU code** (custom kernels, full control)
- **Use GPU-accelerated libraries** (high-level, productive)

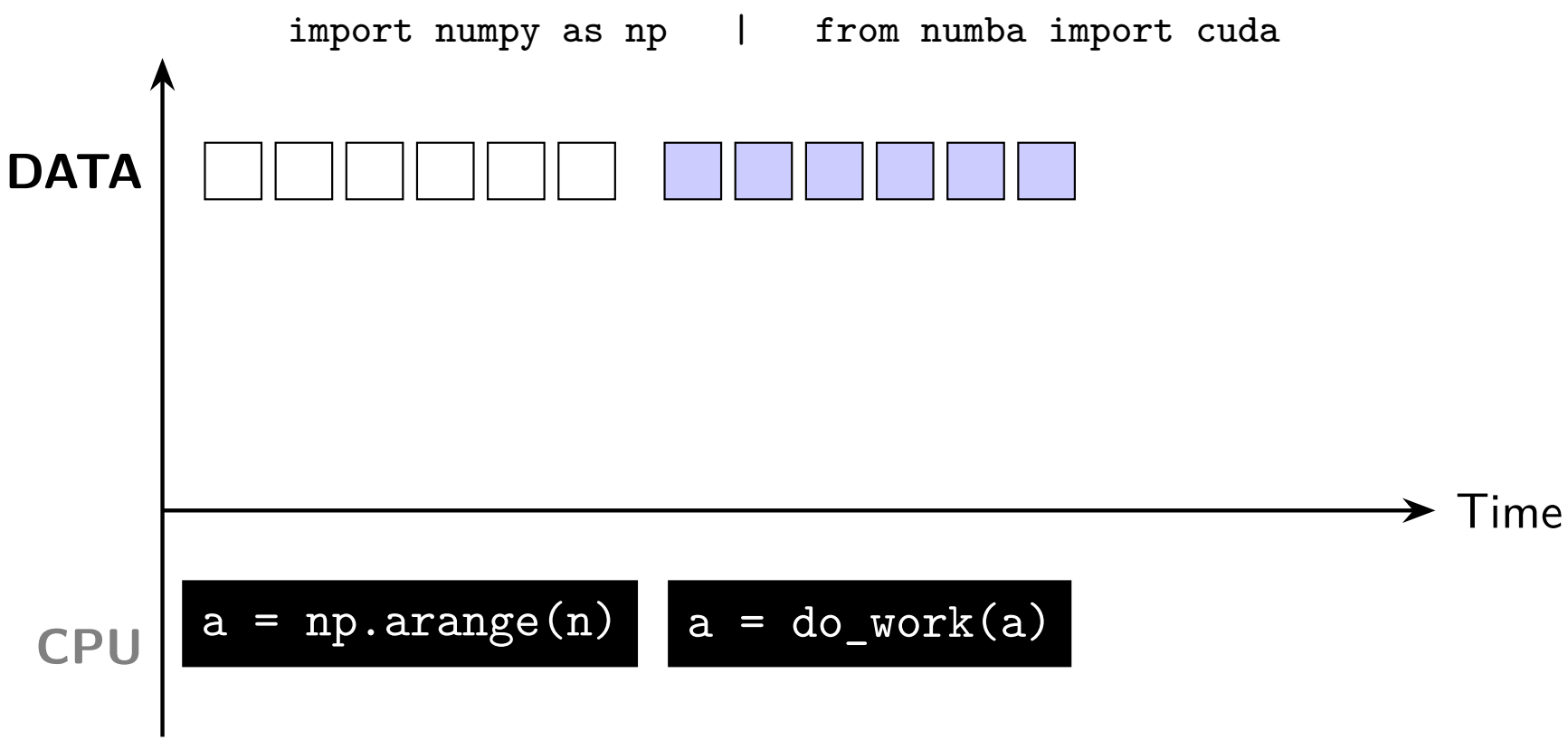
Core packages and where they fit

```
Low-level / kernels:
  numba.cuda                                  (write custom GPU kernels)

Array computing (NumPy-like):
  cupy                                          (GPU arrays, math, FFTs)

Tabular data (pandas-like):
  cudf                                                  (GPU DataFrames)

Machine learning (scikit-learn-like):
  cuml                                       (regression, clustering, ML)
```

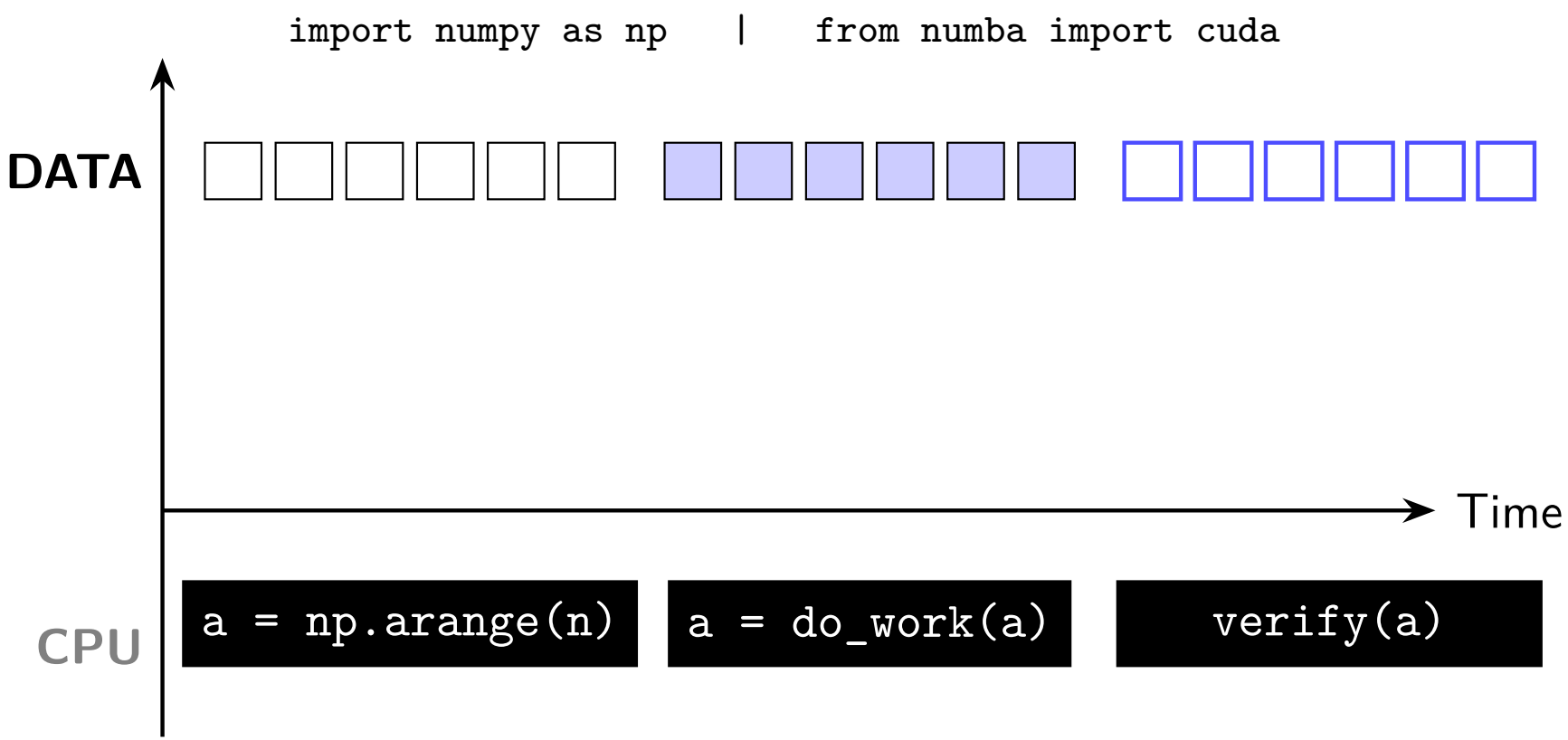
CPU-only execution



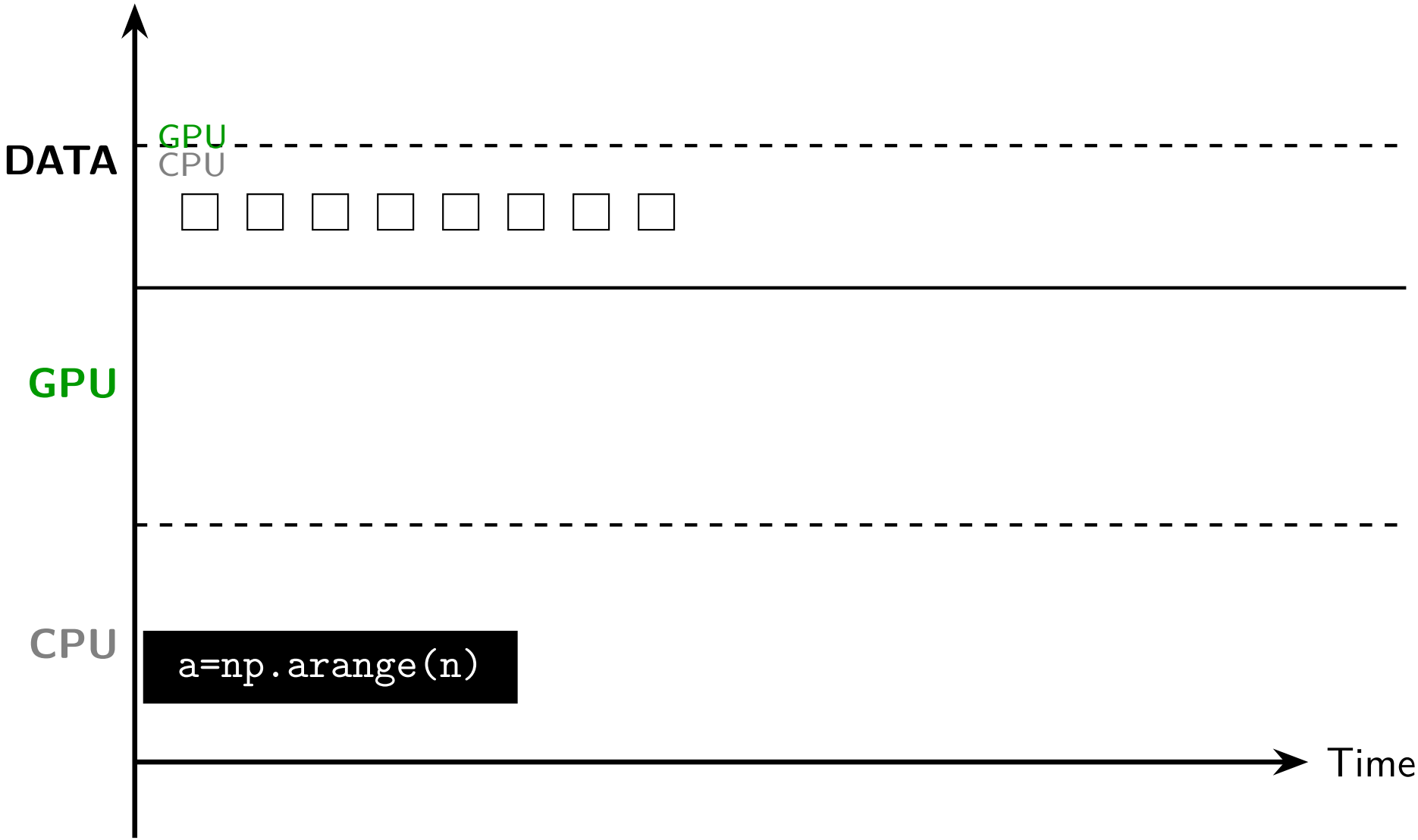
CPU-only execution



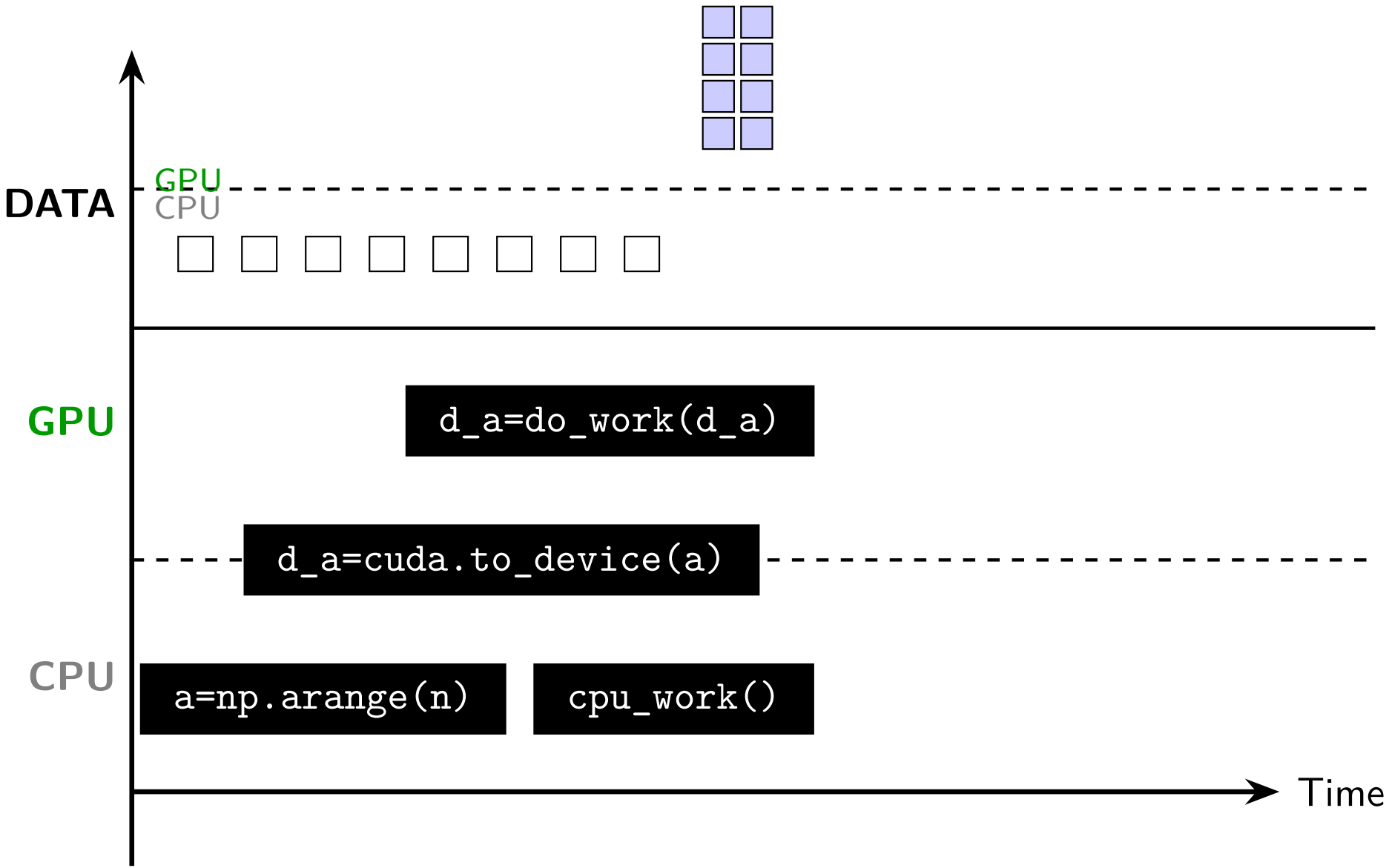
CPU-only execution



CPU + GPU execution (with synchronization)



CPU + GPU execution (with synchronization)



CPU + GPU execution (with synchronization)