Scientific Software Center

IWR Interdisciplinary Center for Scientific Computing

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Python Performance Profiling

Liam Keegan, SSC

ssciwr.github.io/python-performance-profiling

2026.01.28

Course Outline

- Python Profiling
  - Introduction
  - Profiling Python code
  - Profiling multiprocessing Python code
  - Profiling compiled extensions
  - Profiling GPU code
  - Memory profiling
- Hands on
  - Python code
  - Python code with multiprocessing
  - Pytorch code

Profiling

Profiling

Profiling means finding out where your code spends time when it is running.

This is the key first step to making your code run faster - identifying “bottlenecks” or “hotspots” that account for a large fraction of the runtime, which you can then optimise.

You can also profile memory usage, I/O usage, network use, or any other resource that you care about.

Why profile?

Trying to make your code faster (or use less memory), without first understanding which parts are slow (or use too much memory), is a guessing game.

The end result is often wasted effort optimising code that doesn't actually result in a significant change in the overall performance.

For example, if one function is responsible for 90% of the execution time, then it's probably a waste of time optimising anything else in the codebase.

Without profiling, attempting to improve performance can be very inefficient.

Profiler output

To profile our code we will use a tool called a profiler.

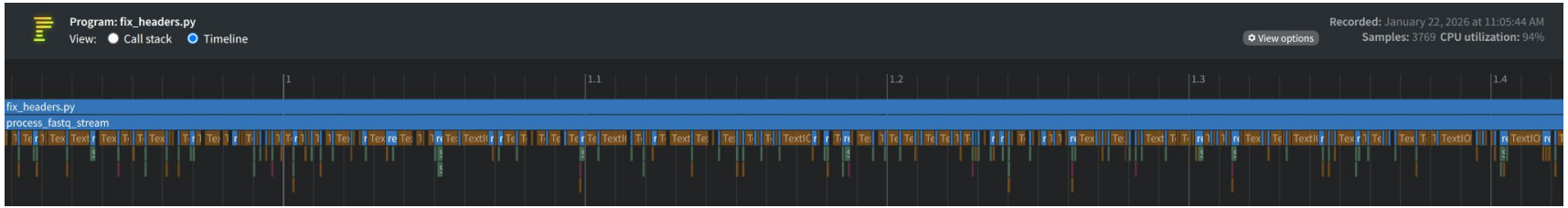
There are a variety of profilers which can produce all kinds of different outputs.

But there are two key types of output:

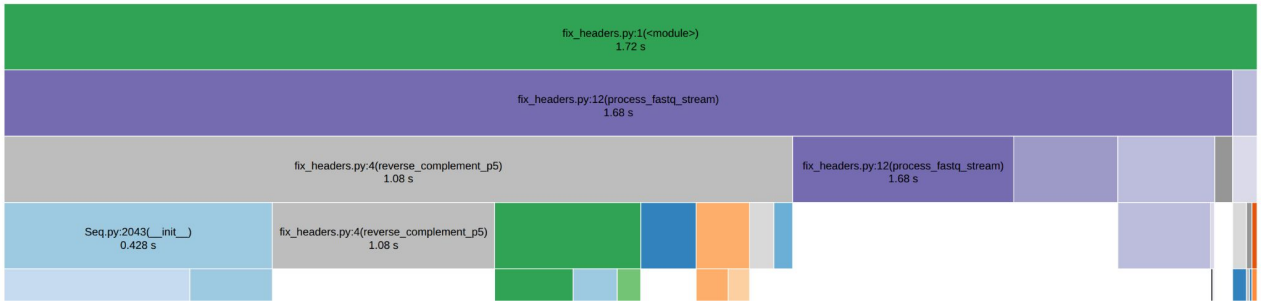
- Trace (time series data, e.g. a timeline)
  - What is happening as a function of time, i.e. WHEN things happen
- Profile (time-averaged data, e.g. a flamegraph)
  - How much each function contributes in total, i.e. WHAT happens

Profiler output

Trace output: x-axis is timestamps



Profile output: x-axis is total amount of time



Types of profilers

Profilers come in two flavours:

- Tracing (deterministic) profilers
- Sampling (statistical) profilers

They both record information about what your code is doing as it runs, in particular

- Which functions get called
- Which functions do these functions call
- How long do the functions run for

Tracing profilers

Instrument and record every single function call in your program.

- Advantages
  - Accurate (in the sense that every single function call is recorded)
  - Deterministic (no statistical sampling of function calls)
  - Can be very granular (down to individual lines within a function)
- Disadvantages
  - Large overhead (which can reduce accuracy by affecting the performance)
  - Generates a lot of data (can be too much to easily deal with)
- Examples
  - cProfile
  - line_profiler

Sampling profilers

Periodically interrupt execution and record what is running.

- Advantages
  - Significantly less overhead than tracing profilers
  - Typically less intrusive (e.g. don’t require changes in the code)
- Disadvantages
  - Missing data (statistically not an issue but may be if you zoom in on a trace/timeline)
  - Biased against rare events (may be a problem if you’re worried about worst case / latency)
  - Not deterministic (if you run it again you will sample different function calls)
- Examples
  - py-spy
  - perf

A note on benchmarking

Benchmarking and profiling are both concerned with performance, but they are answering slightly different questions:

- Profiling is about understanding **where** and **why** your code is slow
  - Typically used to find out where and how you can improve performance
- Benchmarking is about understanding **how fast** your code is
  - Typically used to compare performance between alternative implementations

A benchmark typically runs the same piece of code many times, and records the average (or best) runtime for that piece of code.

Profiling Python code: cProfile

cProfile

- Built-in tracing (deterministic) profiling tool in Python
- Records count and execution time of every function call
- Profiles only Python-level code

Easy to use, nothing to install, can just prepend cProfile to your script:

```
python -m cProfile pipeline.py
```

This command runs the pipeline.py script and prints a summary of the profiling data to the console

cProfile output for each function

- ncalls
  - Number of times the function was called
  - For recursive functions shows two values: total_calls (including recursive) / primitive_calls (excluding recursive)
- tottime
  - Time spent inside the function body itself
  - Excludes time spent in functions it calls
- percall (tottime)
  - tottime / ncalls
  - Average self-time per call
- cumtime
  - Cumulative time spent in the function including all subcalls
  - This is usually the most useful column for finding bottlenecks
- percall (cumtime)
  - cumtime / primitive_calls
  - Average cumulative time per (non-recursive) call
- filename:lineno(function)
  - Source file, line number, and function name
  - Built-ins appear in braces, e.g. {built-in method builtins.exec}

cProfile output example

```
         58372623 function calls (58372593 primitive calls) in 10.982 seconds

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    7.054    7.054   10.504   10.504 pipeline.py:93(find_similar_records)
 49995000    2.895    0.000    2.895    0.000 {built-in method builtins.abs}
  7181275    0.581    0.000    0.581    0.000 {method 'append' of 'list' objects}
        1    0.008    0.008    0.208    0.208 pipeline.py:83(compute_statistics)
    10000    0.085    0.000    0.156    0.000 pipeline.py:30(normalize_name)
        1    0.007    0.007    0.143    0.143 pipeline.py:10(generate_records)
    10000    0.053    0.000    0.093    0.000 pipeline.py:26(generate_values)
```

cProfile visualization

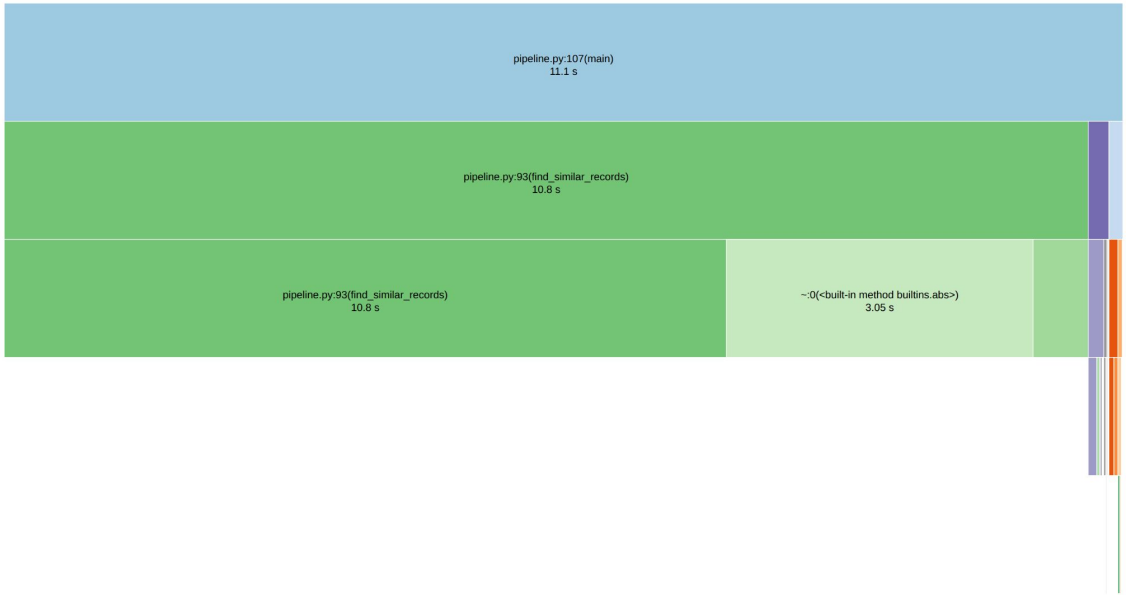
- This text output is useful, but not very easy to read!
- Typically we write the profiling data to a file then visualise with another tool
- Snakeviz provides an interactive visual version of the profiling data
  - `pip install snakeviz`
- gprof2dot + Graphviz provides a call graph
  - `pip install gprof2dot`
  - Graphviz is usually pre-installed on linux, on mac: `brew install graphviz`

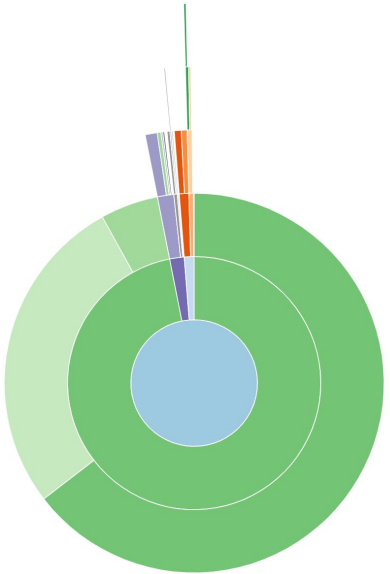
To output the profile data to a file we use the -o option:

```
python -m cProfile -o out.prof pipeline.py
```

cProfile + snakeviz

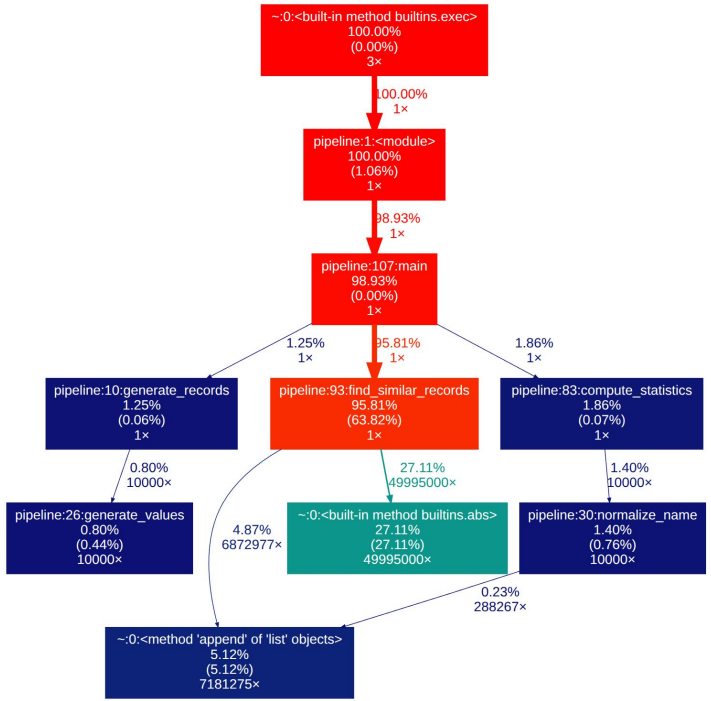
```
snakeviz out.prof
```





cProfile + gprof2dot

```
gprof2dot -f pstats out.prof | dot -Tsvg -o out.svg
```



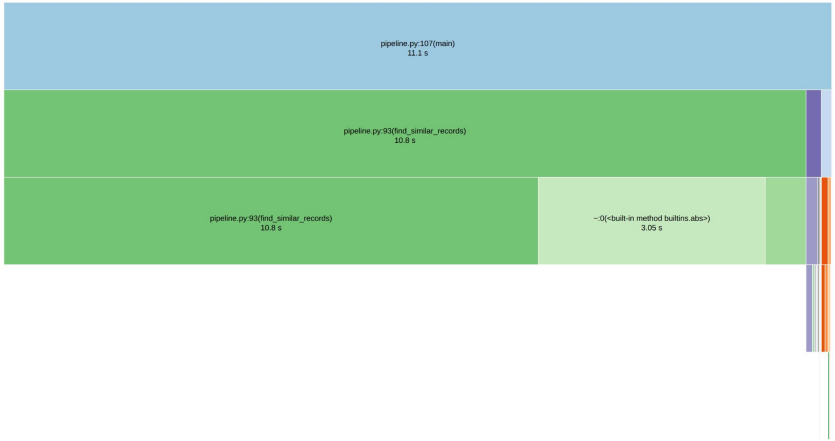
cProfile as a library

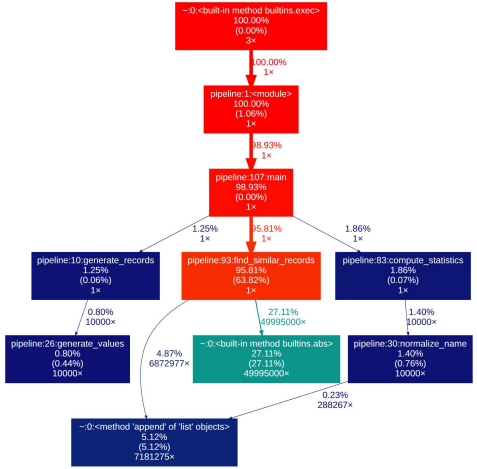
- For more control over what gets profiled you can use cProfile as a library in your code
- One way to do this is using it as a context manager:

```
import cProfile


with cProfile.Profile() as pr:

    # ... do the work that you want to profile here ...

    pr.dump_stats(“out.prof”)
```

cProfile summary

- Built-in profiling tool in Python
- Use snakeviz and gprof2dot to visualise the output
- Provides a nice overview
- Good way to identify hotspots, understand where time is being spent





Profiling Python code: line_profiler

line_profiler

- Line-by-line tracing profiling tool in Python
- Records count and execution time of every line of code within a function
- Profiles only Python-level code
- Not suited for multi-threaded, multi-processing, or asynchronous code

Install with pip

```
pip install line_profiler
```

line_profiler use

In your code import the profile decorator:

```
from line_profiler import profile
```

And decorate any functions you want to profile with

```
@profile
```

By default this does nothing unless the `LINE_PROFILE` env var is set.

To see the profiling output, run your script normally (with this env var set):

```
LINE_PROFILE=1 python script.py
```

line_profiler output

```
Function: normalize_name at line 31

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    31                                           def normalize_name(name):
    32                                               """
    33                                               Normalize a person's name for matching.
    34
    35                                               Steps:
    36                                               - Unicode normalization
    37                                               - Case folding
    38                                               - Remove accents
    39                                               - Remove punctuation
    40                                               - Normalize whitespace
    41                                               """
    42                                               # Unicode normalization
    43     10000       6336.6      0.6      1.8      name = unicodedata.normalize("NFKD", name)
    44
    45                                               # Remove accents
    46     10000       3408.6      0.3      1.0      chars = []
    47    168025      56613.2      0.3     16.0      for c in name:
    48    158025      59946.0      0.4     16.9          if not unicodedata.combining(c):
    49    145153      47437.5      0.3     13.4              chars.append(c)
    50     10000       4805.4      0.5      1.4      name = "".join(chars)
    51
    52                                               # Case folding
    53     10000       4000.0      0.4      1.1      name = name.casefold()
    54
    55                                               # Remove punctuation
    56     10000       3392.2      0.3      1.0      cleaned = []
    57    155153      51032.0      0.3     14.4      for c in name:
    58    145153      49839.1      0.3     14.1          if c.isalnum() or c.isspace():
    59    143114      48042.6      0.3     13.6              cleaned.append(c)
    60     10000       4601.2      0.5      1.3      name = "".join(cleaned)
    61
    62                                               # Normalize whitespace
    63     10000      10352.2      1.0      2.9      name = WHITESPACE_RE.sub(" ", name).strip()
    64
    65     10000       3907.4      0.4      1.1      return name

Total time: 0.661204 s
```

line_profiler alternative use

If you just want to profile every line of script.py you can instead do:

```
python -m kernprof -lv -p script.py script.py
```

This doesn’t require any modifications to the source code.

But it may result in a huge amount of output depending on how large your code is.

line_profiler summary

- Provides line-level profiling data
- Useful to investigate which parts of a function is expensive
- For example a function with a bunch of numpy/pytorch calls
- Good way to identify hotspots within a function

```
Function: normalize_name at line 31

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    31                                           def normalize_name(name):
    32                                               """
    33                                               Normalize a person's name for matching.
    34
    35                                               Steps:
    36                                               - Unicode normalization
    37                                               - Case folding
    38                                               - Remove accents
    39                                               - Remove punctuation
    40                                               - Normalize whitespace
    41                                               """
    42                                               # Unicode normalization
    43     10000       6336.6      0.6      1.8      name = unicodedata.normalize("NFKD", name)
    44
    45                                               # Remove accents
    46     10000       3408.6      0.3      1.0      chars = []
    47    168025      56613.2      0.3     16.0      for c in name:
    48    158025      59946.0      0.4     16.9          if not unicodedata.combining(c):
    49    145153      47437.5      0.3     13.4              chars.append(c)
    50     10000       4805.4      0.5      1.4      name = "".join(chars)
```

Profiling Python code: pyinstrument

pyinstrument

- Sampling (statistical) profiling tool in Python
- Displays call stack and timeline in a web browser

Install with pip

```
pip install pyinstrument
```

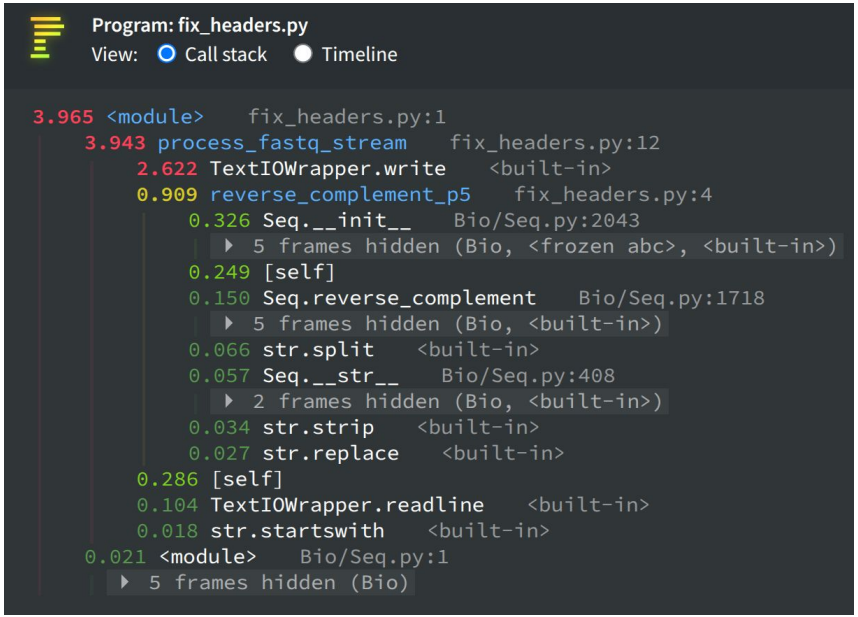
Replace python with pyinstrument when running your code.

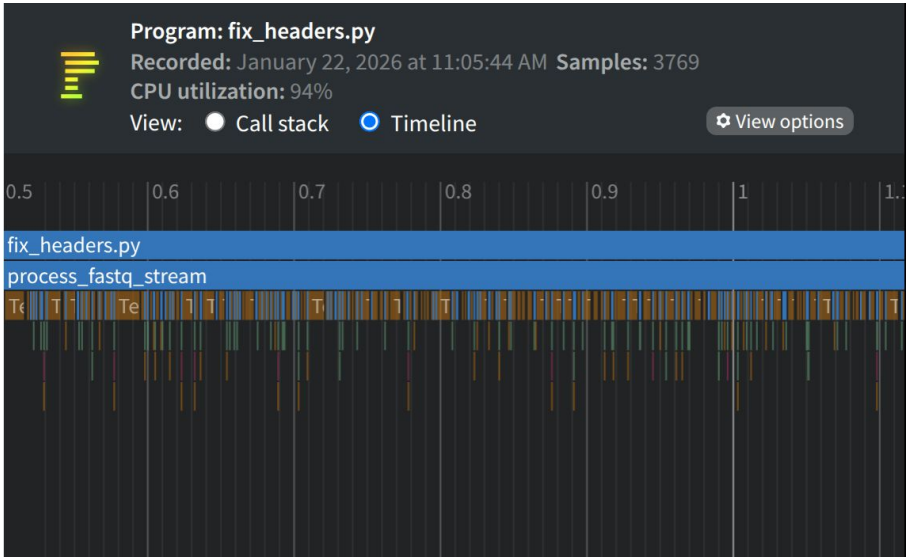
Multiple different renders are available, html is a good option:

```
pyinstrument -r html script.py
```

Pyinstrument output

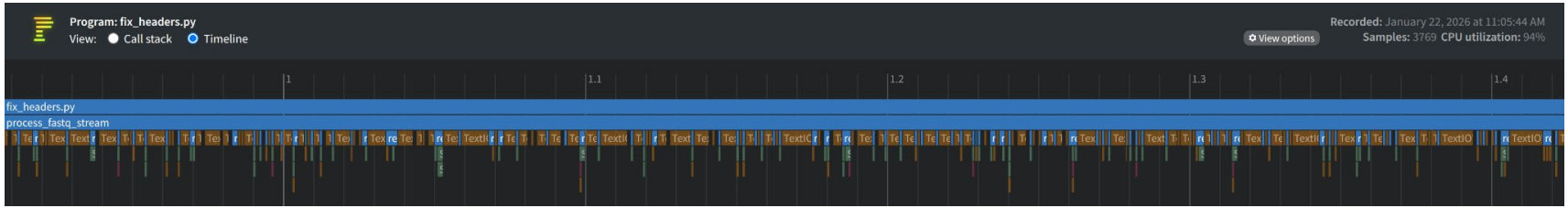
You can choose between a call stack view and a timeline view:





Pyinstrument summary

- Sampling (statistical) profiling tool in Python
- Simple to use, no need for additional tools to visualise output
- Offers a timeline view which is not available with cProfile



Profiling Python code: memray

Scientific Software Center

memray

- Memory profiling tool in Python
- Displays call stack and timeline in a web browser

Install with pip

```
pip install memray
```

Run your script with memray

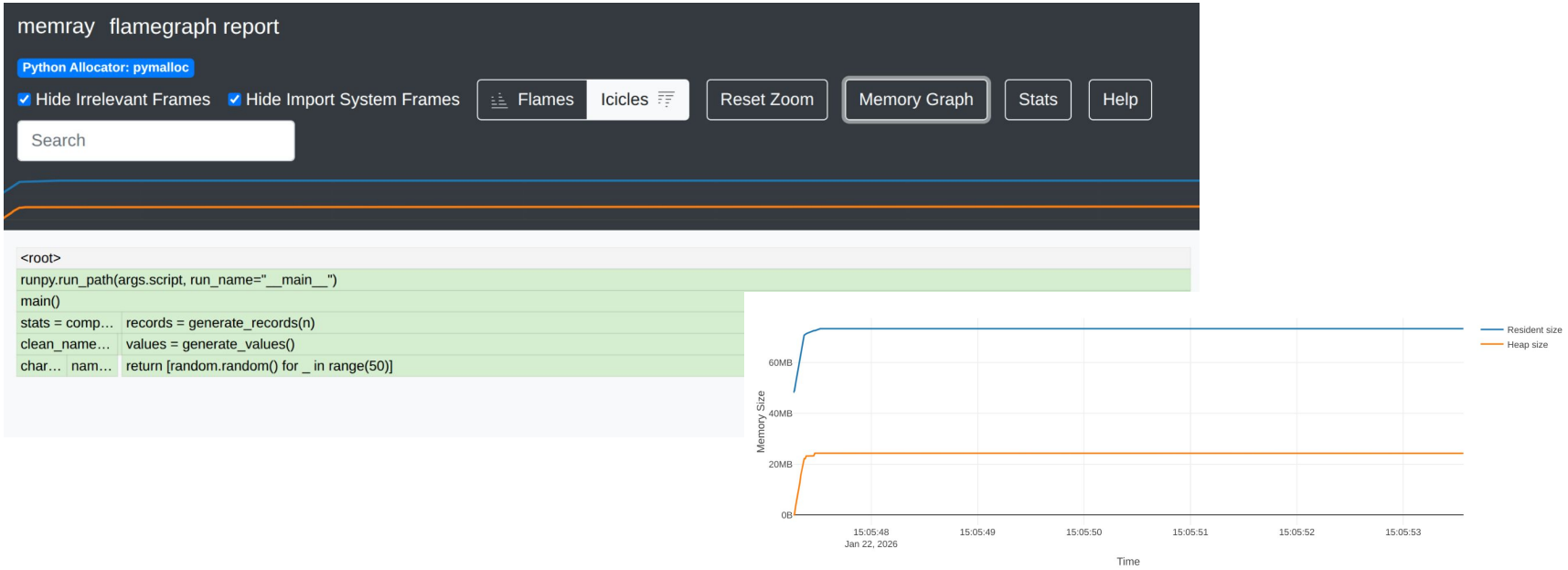
```
memray run -o output.bin script.py
```

Generate a flamegraph of the memory usage

```
memray flamegraph output.bin
```

memray

Flame graph and timeline of memory usage:



Other Python profilers

Many other python profilers are available, including

- austin
- py-spy
- yappi
- scalene
- viztracer
- ...

Each have their pros and cons, and we'll cover some of these later in the course - but the key concepts covered in this course apply to all of them.

Python profiling summary

- So far we looked at four python profiling tools
- Each allows us to understand performance of our code in different ways
- **cProfile**: flame graphs and call graphs
- **line_profiler**: profile individual lines of code within a function
- **pyinstrument**: call stack and timeline
- **memray**: memory allocation flame graph and timeline

These tools are often sufficient, even when your Python code is calling compiled or gpu code, to learn how best to improve the performance of your code.

But sometimes you need to go deeper...

Profiling multiprocessing code

Profiling multiprocessing code

If you use multiprocessing with “spawn” in your python code, then each spawned process is a new process with its own Python interpreter.

Typical profiling will not include any data from these other Python interpreters.

It is possible to use cProfile if you launch it within each sub-process, then manually combine the results at the end, but this is not very convenient.

A better option is to use a profiler that can automatically detect and profile any spawned sub-processes, such as py-spy or austin.

Profiling multiprocessing code: py-spy

py-spy

- Sampling (statistical) profiling tool in Python
- Can attach to a running process
- Can optionally profile subprocesses
- Can optionally also profile native (Cython/C/C++) extensions
- Note: Python 3.14 not yet supported (as of 2026/01/22)

Install with pip

```
pip install py-spy
```

Use py-spy record to generate an svg flame graph:

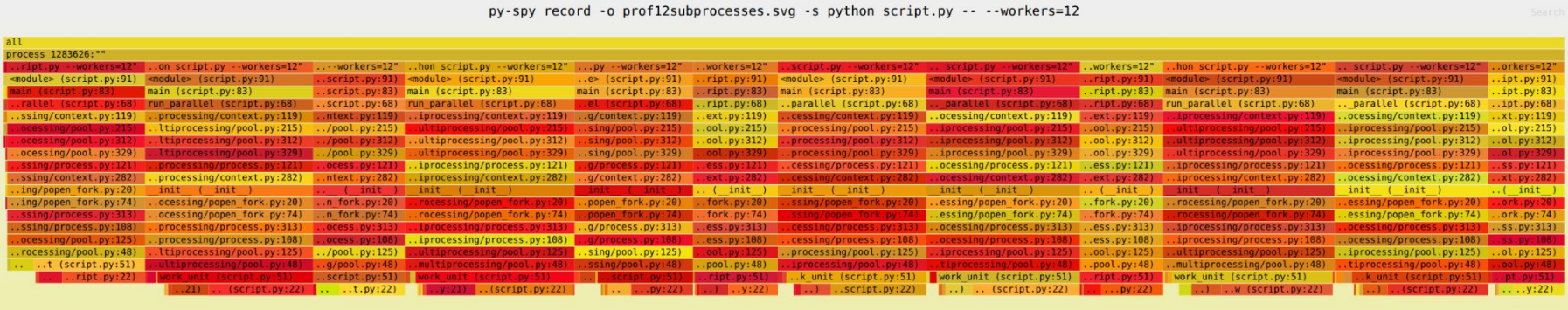
```
py-spy record -o profile.svg -- python script.py
```

Without subprocesses

Multiprocessing with 12 workers: normal profiling only shows module loading and argparse. All the multiprocessing work is done in subprocesses that are not visible to the profiler:

With subprocesses

With the `-s` or `--subprocess` flag, py-spy is able to profile all the spawned subprocesses, and now we see profiling data for all 12 workers in addition to the original Python process:



Profiling multiprocessing code: viztracer

Scientific Software Center

viztracer

- Sampling (statistical) tracing tool in Python
- Can attach to a running process
- Supports all variants of multithreading / multiprocessing

Install with pip

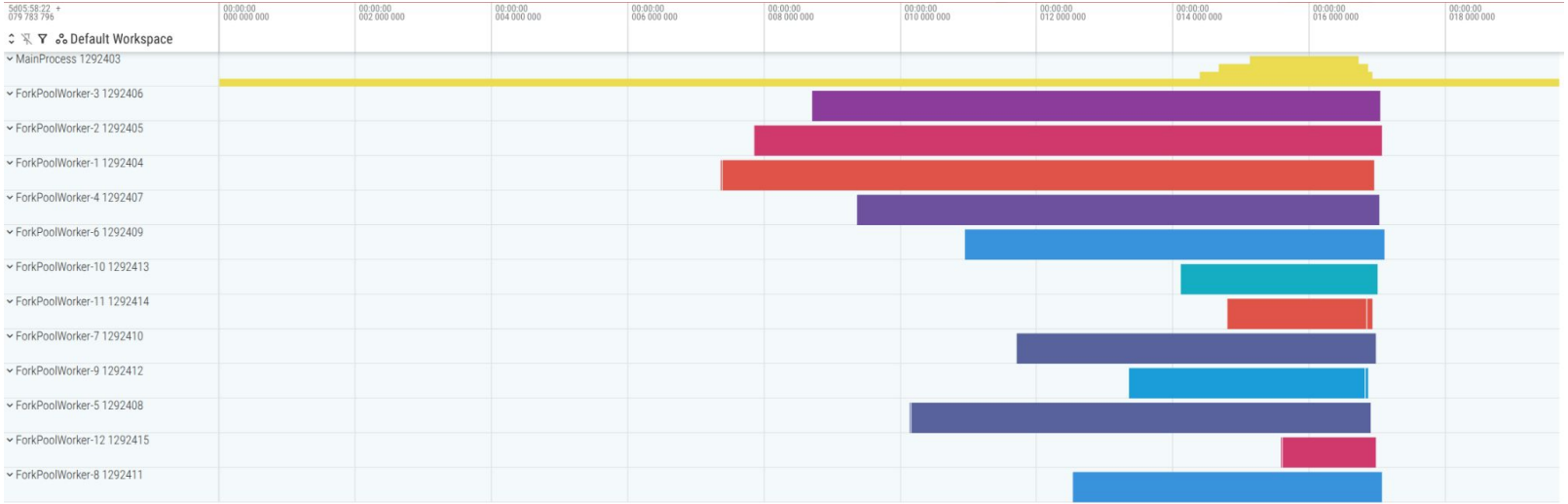
```
pip install viztracer
```

Use viztracer to generate a result.json trace file, then open with ui.perfetto.dev:

```
viztracer script.py
```

Multiprocessing traces

Trace output can be very informative when dealing with multithreaded code, for example to see which workers are active when. Here is a viztracer trace of the same code displayed using ui.perfetto.dev:



Profiling compiled code

Profiling native extensions

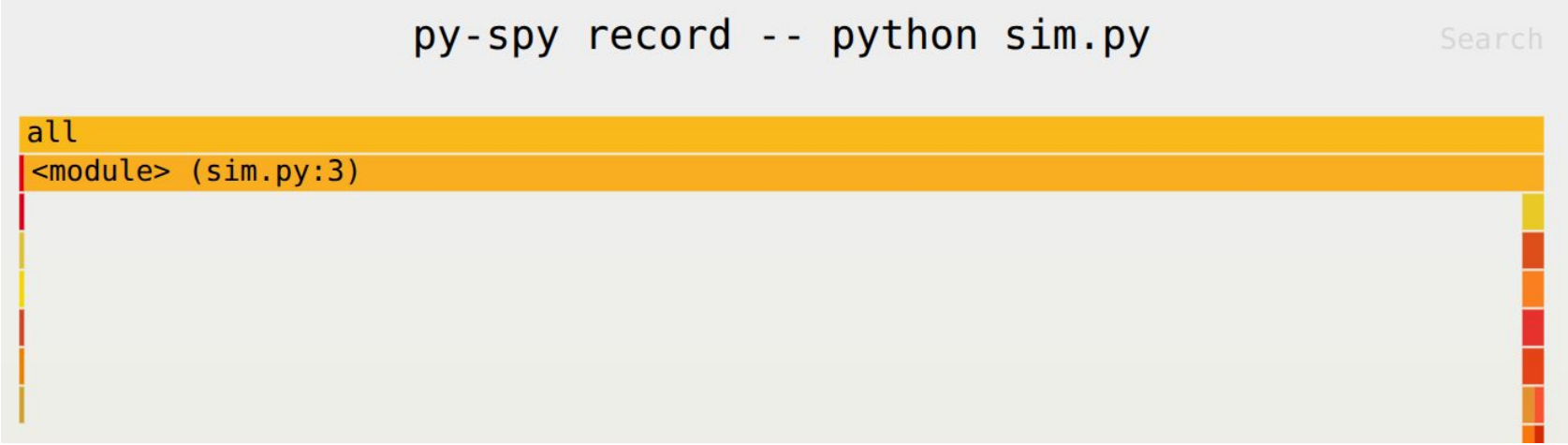
If your python code calls a compiled/native (Cython / C / C++) extension, the profilers we've looked at so far aren't able to see what the extension is doing.

Most of the time this is not a problem, since typically you didn't write the extension / aren't going to modify it, so there is no need to profile what is going on inside it - the only thing relevant to you is how long your Python code that calls the extension is running for.

If, however, you are developing your own compiled extension, then it can be helpful to see both Python and the compiled code in the profiling traces

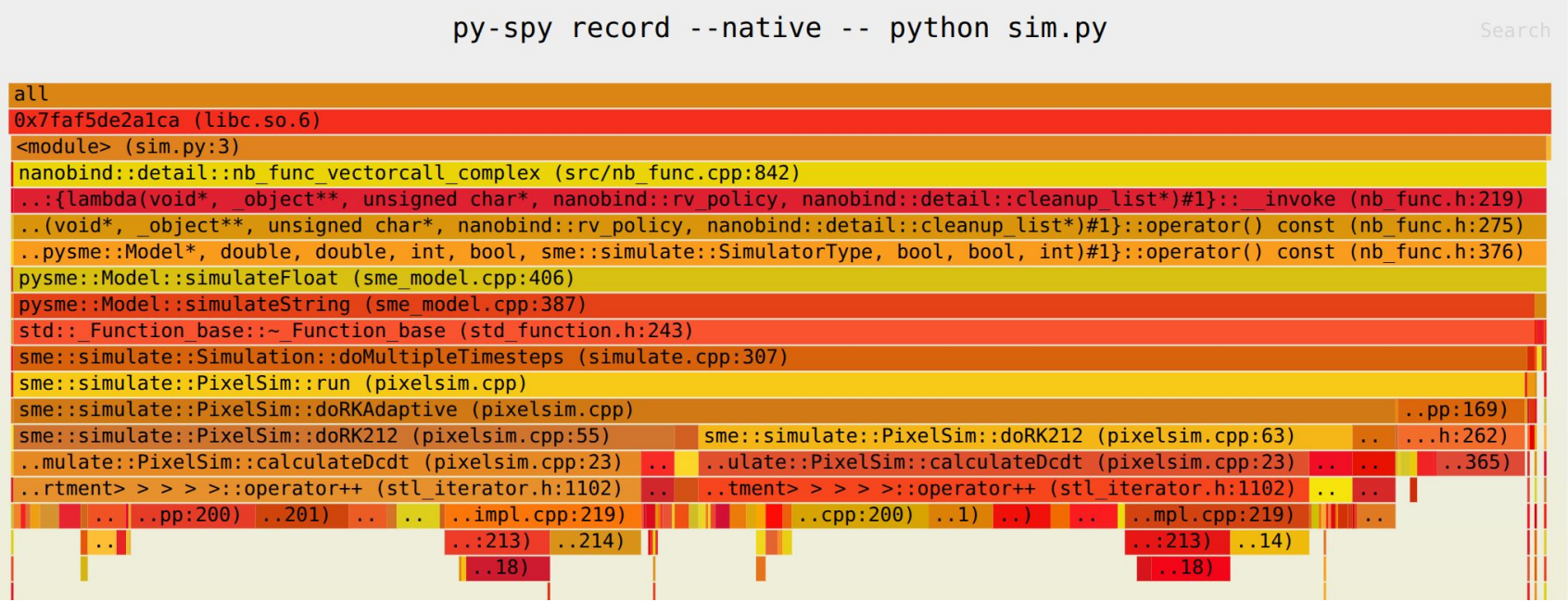
py-spy

If we run py-spy record on this script we see that all the time is spent in line 3 of sim.py, but since this calls a compiled extension, we can't dig any deeper:



py-spy --native

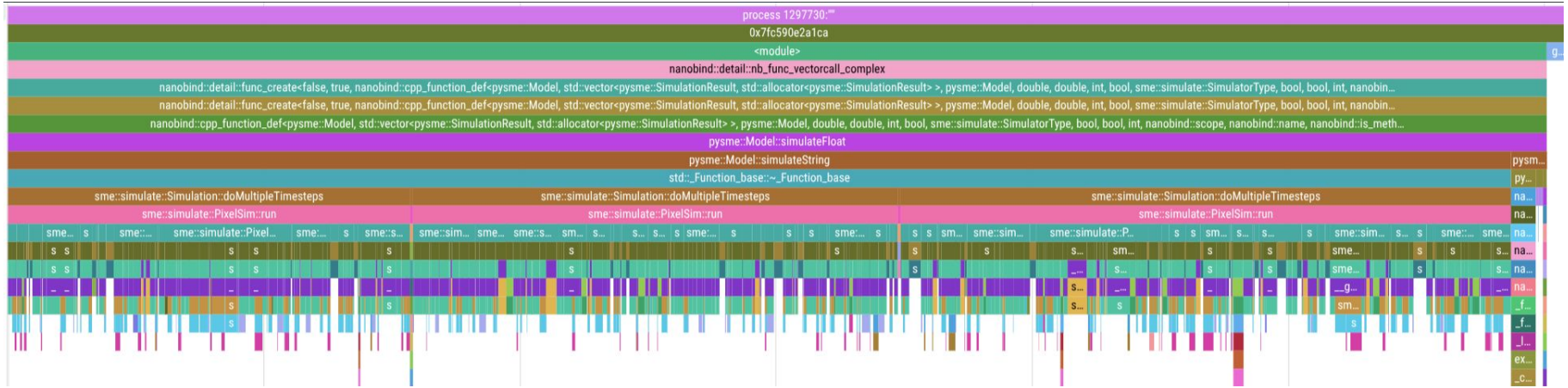
If we add the --native flag (and compile our extension with debugging symbols) then we can also see profiling data for the compiled functions:



py-spy --native -f chrometrace

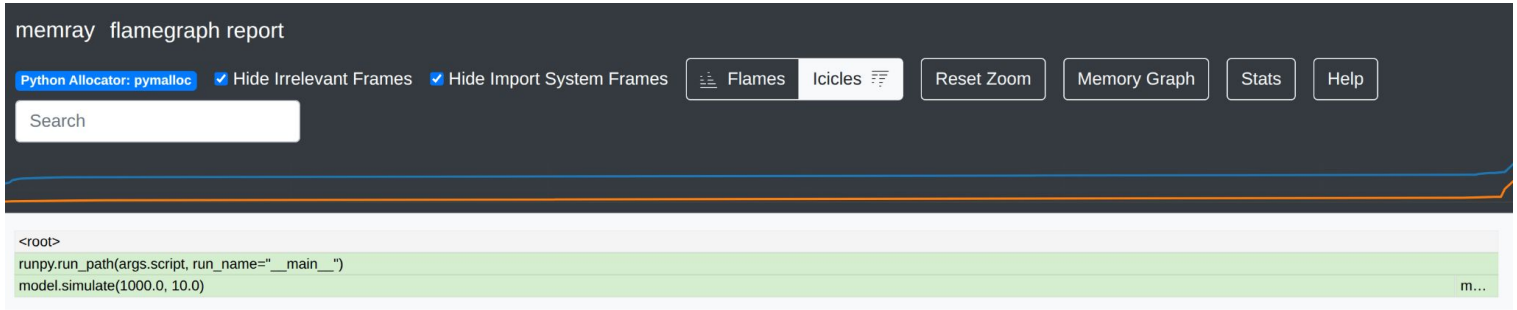
With `-f chrometrace` we generate a trace that we can view in ui.perfetto.dev:

```
py-spy record -s -n -o trace.json -f chrometrace python sim.py
```



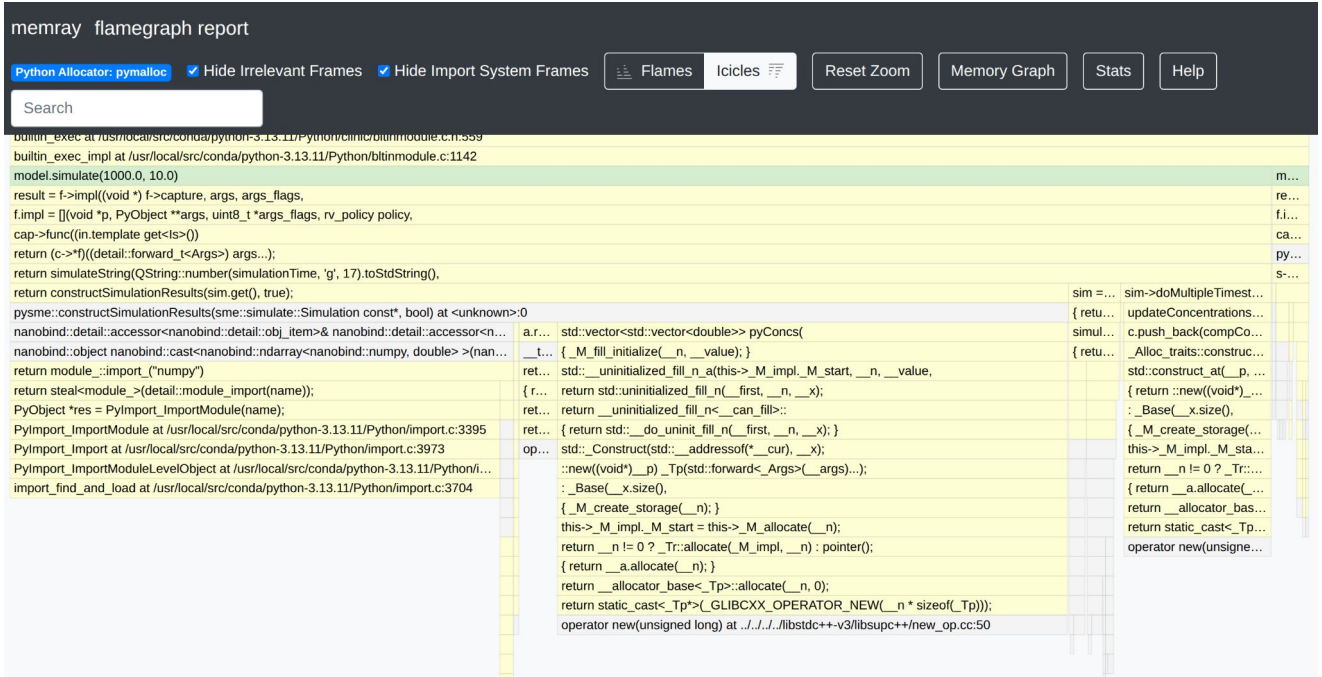
Memory use of native extensions

If we use memray as we did before with this script, we see that model.simulate() allocates a bunch of memory, but there is no more information since all these allocations occur within the compiled module:



Memory use of native extensions

If we call memray with the --native option we also see compiled code memory info:



Profiling GPU code

torch.profiler: CPU

To use the pytorch profiler, we need to modify our code and wrap the bit we want to profile with a profile context manager (in the same way we did with cProfile).

For example, to profile the CPU activity:

```
from torch.profiler import profile, ProfilerActivity
with profile(activities=[ProfilerActivity.CPU]) as prof:
    # ...do some work here...
print(prof.key_averages().table(sort_by="cpu_time_total"))
```

torch.profiler: CPU output

The output is a table of functions with their self and total CPU times.

Here self = tottime in cProfile, and total = cumtime in cProfile

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls
aten::linear	0.04%	32.891us	76.18%	59.661ms	29.831ms	2
aten::addmm	60.71%	47.547ms	76.05%	59.557ms	29.778ms	2
aten::relu	0.07%	51.910us	23.81%	18.649ms	18.649ms	1
aten::clamp_min	0.08%	62.841us	23.75%	18.597ms	18.597ms	1
cudaLaunchKernel	23.69%	18.550ms	23.69%	18.550ms	6.183ms	3
cudaGetSymbolAddress	8.28%	6.488ms	8.28%	6.488ms	6.488ms	1
cudaOccupancyMaxActiveBlocksPerMultiprocessor	4.04%	3.164ms	4.04%	3.164ms	1.055ms	3
cudaFree	1.35%	1.060ms	1.35%	1.060ms	1.060ms	1
cudaDeviceGetAttribute	1.28%	1.006ms	1.28%	1.006ms	47.903us	21
cudaMalloc	0.33%	261.292us	0.33%	261.292us	65.323us	4
aten::t	0.05%	43.490us	0.09%	71.340us	35.670us	2

torch.profiler: tensor shapes

It can also be helpful to see the shapes of the tensor inputs to these functions, which can be done by setting `record_shapes=True`:

```
from torch.profiler import profile, ProfilerActivity
with profile(activities=[ProfilerActivity.CPU], record_shapes=True) as prof:
    # ...do some work here...
print(prof.key_averages(group_by_input_shape=True).table())
```

torch.profiler: tensor shapes output

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls	Input Shapes
aten::linear	0.04%	28.440us	76.06%	58.742ms	58.742ms	1	[[512, 1024], [2048, 1024], [2048]]
aten::t	0.05%	35.541us	0.07%	56.231us	56.231us	1	[[2048, 1024]]
aten::transpose	0.02%	11.600us	0.03%	20.690us	20.690us	1	[[2048, 1024], [], []]
aten::as_strided	0.01%	9.090us	0.01%	9.090us	9.090us	1	[[2048, 1024], [], [], []]
aten::addmm	60.42%	46.662ms	75.95%	58.658ms	58.658ms	1	[[2048], [512, 1024], [1024, 2048], [], []]
cudaStreamIsCapturing	0.00%	3.570us	0.00%	3.570us	3.570us	1	[]
cudaMalloc	0.37%	283.075us	0.37%	283.075us	70.769us	4	[]
cudaFree	1.38%	1.064ms	1.38%	1.064ms	1.064ms	1	[]
cudaDeviceGetAttribute	1.32%	1.022ms	1.32%	1.022ms	48.657us	21	[]
cudaGetDriverEntryPoint	0.00%	0.590us	0.00%	0.590us	0.295us	2	[]
cudaGetSymbolAddress	8.42%	6.505ms	8.42%	6.505ms	6.505ms	1	[]
cudaOccupancyMaxActiveBlocksPerMultiprocessor	4.02%	3.102ms	4.02%	3.102ms	1.034ms	3	[]
cudaMemsetAsync	0.02%	11.850us	0.02%	11.850us	5.925us	2	[]
cudaLaunchKernel	23.65%	18.265ms	23.65%	18.265ms	6.088ms	3	[]
aten::relu	0.08%	58.301us	23.79%	18.376ms	18.376ms	1	[[512, 2048]]
aten::clamp_min	0.09%	67.901us	23.72%	18.318ms	18.318ms	1	[[512, 2048], []]
aten::linear	0.01%	4.920us	0.14%	106.171us	106.171us	1	[[512, 2048], [1024, 2048], [1024]]
aten::t	0.01%	8.560us	0.03%	21.790us	21.790us	1	[[1024, 2048]]
aten::transpose	0.01%	9.070us	0.02%	13.230us	13.230us	1	[[1024, 2048], [], []]
aten::as_strided	0.01%	4.160us	0.01%	4.160us	4.160us	1	[[1024, 2048], [], [], []]
aten::addmm	0.09%	67.151us	0.10%	79.461us	79.461us	1	[[1024], [512, 2048], [2048, 1024], [], []]
cudaDeviceSynchronize	0.01%	5.910us	0.01%	5.910us	5.910us	1	[]

Self CPU time total: 77.230ms

torch.profiler: GPU

If we are using a gpu then we can add CUDA (for nvidia GPUs) or XPU (for Intel gpus) to the activities we want to profile:

```
from torch.profiler import profile, ProfilerActivity

with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA]) as prof:

    # ...do some work...

print(prof.key_averages().table(sort_by="cuda_time_total"))
```

torch.profiler: GPU output

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	# of Calls
aten::linear	0.04%	35.261us	76.36%	59.883ms	29.941ms	0.000us	0.00%	460.961us	230.481us	2
aten::addmm	57.87%	45.387ms	76.22%	59.773ms	29.887ms	120.929us	96.92%	460.961us	230.481us	2
Runtime Triggered Module Loading	27.67%	21.696ms	27.67%	21.696ms	3.616ms	234.368us	187.84%	234.368us	39.061us	6
cudaOccupancyMaxActiveBlocksPerMultiprocessor	0.02%	12.140us	4.03%	3.162ms	1.054ms	0.000us	0.00%	170.016us	56.672us	3
ampere_sgemm_128x64_tn	0.00%	0.000us	0.00%	0.000us	0.000us	119.969us	96.15%	119.969us	59.985us	2
cudaGetSymbolAddress	8.13%	6.376ms	8.42%	6.602ms	6.602ms	0.000us	0.00%	113.344us	113.344us	1
Lazy Function Loading	0.08%	60.722us	0.08%	60.722us	30.361us	60.512us	48.50%	60.512us	30.256us	2
Activity Buffer Request	2.62%	2.053ms	2.62%	2.053ms	2.053ms	56.672us	45.42%	56.672us	56.672us	1
aten::relu	0.06%	47.631us	23.63%	18.535ms	18.535ms	0.000us	0.00%	15.360us	15.360us	1
aten::clamp_min	0.10%	80.391us	23.57%	18.487ms	18.487ms	3.840us	3.08%	15.360us	15.360us	1
cudaLaunchKernel	0.06%	50.281us	23.50%	18.432ms	6.144ms	0.000us	0.00%	11.520us	3.840us	3
void at::native::vectorized_elementwise_kernel<4, at...	0.00%	0.000us	0.00%	0.000us	0.000us	3.840us	3.08%	3.840us	3.840us	1
Memset (Device)	0.00%	0.000us	0.00%	0.000us	0.000us	0.960us	0.77%	0.960us	0.480us	2
aten::t	0.06%	45.450us	0.09%	73.850us	36.925us	0.000us	0.00%	0.000us	0.000us	2
aten::transpose	0.03%	21.140us	0.04%	28.400us	14.200us	0.000us	0.00%	0.000us	0.000us	2
aten::as_strided	0.01%	7.260us	0.01%	7.260us	3.630us	0.000us	0.00%	0.000us	0.000us	2
cudaStreamIsCapturing	0.00%	3.020us	0.00%	3.020us	3.020us	0.000us	0.00%	0.000us	0.000us	1
cudaMalloc	0.35%	272.325us	0.35%	272.325us	68.081us	0.000us	0.00%	0.000us	0.000us	4
cudaFree	1.60%	1.251ms	1.60%	1.251ms	1.251ms	0.000us	0.00%	0.000us	0.000us	1
cudaDeviceGetAttribute	1.28%	1.001ms	1.28%	1.001ms	47.665us	0.000us	0.00%	0.000us	0.000us	21
cudaGetDriverEntryPoint	0.00%	0.560us	0.00%	0.560us	0.280us	0.000us	0.00%	0.000us	0.000us	2
cudaMemsetAsync	0.02%	15.360us	0.02%	15.360us	7.680us	0.000us	0.00%	0.000us	0.000us	2
cudaDeviceSynchronize	0.01%	7.100us	0.01%	7.100us	3.550us	0.000us	0.00%	0.000us	0.000us	2

torch.profiler: memory

To also see the RAM used by functions, set `profile_memory=True`:

```
from torch.profiler import profile

with profile(profile_memory=True) as prof:
    # ...do some work here…

print(prof.key_averages().table(sort_by="cuda_memory_usage"))
```

torch.profiler: memory output

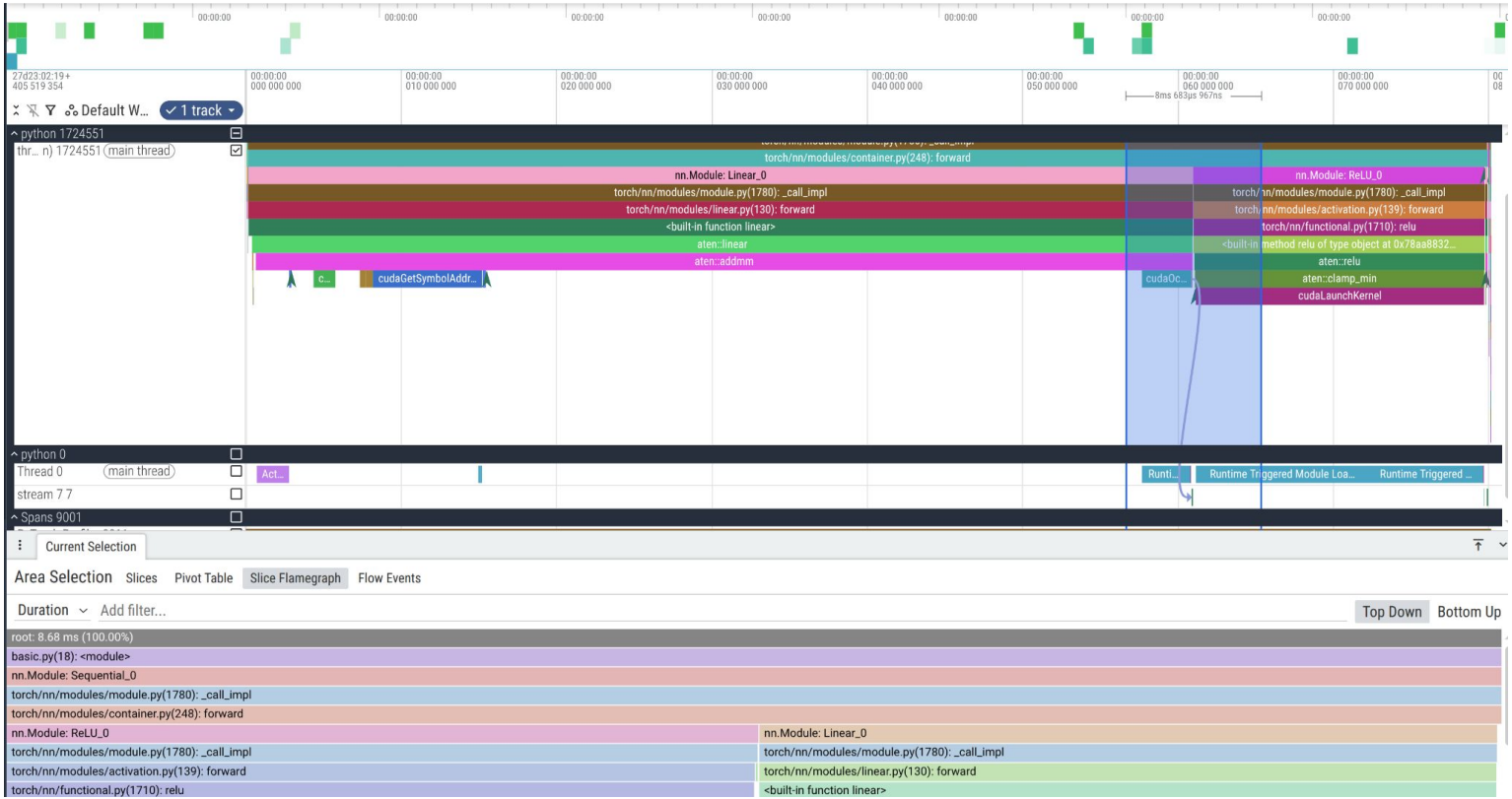
Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	CPU Mem	Self CPU Mem	CUDA Mem	Self CUDA Mem	# of Call
aten::linear	0.05%	35.150us	77.33%	59.810ms	29.905ms	0.000us	0.00%	466.600us	233.300us	0 B	0 B	15.12 MB	0 B	2
aten::addmm	58.47%	45.226ms	77.18%	59.697ms	29.849ms	121.570us	97.04%	466.600us	233.300us	0 B	0 B	15.12 MB	15.12 MB	2
aten::relu	0.06%	47.511us	22.66%	17.529ms	17.529ms	0.000us	0.00%	14.848us	14.848us	0 B	0 B	4.00 MB	0 B	1
aten::clamp_min	0.10%	75.231us	22.60%	17.482ms	17.482ms	3.712us	2.96%	14.848us	14.848us	0 B	0 B	4.00 MB	4.00 MB	1
aten::t	0.06%	47.481us	0.10%	77.031us	38.516us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	2
aten::transpose	0.03%	20.700us	0.04%	29.550us	14.775us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	2
aten::as_strided	0.01%	8.850us	0.01%	8.850us	4.425us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	2
Activity Buffer Request	2.69%	2.083ms	2.69%	2.083ms	2.083ms	57.505us	45.90%	57.505us	57.505us	0 B	0 B	0 B	0 B	1
cudaStreamIsCapturing	0.00%	3.050us	0.00%	3.050us	3.050us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	1
cudaMalloc	0.37%	288.033us	0.37%	288.033us	72.008us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	4
cudaFree	1.68%	1.298ms	1.68%	1.298ms	1.298ms	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	1
cudaDeviceGetAttribute	1.30%	1.008ms	1.30%	1.008ms	47.979us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	21
cudaGetDriverEntryPoint	0.00%	0.620us	0.00%	0.620us	0.310us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	2
cudaGetSymbolAddress	8.21%	6.347ms	8.50%	6.572ms	6.572ms	0.000us	0.00%	115.010us	115.010us	0 B	0 B	0 B	0 B	1
Runtime Triggered Module Loading	26.77%	20.709ms	26.77%	20.709ms	3.452ms	237.444us	189.53%	237.444us	39.574us	0 B	0 B	0 B	0 B	6
cudaOccupancyMaxActiveBlocksPerMultiprocessor	0.02%	13.490us	4.11%	3.178ms	1.059ms	0.000us	0.00%	172.515us	57.505us	0 B	0 B	0 B	0 B	3
Lazy Function Loading	0.08%	62.050us	0.08%	62.050us	31.025us	61.217us	48.86%	61.217us	30.609us	0 B	0 B	0 B	0 B	2
cudaMemsetAsync	0.02%	15.990us	0.02%	15.990us	7.995us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	2
Memset (Device)	0.00%	0.000us	0.00%	0.000us	0.000us	0.992us	0.79%	0.992us	0.496us	0 B	0 B	0 B	0 B	2
cudaLaunchKernel	0.06%	50.201us	22.54%	17.431ms	5.810ms	0.000us	0.00%	11.136us	3.712us	0 B	0 B	0 B	0 B	3
ampere_sgemm_128x64_tn	0.00%	0.000us	0.00%	0.000us	0.000us	120.578us	96.25%	120.578us	60.289us	0 B	0 B	0 B	0 B	2
void at::native::vectorized_elementwise_kernel<4, at...	0.00%	0.000us	0.00%	0.000us	0.000us	3.712us	2.96%	3.712us	3.712us	0 B	0 B	0 B	0 B	1
cudaDeviceSynchronize	0.01%	6.520us	0.01%	6.520us	3.260us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	0 B	0 B	2
[memory]	0.00%	0.000us	0.00%	0.000us	0.000us	0.000us	0.00%	0.000us	0.000us	0 B	0 B	-10.00 MB	-10.00 MB	3

torch.profiler: Traces

The torch profiler can also output traces that can be viewed in ui.perfetto.dev:

```
from torch.profiler import profile, ProfilerActivity

with profile(
  activities = [ProfilerActivity.CPU, ProfilerActivity.CUDA],
  profile_memory = True,
  record_shapes = True,
  with_stack = True
) as prof:
    # ...do some work...
prof.export_chrome_trace("trace.json")
```

torch.profiler: Traces output



torch.profiler: schedule

Scientific Software Center

Often you don't want to profile everything - for example in a long training run you may just want to profile a couple of training steps. You can use `profiler.schedule` for this:

```
def trace_handler(p):
    p.export_chrome_trace(f"/tmp/trace_{p.step_num}.json")

with profile(
    activities=activities,
    schedule=torch.profiler.schedule(wait=5, warmup=1, active=1, repeat=3),
    on_trace_ready=trace_handler,
) as p:
    for idx in range(100):
        # ...model training step...
        p.step()
```

HolisticTraceAnalysis

If you are training models using many GPU nodes, then profiling (and understanding the profiling data) becomes more challenging.

Holistic Trace Analysis (HTA) is a library that may be useful for this:

hta.readthedocs.io

They also provide example Jupyter Notebooks:

github.com/facebookresearch/HolisticTraceAnalysis/tree/main/examples

(I haven't used it though)

JAX

If you are using jax, it also has a profiler that can generate json traces that you can then view using in ui.perfetto.dev:

```
import jax


f = jax.jit(lambda x: x @ x)
x = jax.random.normal(jax.random.key(0), (4096, 4096))
f(x).block_until_ready()


with jax.profiler.trace("/tmp/jax_trace.json"):
    y = f(x)
    y.block_until_ready()
```

Hands on profiling

Hands on profiling

If you haven't already done so, clone the repo and install the dependencies:

```
git clone https://github.com/ssciwr/python-performance-profiling
```

```
cd python-performance-profiling
```

```
pip install -r requirements.txt
```

Scientific Software Center

Hands on: example1

Example 1: pipeline.py

- The script takes a few seconds to run:
  - `time python pipeline.py`
- There are some tests included:
  - `python -m pytest`
- Profile with cProfile:
  - inspect console output
    - python -m cProfile
  - snakeviz
  - call graph output
- What is the current bottleneck?

Example 1: pipeline_opt.py

Main bottleneck: `count_similar_record_pairs`

- More than 90% of the execution time is spent in this function
- O(n^2) algorithm compares means of all pairs of records
- Can replace with O(n log n) if we first sort the means

...

- Run tests to ensure our implementation is correct
- Re-run profiling - is that already fast enough? If so no need to profile further!
    - Otherwise look for the next bottleneck

Example 1: pipeline_opt2.py

- Python script loading and module import now non-negligible
  - Consider increasing n so that the main pipeline dominates the runtime
- New bottleneck is now `compute_statistics`
  - In particular the `normalize_name` function
- Use line_profiler
  - no obvious bottleneck here - instead “death by a thousand cuts”
- Replace function with call to specialized library like PyICU or Unidecode
  - Compiled specialized libraries - typically significantly faster than pure Python implementation
- Re-run profiling - is that already fast enough? If so no need to profile further!
  - Otherwise look for the next bottleneck

Example 1: pipeline_opt3.py

- Variance and mean calculations now the bottleneck
  - Consider increasing n so that the main pipeline dominates the runtime
- We calculate the mean twice, once for the mean, once for the variance
  - This is wasted computation
- Combine in one function that calculates both together
  - Same result with less work
- Could/should consider using a library e.g. numpy for this
  - But would need to refactor to use numpy arrays everywhere to avoid conversion costs
- Re-run profiling - is that already fast enough? If so no need to profile further!
  - Otherwise look for the next bottleneck

Example 1: possible further steps

With each new round of profiling, the first question to ask is if you are done.

The gains typically go down after the first few rounds, while the costs (e.g. difficulty of implementation, testing and code maintenance) typically increase.

You also need to take care that you are not “over-fitting” to your test case, but that the improvements translate to real use of the code with real data.

Further steps here could include

- Using numpy types instead of python lists
- Using numba to just-in-time compile numerical code
- Porting this script to a faster language like rust or c++

Hands on: example2

Example 2

More pure Python code: reads a text file, modifies it, writes it to an output file.

- Generate some test data
  - `bash generate_data.sh`
- Run the code
  - `python fix_headers.py test.fastq out.fastq`
- Profile the code
  - Compare cProfile, pyinstrument and py-spy
  - Use line_profiler
  - Use memray, see how the memory use changes with the data size

Scientific Software Center

Hands on: example3

Example 3

Python multiprocessing code that does some calculations

- Run the code using 4 workers
  - `python script.py --workers=4`
- Profile the code
  - Use py-spy to generate a flame graph
  - Use viztracer to generate a trace and view using ui.perfetto.dev
  - Use the `--ignore_c_function` option for viztracer to reduce trace size
- Let's pretend the work_unit function is already optimised
- Investigate what happens as we change
  - --size
  - --tasks
  - --workers
  - --chunksize

Example 3 multiprocessing scenarios

- Good parallel performance
  - `--workers 4 --tasks 32 --size 300000 --chunksize 1`
- Overhead dominates (tasks too small - actually gets slower with more workers)
  - `--workers 4 --tasks 50000 --size 10 --chunksize 1`
- Waiting for one worker to finish (too few / too different tasks)
  - `--workers 4 --tasks 10 --size 500000 --chunksize 1`
- Balance vs overhead tradeoff (smaller chunks = better balance but more overhead)
  - `--workers 4 --tasks 120000 --size 500 --chunksize 1`
  - `--workers 4 --tasks 120000 --size 500 --chunksize 250`
  - `--workers 4 --tasks 120000 --size 500 --chunksize 12000`

Hands on: example4

Example 4

Pytorch code that does some calculations

- Install pytorch: pytorch.org/get-started/locally
- Run the code
  - `python calc.py --workers=4`
- Inspect the trace using ui.perfetto.dev
- Compare with calc_batched.py
- If you don’t have a gpu, use the included example traces
- If you do have a gpu, try different options for the profiler
  - Scheduler
  - Label sections

Summary

Summary

In this course we covered:

- What profiling is and why it is useful
- Different ways to profile your code
- Profiling CPU use of Python code
- Profiling CPU use of compiled extensions
- Profiling GPU use of pytorch / jax code
- Profiling memory use
- Hands on profiling examples

Next steps

- Profile your code!
  - Pick a profiler and try it, e.g. `pyinstrument -r html script.py`
- Try out a few different profilers
  - Compare your results with e.g. `cProfile` or `py-spy`
- Get familiar with ui.perfetto.dev
  - Many profilers (not just Python ones) can export traces to be displayed here
- SSC consultations
  - Make an appointment by email: ssc@uni-heidelberg.de