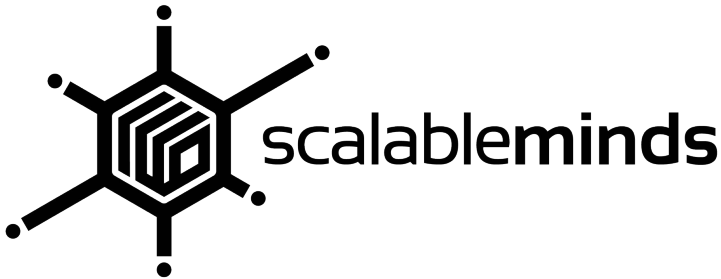
Profile, Optimize, Repeat
One Core Is All You Need™

Jonathan Striebel & Valentin Nieper

EuroPython2024

Hi, we're

Valentin Nieper & Jonathan Striebel



aignostics

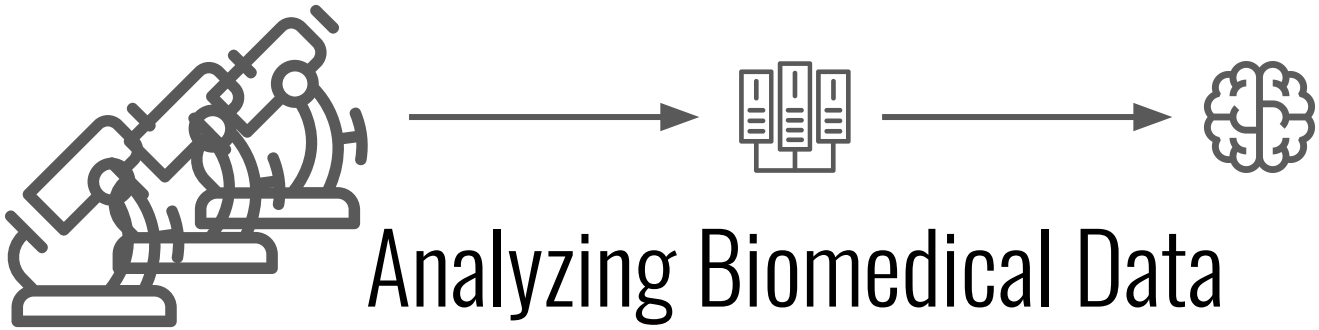
@jostriebel

codical.org

Data Science Workloads Grow

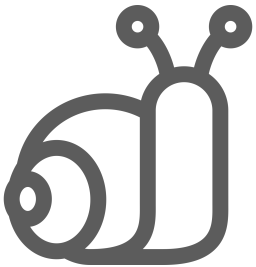
Analyzing Biomedical Data

Data Science Workloads Grow

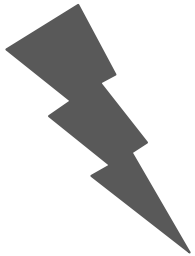


Analyzing Biomedical Data

even^more

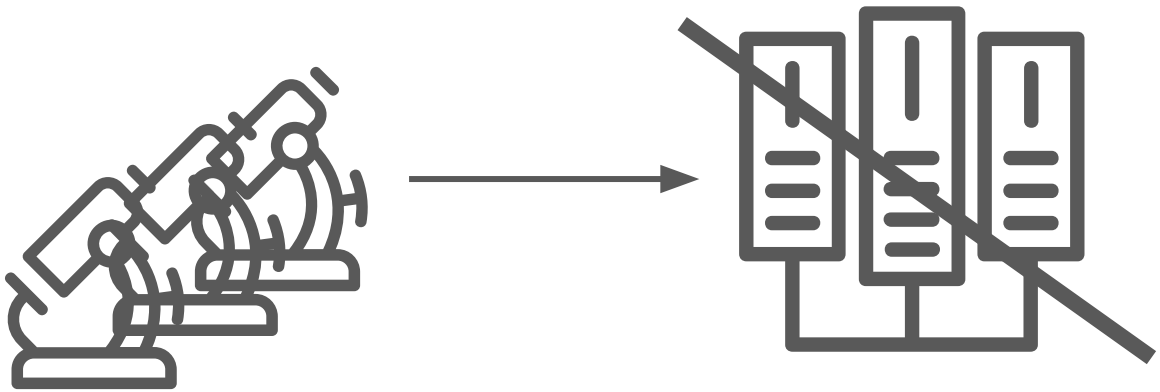


Too slow



OOM :-(

Data Science Workloads Grow



Don’t scale up/out (yet)

Data Science Workloads Grow

- Single core improvements **pay off, also when parallelized** later
- Parallelization **needs resources**

  (cores, memory, cluster-nodes, money, time)
- Code may be **hard** / **impossible** to parallelize

  (e.g. when reducing results from a map-reduce)

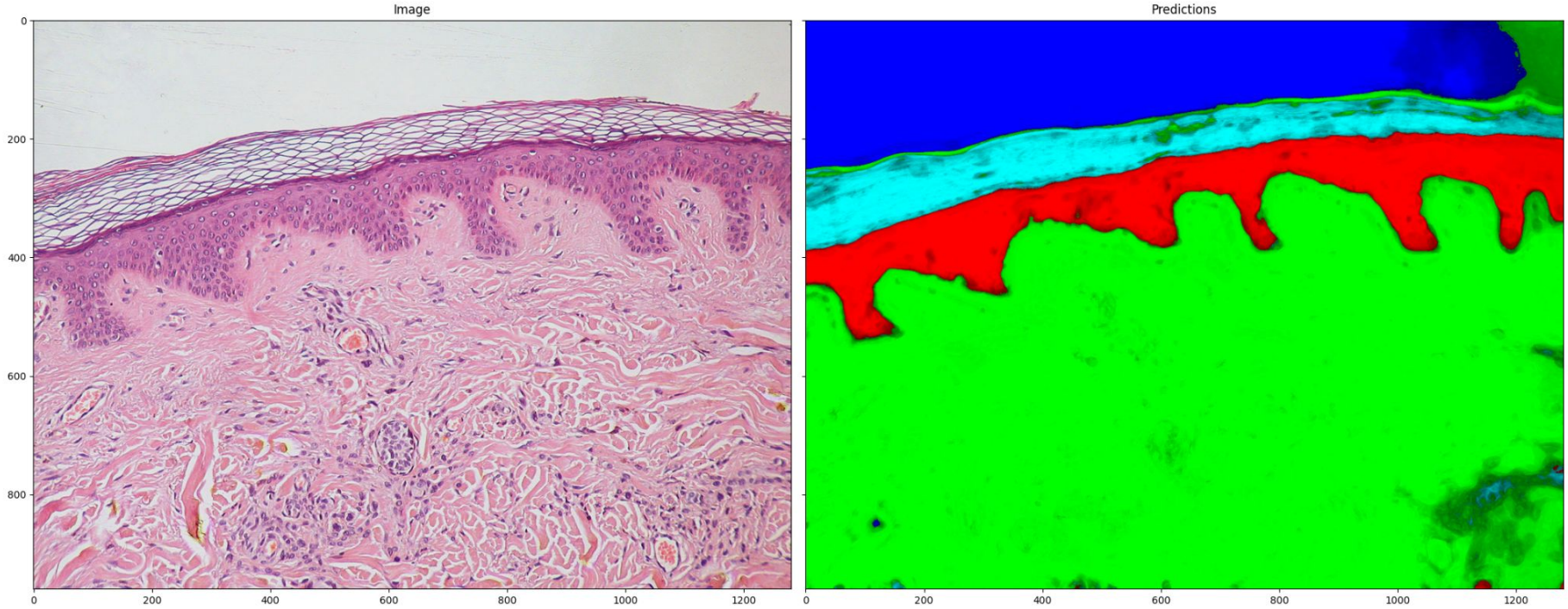
Don’t scale up/out (yet)

One Core Is All You Need™

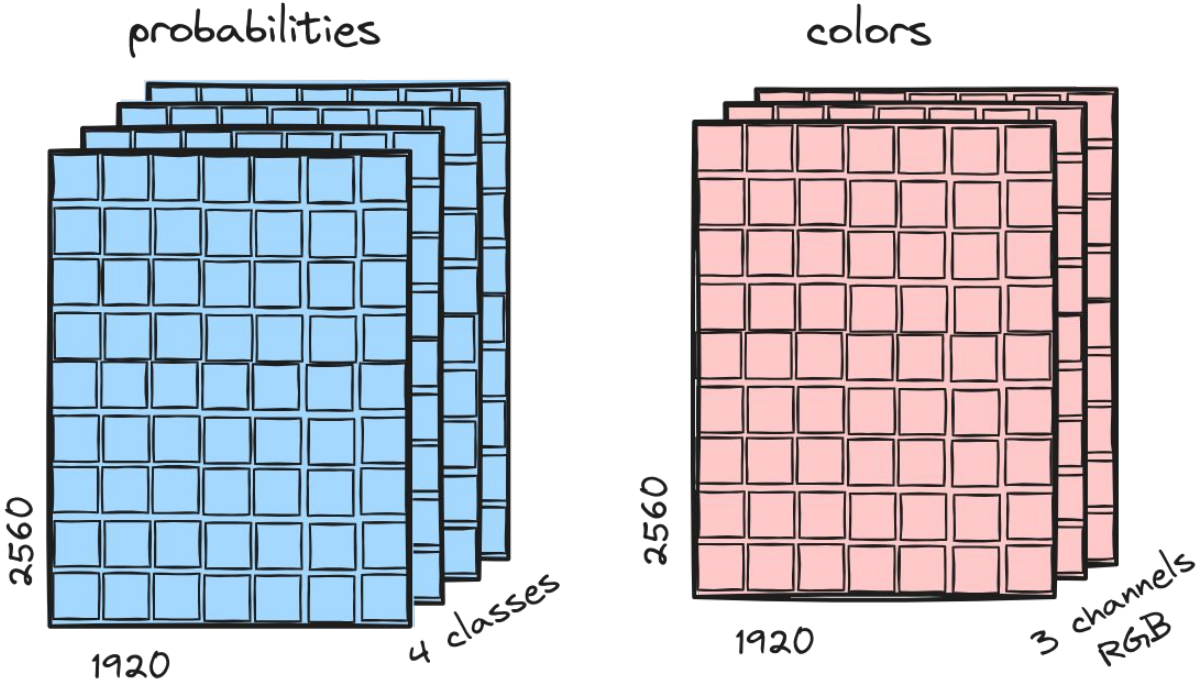
Example: Colorcode Predictions

Based on scikit-image tutorial

"Trainable segmentation using local features and random forests"



Example: Colorcode Predictions



Example: Colorcode Predictions

```
def load_probabilities(folder):
    arrays = [np.loadtxt(folder / f"{c}.csv") for c in range(4)]
    return np.stack(arrays, axis=-1)
```

Example: Colorcode Predictions

```python
def colorcode_probabilities(probabilities):
    class_colors = [[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]]
    colored_probabilities = []
    for probabilities_row in probabilities:
        colored_probabilities_row = []
        for class_probabilities in probabilities_row:
            class_index = None
            max_prob = 0
            for i, prob in enumerate(class_probabilities):
                if prob > max_prob:
                    max_prob = prob
                    class_index = i
            colored_probabilities_row.append([c * max_prob for c in class_colors[class_index]])
        colored_probabilities.append(colored_probabilities_row)
    return colored_probabilities
```

Example: Colorcode Predictions

Profile, Optimize, Repeat

15s → 150ms

800MB → 80MB

Profile

- Measure
- Speed & Memory Bottlenecks

What?

- to Identify & Analyse them
- for Mitigation

Why?

Time/CPU Profiling

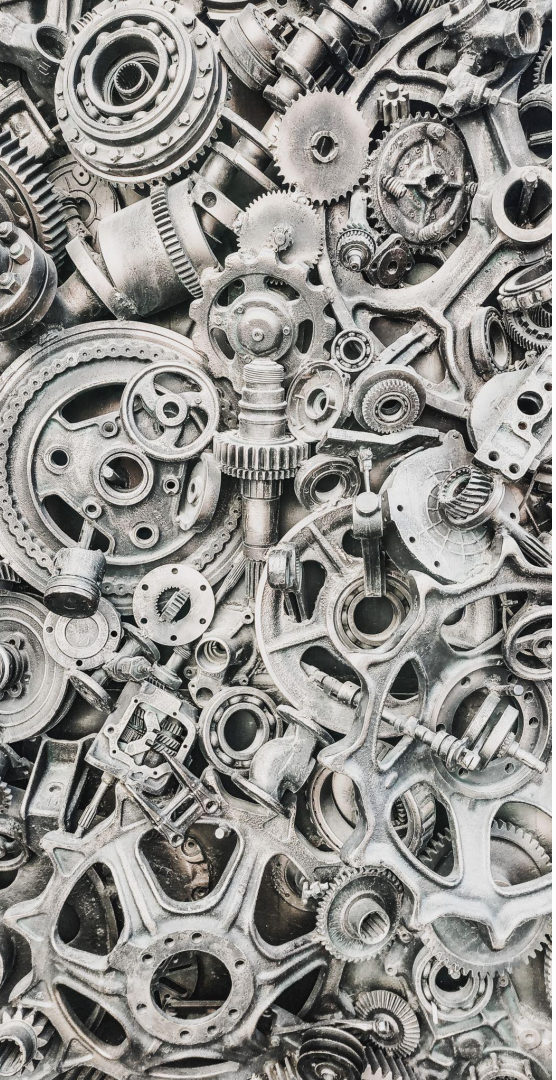
Slow Code

- Where?
- How slow?
- Why?

Memory Profiling

High Memory Usage

- Where?
- How much?
- Why?



Instrumenting vs Sampling

Deterministic Instrumenting

- Measure **each function or code block** of a program
- Potential overhead
- Inaccuracies for calls with lots of instrumentation inside

Statistical Sampling

- **Periodically sample** the program's state
- Can miss brief invocations
  (usually fine for timing, problematic for short memory spikes)

Scalene

CPU, Memory, GPU profiling

sampling-based

nice visualization

More

Profiler	Link	Time	Mem	Viz	Type	Other
cProfile			✗	✗	instrumenting	Viz via SnakeViz or Tuna
py-spy			✗		sampling	
memray		✗			instrumenting	
Scalene					sampling	also GPU time
Austin					sampling	can also be used with pprof
pyinstrument			✗		sampling	
line_profiler			✗	✗	instrumenting	
Fil		✗			instrumenting	
Guppy3		✗		✗	instrumenting	
Yappi			✗	✗	instrumenting	
Palanteer					instrumenting	also supports C++
vprof					sampling	
VizTracer			✗		instrumenting	
sciagraph					sampling	proprietary profiler

codical.org

Run scalene

main.py

```
from scalene import scalene_profiler
scalene_profiler.start()
CODE
scalene_profiler.stop()
```

```
scalene --off --cli --cpu main.py --- v0 --data data/m
```

15s runtime

Line	Time Python	------ native	------ system	
1				`import numpy as np`
2				
3				
4				`def load_probabilities(folder):`
5	24%			`    arrays = [np.loadtxt(folder / f"{c}.csv") for c in range(4)]`
6				`    return np.stack(arrays, axis=-1)`
7				
8				
9				`def colorcode_probabilities(probabilities):`
10				`    class_colors = [[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]]`
11				`    colored_probabilities = []`
12				`    for probabilities_row in probabilities:`
13				`        colored_probabilities_row = []`
14				`        for class_probabilities in probabilities_row:`
15				`            class_index = None`
16				`            max_prob = 0`
17	6%			`            for i, prob in enumerate(class_probabilities):`
18				`                if prob > max_prob:`
19				`                    max_prob = prob`
20				`                    class_index = i`
21	**41%**	**9%**	1%	`            colored_probability = [c * max_prob for c in class_colors[class_index]]`
22	3%			`            colored_probabilities_row.append(colored_probability)`
23				`        colored_probabilities.append(colored_probabilities_row)`
24				`    return colored_probabilities`

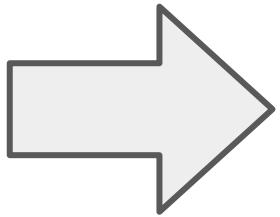
Efficient IO

Text-based

csv

json

yaml



Binary format

hdf5

npy

parquet

pickle

sqlite

zarr

...

see also

github.com/mverleg/array_storage_benchmark

Line	Time Python	------ native	------ system	10.6s runtime
1				`import h5py`
2				
3				`def load_probabilities(folder):`
4				`    with h5py.File(folder / "data.hdf5", "r") as f:`
5				`        return f["probabilities"][:]`
6				
7				`def colorcode_probabilities(probabilities):`
8				`    class_colors = [[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]]`
9				`    colored_probabilities = []`
10				`    for probabilities_row in probabilities:`
11				`        colored_probabilities_row = []`
12				`        for class_probabilities in probabilities_row:`
13				`            class_index = None`
14				`            max_prob = 0`
15	9%			`            for i, prob in enumerate(class_probabilities):`
16				`                if prob > max_prob:`
17				`                    max_prob = prob`
18				`                    class_index = i`
19	**52%**	**13%**	3%	`            colored_probability = [c * max_prob for c in class_colors[class_index]]`
20	5%			`            colored_probabilities_row.append(colored_probability)`
21				`        colored_probabilities.append(colored_probabilities_row)`
22				`    return colored_probabilities`

Vectorization

NumPy

- compact in-memory representation for homogeneous data

  ⟶less powerful than Python types, but less overhead
- optimized primitives for
  - add, multiply, ...
  - argmax, unique, ...
  - fancy indexing

  ⟶NumPy core is written in C

Vectorization

```
def colorcode_probabilities(probabilities):
    predicted_classes = np.argmax(probabilities, axis=2)
    max_prob = np.max(probabilities, axis=2)
    class_colors = np.array( object: [[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]], dtype=np.uint8)
    colored_probabilities = class_colors[predicted_classes]
    colored_probabilities = colored_probabilities * max_prob[..., np.newaxis]
    return colored_probabilities.astype(np.uint8)
```

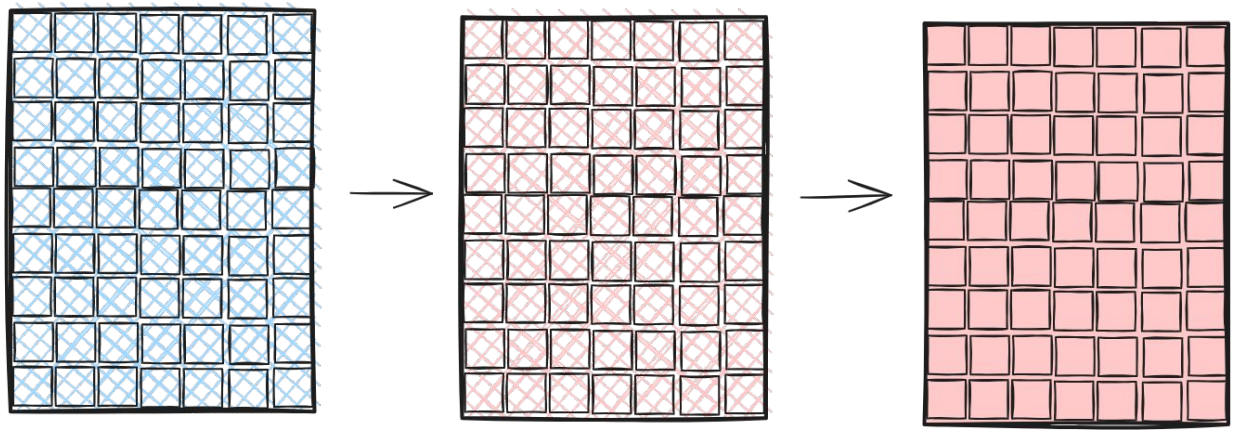
830ms runtime

Line	Time *Python*	------ *native*	------ *system*	
1				`import numpy as np`
2				`import h5py`
3				
4				`def load_probabilities(folder):`
5				`    with h5py.File(folder / "data.hdf5", "r") as f:`
6				`        return f["probabilities"][:]`
7				
8				`def colorcode_probabilities(probabilities):`
9	1%	4%	1%	`    predicted_classes = np.argmax(probabilities, axis=2)`
10		26%	16%	`    max_prob = np.max(probabilities, axis=2)`
11				`    class_colors = np.array([[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]], dtype=np.uint8)`
12				`    colored_probabilities = class_colors[predicted_classes]`
13				`    colored_probabilities = colored_probabilities * max_prob[..., np.newaxis]`
14		17%	32%	`    return colored_probabilities.astype(np.uint8)`

Vectorization

```python
def colorcode_probabilities(probabilities):
    predicted_classes = np.argmax(probabilities, axis=2)
    max_prob = np.max(probabilities, axis=2)
    class_colors = np.array( object: [[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]], dtype=np.uint8)
    colored_probabilities = class_colors[predicted_classes]
    colored_probabilities = colored_probabilities * max_prob[..., np.newaxis]
    return colored_probabilities.astype(np.uint8)
```

157MB



15MB

352MB peak mem

Memory Python	------ peak	---------- timeline/%	Copy (MB/s)	
				`import numpy as np`
				`import h5py`
				`# Change: all numpy`
				`def load_probabilities(folder):`
				`    with h5py.File(folder / "data.hdf5", "r") as f:`
	151M	▂ 41%		`        return f["probabilities"][:]`
				`def colorcode_probabilities(probabilities):`
68%	37M	▂ 10%		`    predicted_classes = np.argmax(probabilities, axis=2)`
36%	37M	▂ 10%		`    max_prob = np.max(probabilities, axis=2)`
				`    class_colors = np.array([[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]], dtype=np.uint8)`
7%	14M	▂ 4%	80	`    colored_probabilities = class_colors[predicted_classes]`
	113M	▃ 31%		`    colored_probabilities = colored_probabilities * max_prob[..., np.newaxis]`
	14M	▂ 4%		`    return colored_probabilities.astype(np.uint8)`

Reduce Memory Usage

- Optimize data types

uint8	0 to 255	1 byte
int32	–2147483648 to 2147483647	4 bytes
float32	-3.4e+38 to 3.4e+38	4 bytes
float64	-1.79e+308 to 1.79e+308	8 bytes

- Manual GC
- Manual loop-unrolling (avoid broadcasting)

Optimized Numpy

```
def colorcode_probabilities(probabilities):
    # use uint8 to reference the classes (we only have 4)
    predicted_classes = np.argmax(probabilities, axis=2).astype(np.uint8)
    # Do no recompute max, use argmax resul to index
    max_prob = np.take_along_axis(probabilities, predicted_classes[..., None], axis=2).squeeze()
    # manual GC, we don't need the probabilities anymore
    del probabilities
    # use float32 to save memory
    max_prob = max_prob.astype(np.float32)
    class_colors = np.array( object: [[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]], dtype=np.uint8)
    colored_probabilities = class_colors[predicted_classes]
    # avoid broadcasting by unrolling the RGB loop
    colored_probabilities[:, :, 0] = colored_probabilities[:, :, 0] * max_prob
    colored_probabilities[:, :, 1] = colored_probabilities[:, :, 1] * max_prob
    colored_probabilities[:, :, 2] = colored_probabilities[:, :, 2] * max_prob
    return colored_probabilities
```

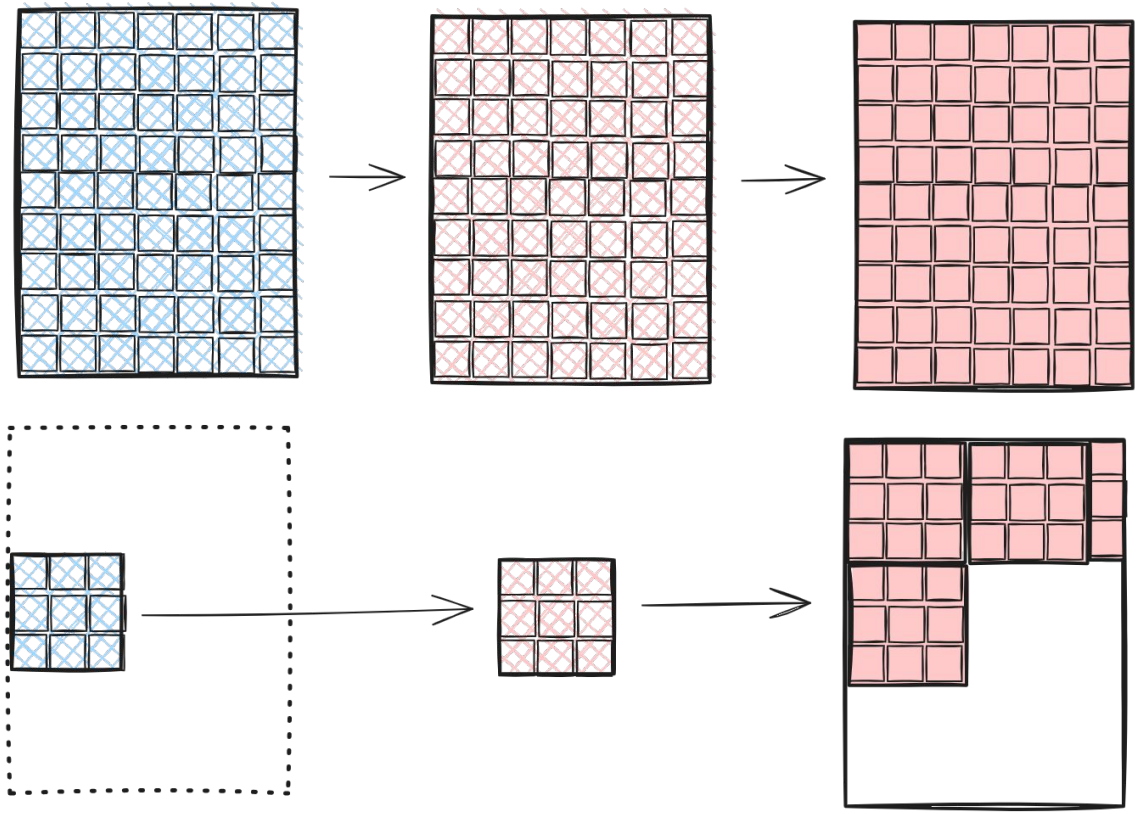
211MB peak mem

Memory Python	peak	timeline/%	Copy (MB/s)	
				`import numpy as np`
				`import h5py`
				`def load_probabilities(folder):`
				`    with h5py.File(folder / "data.hdf5", "r") as f:`
	151M	▄ 48%		`        return f["probabilities"][:]`
				`def colorcode_probabilities(probabilities):`
				`    """Use uint8 to reference classes (we only have 4):"""`
53%	37M	▂ 12%		`    predicted_classes = np.argmax(probabilities, axis=2).astype(np.uint8)`
				`    """Do no re-compute max, use argmax result to index:"""`
33%	38M	▂ 12%	133	`    max_prob = np.take_along_axis(probabilities, predicted_classes[..., None], axis=2).squeeze()`
				`    """Manual GC, no longer needed:"""`
				`    del probabilities`
				`    """Use float32 to save memory:"""`
	19M	▂ 6%		`    max_prob = max_prob.astype(np.float32)`
				`    class_colors = np.array([[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]], dtype=np.uint8)`
	14M	▁ 4%		`    colored_probabilities = class_colors[predicted_classes]`
				`    """unroll RGB loop:"""`
	19M	▂ 6%		`    colored_probabilities[:, :, 0] = colored_probabilities[:, :, 0] * max_prob`
	19M	▂ 6%		`    colored_probabilities[:, :, 1] = colored_probabilities[:, :, 1] * max_prob`
	19M	▂ 6%		`    colored_probabilities[:, :, 2] = colored_probabilities[:, :, 2] * max_prob`
				`    return colored_probabilities`

300ms runtime

Line	Time Python	------ native	------ system	
1				`import numpy as np`
2				`import h5py`
3				
4				`def load_probabilities(folder):`
5				`    with h5py.File(folder / "data.hdf5", "r") as f:`
6				`        return f["probabilities"][:]`
7				
8				`def colorcode_probabilities(probabilities):`
9				`    """Use uint8 to reference classes (we only have 4):"""`
10		12%	29%	`    predicted_classes = np.argmax(probabilities, axis=2).astype(np.uint8)`
11				`    """Do no re-compute max, use argmax result to index:"""`
12	2%	9%	4%	`    max_prob = np.take_along_axis(probabilities, predicted_classes[..., None], axis=2).squeeze()`
13				`    """Manual GC, no longer needed:"""`
14				`    del probabilities`
15				`    """Use float32 to save memory:"""`
16		1%	16%	`    max_prob = max_prob.astype(np.float32)`
17				`    class_colors = np.array([[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]], dtype=np.uint8)`
18				`    colored_probabilities = class_colors[predicted_classes]`
19				`    """unroll RGB loop:"""`
20				`    colored_probabilities[:, :, 0] = colored_probabilities[:, :, 0] * max_prob`
21				`    colored_probabilities[:, :, 1] = colored_probabilities[:, :, 1] * max_prob`
22				`    colored_probabilities[:, :, 2] = colored_probabilities[:, :, 2] * max_prob`
23				`    return colored_probabilities`

Chunking



Chunking

```python
def get_chunk_slices(length, chunk_size):
    for start in range(0, length, chunk_size):
        end = min(start + chunk_size, length)
        yield slice(start, end)



def load_probabilities(folder):
    with h5py.File(folder / "data.hdf5", mode: "r") as f:
        x_len, y_len = f["probabilities"].shape[:2]
        for x_slice in get_chunk_slices(x_len, CHUNK_SIZE):
            for y_slice in get_chunk_slices(y_len, CHUNK_SIZE):
                yield (x_slice, y_slice), f["probabilities"][x_slice, y_slice]
```

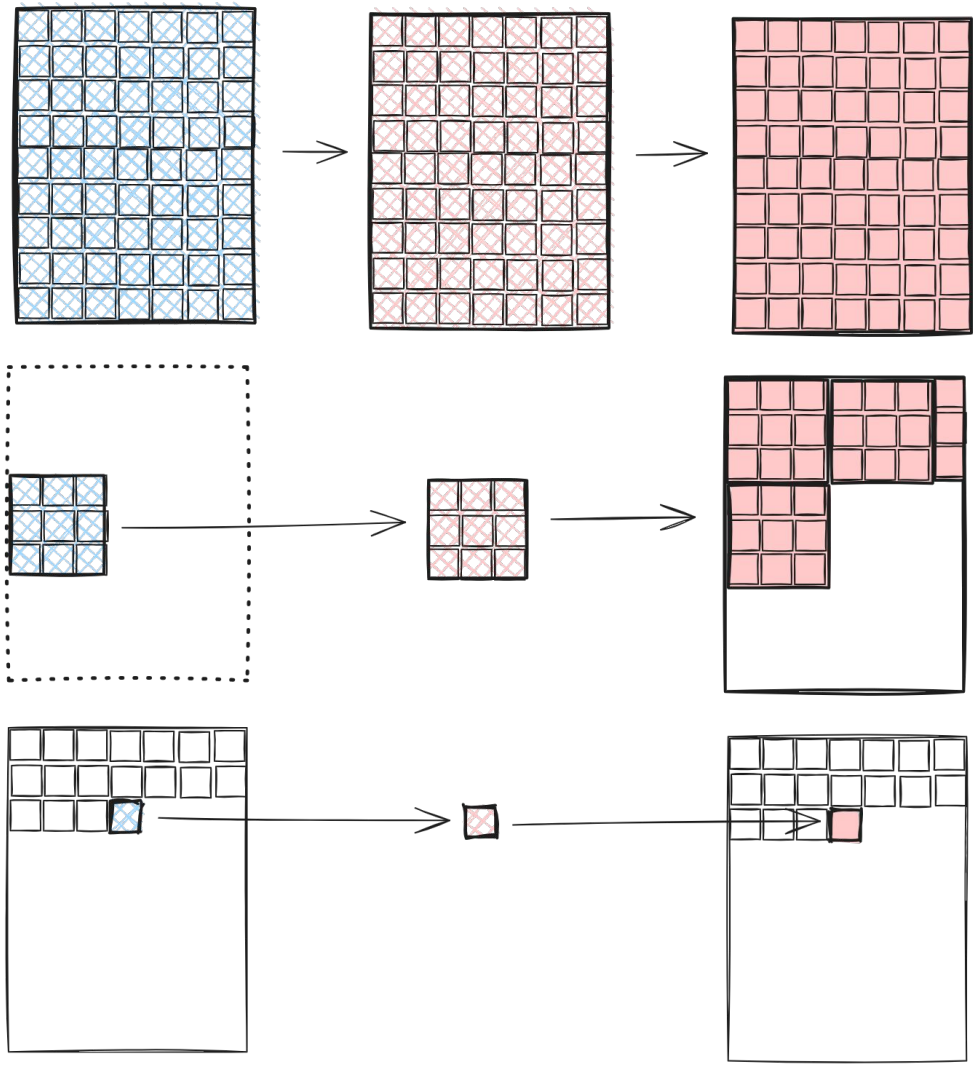
Chunking

```python
def load_and_colorcode_probabilities(folder):
    with h5py.File(folder / "data.hdf5", mode: "r") as f:
        x_len, y_len = f["probabilities"].shape[:2]
    colored_probabilities = np.zeros( shape: (x_len, y_len, 3), dtype=np.uint8)
    for chunk_slice, chunk in load_probabilities(folder):
        colored_probabilities[chunk_slice] = colorcode_probabilities(chunk)
    return colored_probabilities
```

Chunking

	Whole-Array (1920x2560)	Chunked (1024^2)
Runtime	300ms	335ms
Memory	211MB	79MB

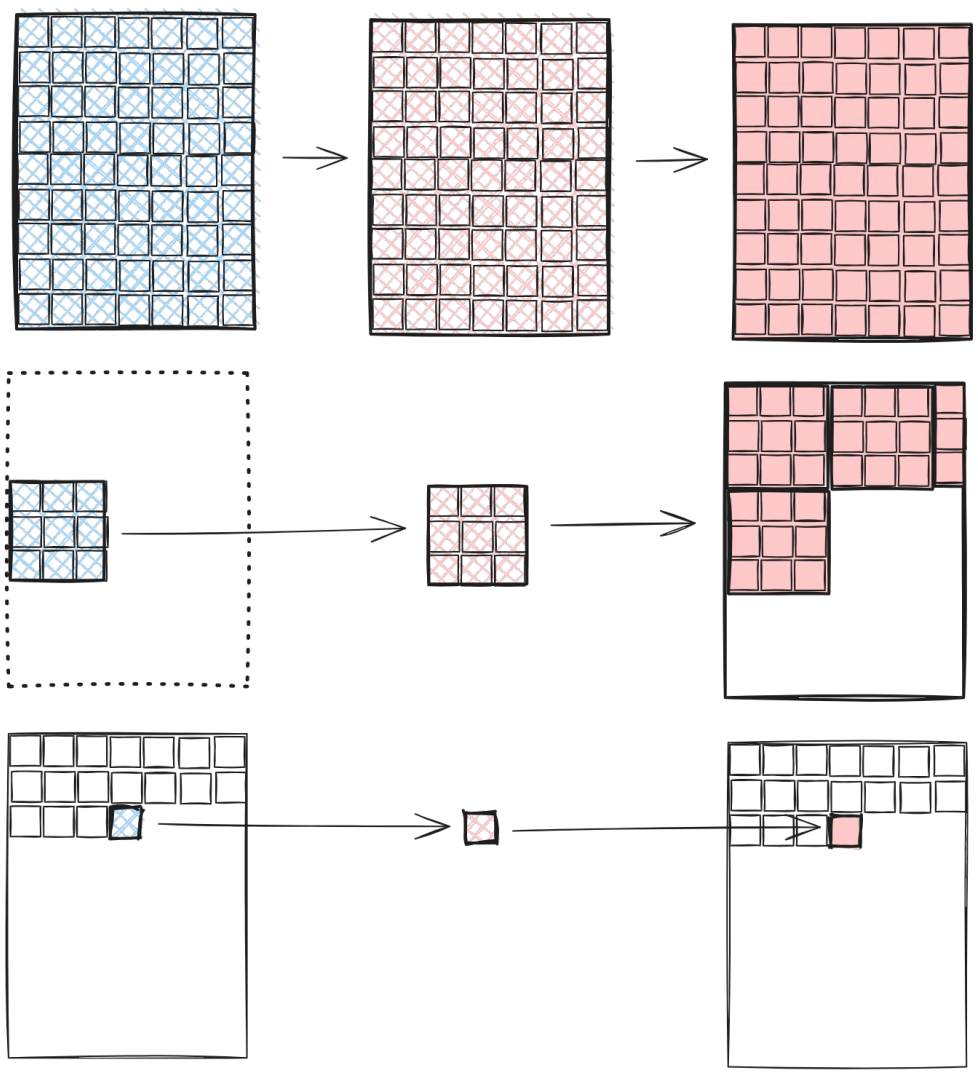
Element-wise Map with Numba



Element-wise Map with Numba

Numba Alternatives

- Pythran
- mypyc
- cython
- ...



Element-wise Map with Numba

```
@numba.njit([numba.uint8[:, :, :](numba.float64[:, :, :])])
def colorcode_probabilities(probabilities):
    class_colors = np.array( object: [[0, 0, 255], [0, 255, 0], [255, 0, 0], [0, 255, 255]], dtype=np.uint8)
    x_len, y_len = probabilities.shape[:2]
    colored_probabilities = np.zeros( shape: (x_len, y_len, 3), dtype=np.uint8)
    for x in range(x_len):
        for y in range(y_len):
            class_index = 0
            max_prob = 0
            for i, prob in enumerate(probabilities[x, y]):
                if prob > max_prob:
                    max_prob = prob
                    class_index = i
            colored_probabilities[x, y, 0] = np.uint8(class_colors[class_index, 0] * max_prob)
            colored_probabilities[x, y, 1] = np.uint8(class_colors[class_index, 1] * max_prob)
            colored_probabilities[x, y, 2] = np.uint8(class_colors[class_index, 2] * max_prob)
    return colored_probabilities
```

Element-wise Map with Numba

	numpy	**numba**
Runtime	300ms	122ms
Memory	211MB	164MB

(non-chunked)

Go More Low-Level

- Custom C++ Extension with pybind11
  - Integration of existing c/c++ high-performance code
  - Fine grained control over memory access
  - SIMD
  - Inline Assembly
- Alternatives: PyO3 (rust)

Pybind11 extension

```cpp
py::array_t<uint8_t> colorcode_probabilities(py::array_t<double> probabilities, py::array_t<uint8_t> colors) {
    auto shape = probabilities.shape();
    auto probabilities_ptr = static_cast<double*>(probabilities.request().ptr);
    auto colors_ptr = static_cast<uint8_t*>(colors.request().ptr);
    auto colored_image = py::array_t<uint8_t>({int(shape[0]), int(shape[1]), 3});
    auto colored_image_ptr = static_cast<uint8_t*>(colored_image.request().ptr);
    for (int i = 0; i < shape[0]; i++) {
        for (int j = 0; j < shape[1]; j++) {
            double max_probability = 0.0;
            size_t max_index = 0;
            for (int k = 0; k < shape[2]; k++) {
                auto probability = *probabilities_ptr++;
                if (probability > max_probability) {
                    max_probability = probability;
                    max_index = k;
                }
            }
            auto class_color_ptr = colors_ptr + (max_index * 3);
            *colored_image_ptr++ = static_cast<uint8_t>(*class_color_ptr++ * max_probability);
            *colored_image_ptr++ = static_cast<uint8_t>(*class_color_ptr++ * max_probability);
            *colored_image_ptr++ = static_cast<uint8_t>(*class_color_ptr++ * max_probability);
        }
    }
    return colored_image;
}
```

Pybind11 extension

	numpy	numba	pybind11
Runtime	300ms	122ms	119ms
Memory	211MB	164MB	164MB

(non-chunked)

- Profile
- Efficient IO
- Vectorization
- Reduce Memory Usage
- Go More Low-Level

Optimize

Repeat

15s – 800MB

150ms – 80MB

Now we can scale, e.g. with

- threading
- multiprocessing
- joblib, Spark, Dask, Ray, ...

Further reading:

- codical.org
- pythonspeed.com/datascience
- calmcode.io

Thanks!

code & slides

github.com/jstriebel/profile-optimize-repeat

Valentin Nieper | valentin.nieper@scalableminds.com

Jonathan Striebel | jonathan@aignostics.com @jostriebel