Modern Approaches to Profiling in Python with Scalene

Tyler Collins
High Performance Computing Consultant
SHARCNET, Brock University
May 3rd, 2023

Wiki note

Before we go any further…

Always always always check the wiki

No, really… please

https://docs.alliancecan.ca/wiki/Technical_documentation

If you can’t find something, contact us!

help@sharcnet.ca

Outline

- We love Python but it’s slow…
- Why people don’t just switch away from Python?
- What do people do when they need to go faster?
- Brief mention of profilers
- Introduce Scalene
  - Some nerdy details!
- Live demos and toy problems
- Questions

Hopefully at the end of this talk, you will use Scalene in your own projects!

This webinar and its materials can be found on my GitHub

Why Python?

- Let me count the ways:
    - Easier learning curve, scikit-learn, NumPy, Pandas, plotting, calling other languages, rapid prototyping, Jupyter Notebooks,
- Awesome isn't free
    - "Each abstraction must provide more benefit than cost"

But my $PERSON (friend, co-worker, supervisor, etc) says it's slow…

They're right!

Will I ever need it to be faster? Does performance matter?

Does performance matter?

Yes

Does performance really matter?

- How many simulations do you need to run?
  - 30? 100? Billions?
- Memory footprint is often a limiting factor of how many things can be run at once
  - True for cloud computing also via “flavours”
- Consider scheduler priority
  - If you are going to run jobs where there is “work”, it should minimize waste
- Performance matters once you go to HPC scale

TL;DR: Still yes

How do I avoid Python's slowness?

Caution - Nerd stuff ahead:

- What's an integer look like in Python?
  - C: 4 bytes – Python: 24 bytes
- How about List access?
  - C: Read memory at given address plus offset and return
  - Python: Make sure the variable indexing the list is numeric, determine if it's within bounds, if negative - do some wrap around magic, more

Not so fun facts:

- More statically typed languages are faster
- Interpreted languages are slower due to lack of compiled optimizations

TL;DR: You can't, without using other languages

Why don’t people swap languages?

Fun facts:

- Some people are not compute science students
  - Some just take a **single semester course** and do their best from there
  - Setup/installation is typically done for students in these environments
- Concepts like memory management, namespaces, threads, and more are found to be challenging
- **Big one:** lack of “workspace” style setup like Jupyter
  - Ask me about this one during the question period for a free rant
- Prototyping is just plain easier
- Codebase or package requirements

Not everyone can (or should) just “swap languages” when the situation calls for it!

So then what do others do?

In a rough order of easy to hard in my own opinion - with demos later:

1. Just-in-time (JIT) support via something like Numba
   a. Limited indirect code annotation
2. Cython
   a. Direct code annotation
3. NumPy and vectorization
   a. Coding style change

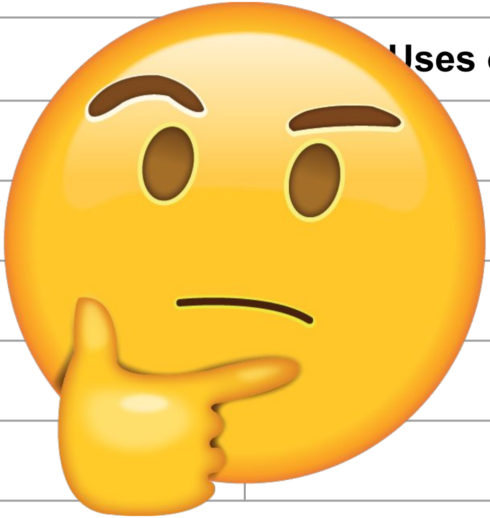
For other bigger problems: scikit, PyTorch, TensorFlow

What do these have in common?

Package:	Uses or implemented with:
NumPy	C/C++/Fortran
Cython	C/C++
Numba	NumPy
Scikit-learn	NumPy, Cython
PyTorch	C++

What do these have in common?

Package:	Uses or implemented with:
NumPy	C/C++/Fortran
Cython	C/C++
Numba	NumPy
Scikit-learn	NumPy, Cython
PyTorch	C++



What do these have in common?

It’s C all the way down!

All of these packages let you continue to write and leverage Python, while taking advantage of the speed of a more “Native” language

More Native - more better

Why are native languages faster?

The short story based on what we've discussed so far:

1. They're often compiled, and can take advantage of optimizations for specific hardware
2. Hyper specific control of memory and its access with less waste
3. Stronger typing means less sanity checking along the way

Many many more things...

How do we measure packages like these?

- … Print line debugging!
  - We all do it
- Time libraries
  - Problematic for Nerd Reasons™
- Watch htop
  - Not going to work for long execution times
- **Profilers**
  - Something which wraps around your code and gives you statistics about how it ran or what it did
  - Where we will focus the rest of this talk!

Unsurprising, as this talk is about a specific profiler

On the topic of profilers...

Some concerns:

- Cost overhead of using a profiler
  - Runtime, memory, and other overhead costs
- Accuracy
  - You never want to optimize the wrong section of your code
- Which one do you even use?
  - CPU profilers are blue box, memory are green box

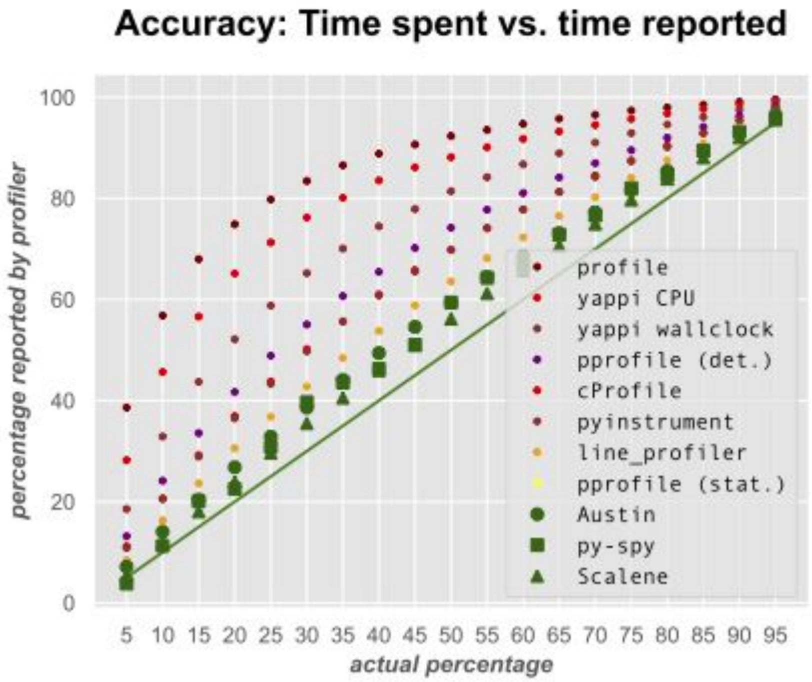
Note: memory_profiler on a day long job could turn it into a 270 day job

Profiler	Slowdown
pprofile (stat.)	1×
py-spy	1×
pyinstrument	1.2×
cProfile	1.4×
yappi wallclock	1.9×
yappi CPU	3×
line_profiler	6×
Profile	13.7×
pprofile (det.)	40×
fil	2×
memory_profiler	270×
memray	2.4×

Disclaimer: Table from Scalene documentation

On the topic of profilers...

Accuracy is a bigger concern that you might think...



Disclaimer: Figure from Scalene documentation

Introducing: Scalene

- CPU profiling
  - Time spent in Native vs Python
- Memory profiling
  - Can detect leaks, and copying problems
- Both function AND line profiling
- No need for decorators
- High accuracy
- Advanced supported features
  - threads, multiprocessing library, GPU, trends

Newer versions have built-in OpenAI suggestions support

Introducing: Scalene

Profiler	Slowdown	Lines or Functions	Unmodified Code	Threads	Multi-processing	Python vs. C Time	System Time	Profiles Memory	Python vs. C Memory	GPU	Memory Trends	Copy Volume	Detects Leaks
CPU-only profilers													
pprofile (stat.)	1×	lines	✓	✓	-	-	-	-	-	-	-	-	-
py-spy	1×	lines	✓	✓	-	-	-	-	-	-	-	-	-
pyinstrument	1.2×	functions	✓	-	-	-	-	-	-	-	-	-	-
cProfile	1.4×	functions	✓	-	-	-	-	-	-	-	-	-	-
yappi wallclock	1.9×	functions	✓	✓	-	-	-	-	-	-	-	-	-
yappi CPU	3×	functions	✓	✓	-	-	-	-	-	-	-	-	-
line_profiler	6×	lines	-	-	-	-	-	-	-	-	-	-	-
Profile	13.7×	functions	✓	-	-	-	-	-	-	-	-	-	-
pprofile (det.)	40×	lines	✓	✓	-	-	-	-	-	-	-	-	-
memory-only profilers													
fil	2×	lines	-	-	-	-	-	*peak only*	-	-	-	-	-
memory_profiler	270×	lines	-	-	-	-	-	*RSS*	-	-	-	-	-
memray	2.4×	lines	-	✓	-	-	-	*peak only*	✓	-	-	-	-
CPU+memory profilers													
Austin (CPU+mem)	1×	lines	✓	✓	-	-	-	*RSS*	-	-	-	-	-
Scalene	**1.4×**	**both**	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Disclaimer: Table from Scalene documentation

How does that even work?

Caution - Nerd stuff ahead:

- Memory: Python likes to allocate memory
  - Don’t track every malloc/free, track only deltas of ~1MB
- Memory leaks: Python can still memory leak, and now we involve C/C++
  - Token based tracking with sampling technique
- CPU Time: Normally you can’t get timings on external Python calls
  - Track process timestamps, any large difference in delta is time spent in native code
  - More Native more better!

Installing Scalene

Run the following in any environment:

- pip install scalene

Done - just to be sure, make sure it's grabbing the latest version to match my examples!

- pip install scalene==1.5.21.2

Problem 1

Here's our first example with NumPy:

```
import numpy as np

def main():
    for i in range(10):
        x = np.array(range(10**7))
        y = np.array(np.random.uniform(0, 100, size=(10**(8))))
```

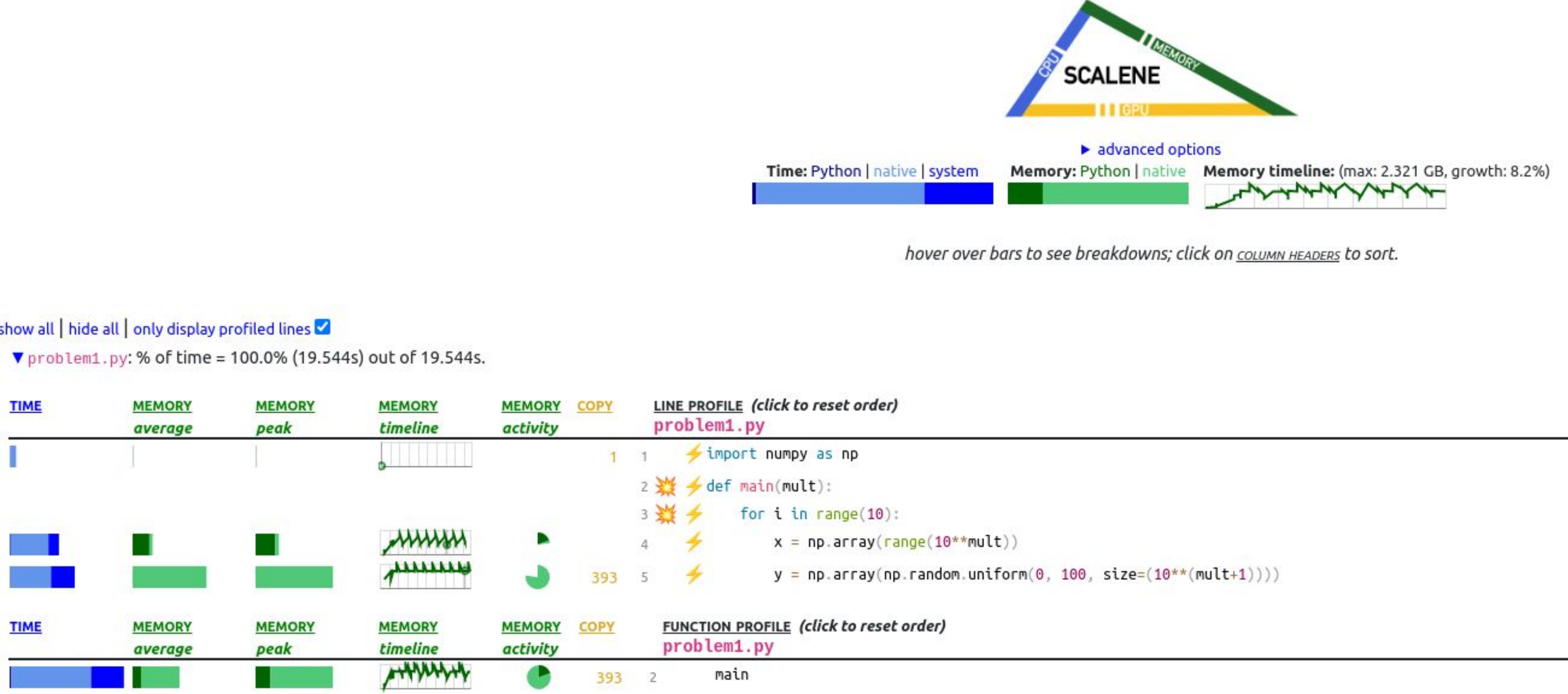
Problem 1

Assuming it’s called “problem1.py”, run Scalene with:

- scalene problem1.py

You should have a new tab/window in your web browser displaying the profiled version of your code

Note: If you are running on a remote server, your mileage may vary



CPU
MEMORY
GPU
SCALENE
advanced options
Time: Python | native | system
Memory: Python | native
Memory timeline: (max: 2.321 GB, growth: 8.2%)
hover over bars to see breakdowns; click on COLUMN HEADERS to sort.
show all | hide all | only display profiled lines
problem1.py: % of time = 100.0% (19.544s) out of 19.544s.
TIME
MEMORY average
MEMORY peak
MEMORY timeline
MEMORY activity
COPY
LINE PROFILE (click to reset order)
problem1.py
1 1 import numpy as np
2 def main(mult):
3 for i in range(10):
4 x = np.array(range(10**mult))
393 5 y = np.array(np.random.uniform(0, 100, size=(10**(mult+1))))
TIME
MEMORY average
MEMORY peak
MEMORY timeline
MEMORY activity
COPY
FUNCTION PROFILE (click to reset order)
problem1.py
393 2 main

SCALENE

▶ advanced options

Time: Python | native | system **Memory:** Python | native **Memory timeline:** (max: 2.321 GB, growth: 8.2%)

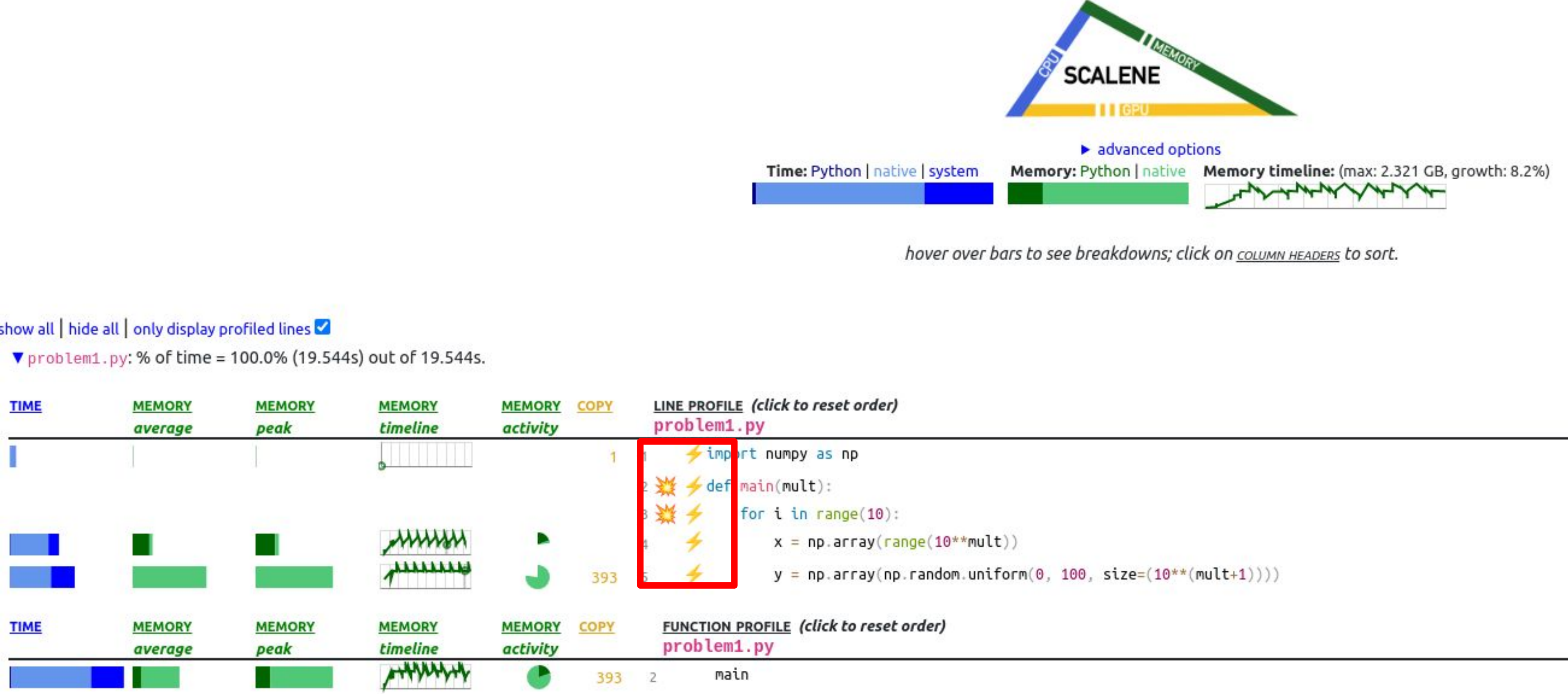
hover over bars to see breakdowns; click on COLUMN HEADERS *to sort.*

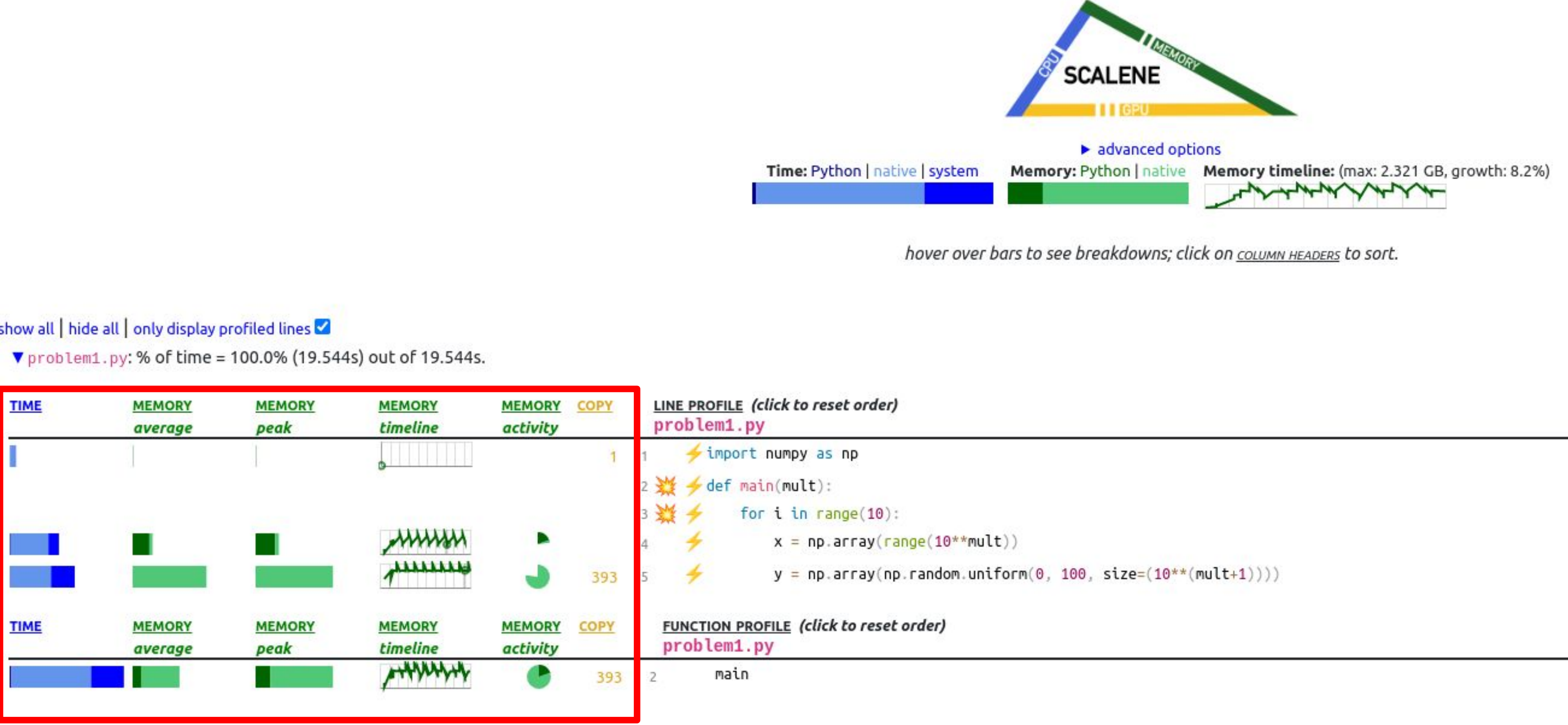
how all | hide all | only display profiled lines ☑

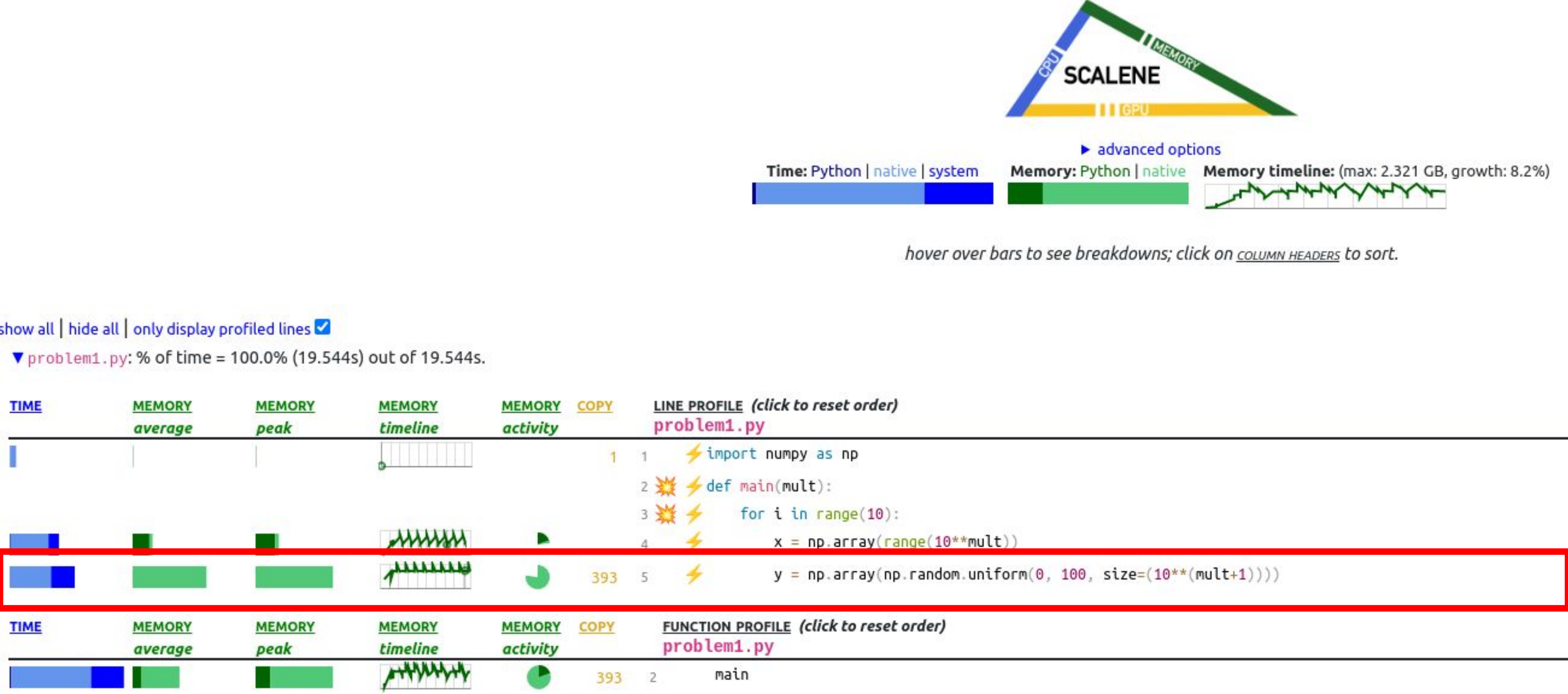
▼ problem1.py: % of time = 100.0% (19.544s) out of 19.544s.

TIME	MEMORY *average*	MEMORY *peak*	MEMORY *timeline*	MEMORY *activity*	COPY		LINE PROFILE *(click to reset order)* problem1.py
					1	1	`import numpy as np`
						2	`def main(mult):`
						3	`    for i in range(10):`
						4	`        x = np.array(range(10**mult))`
					393	5	`        y = np.array(np.random.uniform(0, 100, size=(10**(mult+1))))`

TIME	MEMORY *average*	MEMORY *peak*	MEMORY *timeline*	MEMORY *activity*	COPY		FUNCTION PROFILE *(click to reset order)* problem1.py
					393	2	main


SCALENE
CPU
MEMORY
GPU
advanced options
Time: Python | native | system
Memory: Python | native
Memory timeline: (max: 2.321 GB, growth: 8.2%)
hover over bars to see breakdowns; click on COLUMN HEADERS to sort.
show all | hide all | only display profiled lines
problem1.py: % of time = 100.0% (19.544s) out of 19.544s.
TIME
MEMORY average
MEMORY peak
MEMORY timeline
MEMORY activity
COPY
LINE PROFILE (click to reset order)
problem1.py
1
import numpy as np
def main(mult):
for i in range(10):
x = np.array(range(10**mult))
393
y = np.array(np.random.uniform(0, 100, size=(10**(mult+1))))
FUNCTION PROFILE (click to reset order)
problem1.py
393
2
main


CPU
MEMORY
GPU
SCALENE
advanced options
Time: Python | native | system
Memory: Python | native
Memory timeline: (max: 2.321 GB, growth: 8.2%)
hover over bars to see breakdowns; click on COLUMN HEADERS to sort.
show all | hide all | only display profiled lines
problem1.py: % of time = 100.0% (19.544s) out of 19.544s.
TIME
MEMORY average
MEMORY peak
MEMORY timeline
MEMORY activity
COPY
LINE PROFILE (click to reset order)
problem1.py
1
393
1 import numpy as np
2 def main(mult):
3 for i in range(10):
4 x = np.array(range(10**mult))
5 y = np.array(np.random.uniform(0, 100, size=(10**(mult+1))))
TIME
MEMORY average
MEMORY peak
MEMORY timeline
MEMORY activity
COPY
FUNCTION PROFILE (click to reset order)
problem1.py
393
2 main


SCALENE
CPU
MEMORY
GPU
▶ advanced options
Time: Python | native | system
Memory: Python | native
Memory timeline: (max: 2.321 GB, growth: 8.2%)
hover over bars to see breakdowns; click on COLUMN HEADERS to sort.
show all | hide all | only display profiled lines
▼ problem1.py: % of time = 100.0% (19.544s) out of 19.544s.
TIME
MEMORY average
MEMORY peak
MEMORY timeline
MEMORY activity
COPY
LINE PROFILE (click to reset order)
problem1.py
1 1 import numpy as np
2 def main(mult):
3 for i in range(10):
4 x = np.array(range(10**mult))
393 5 y = np.array(np.random.uniform(0, 100, size=(10**(mult+1))))
TIME
MEMORY average
MEMORY peak
MEMORY timeline
MEMORY activity
COPY
FUNCTION PROFILE (click to reset order)
problem1.py
393 2 main

Problem 1

Switching over to more hands on live demo now

Recording available at the SHARCNET YouTube page

Problem 1: post-mortem

Scalene helped us catch a practically invisible error!

It also provided us with:

- Some nice timings
- Other places to improve
- A breakdown of what is running Natively, and what is running in Python

Problem 2

Our second problem will be that of a **standard prime sieve**

Let's explore some of the code as if it's inside of a notebook on a remote system

This can typically be done via the wiki documentation

The GitHub repository will have a requirements file which will be a snapshot of my working environment on the Graham cluster

Problem 2: post-mortem

- Scalene really does play nice with a wide variety of techniques
- Python is slow until you put some effort into it
- JIT is powerful, though not the focus of this talk

Give it a try on your code base!

Takeaways

- We want to write Python, but be executing native code
- This makes problems potentially really difficult to spot
- Scalene is an excellent tool for helping with this
    - Supports all sorts of programming paradigms and works on the cluster

Thanks very much for you time!

tk11br@sharcnet.ca

Questions?