TRIANGULATING PYTHON PERFORMANCE ISSUES WITH SCALENE

EMERY BERGER, SAM STERN, JUAN ALTMAYER PIZZORNO
UNIVERSITY OF MASSACHUSETTS AMHERST
TRIANGULATING PYTHON PERFORMANCE ISSUES WITH SCALENE

EMERY BERGER, SAM STERN, JUAN ALTMAYER PIZZORNO

UNIVERSITY OF MASSACHUSETTS AMHERST
"Metal" Languages –1980s
"Metal" Languages – 1980s

1949 1957
"Metal" Languages
–1980s

1949  1957  1972
"Metal" Languages – 1980s
Wild Performance Ride (1980s–2010)!
Wild Performance Ride (1980s–2010)!

Transistors (millions)

Clock Speed (MHz)
Wild Performance Ride (1980s–2010)!

Transistors (millions)

Clock Speed (MHz)

Smaller = Faster

Year

Year
Wild Performance Ride (1980s–2010)!

Small = Faster

JUST BUY NEW HARDWARE
AND YOUR CODE RUNS FASTER
"Metal" Languages – 1980s
"Irrational Exuberance" Languages
1990s
"Irrational Exuberance" Languages 1990s
"Irrational Exuberance" Languages

1990s

1991  1993
"Irrational Exuberance" Languages 1990s

1991  1993  1995
"Irrational Exuberance" Languages
1990s
"Irrational Exuberance" Languages
1990s
HOW MANY BYTES IN… (INT, LIST, DICT)?
HOW MANY BYTES IN…(INT, LIST, DICT)?
HOW MANY BYTES IN...(INT, LIST, DICT)?

```c
sizeof(1) → 4
```
HOW MANY BYTES IN…(INT, LIST, DICT)?

\[
\text{sizeof(1)} \rightarrow 4
\]
HOW MANY BYTES IN...(INT, LIST, DICT)?

\[
\begin{align*}
\text{sizeof(1)} & \rightarrow 4 \\
\text{sys.getsizeof(1)} & \rightarrow 28
\end{align*}
\]
HOW MANY BYTES IN… (INT, LIST, DICT)?

```python
>>> sys.getsizeof(1)
28
```

```c++
sizeof(1)  →  4
sizeof(list<int>)  →  24
```
HOW MANY BYTES IN ...(INT, LIST, DICT)?

```
>>> sys.getsizeof(1)
28

sizeof(list<int>)
→ 24

>>> sys.getsizeof([])
56
```
HOW MANY BYTES IN…(INT, LIST, DICT)?

```python
>>> sys.getsizeof(1)
28
>>> sys.getsizeof([])
56
```

C++

```c++
sizeof(1) → 4
sizeof(list<int>) → 24
sizeof(map<int, int>) → 24
```
HOW MANY BYTES IN...(INT, LIST, DICT)?

```python
>>> sys.getsizeof(1)
28
>>> sys.getsizeof([])
56
>>> sys.getsizeof({})
64
```
HOW MANY BYTES IN ...(INT, LIST, DICT)?

\[
\begin{align*}
\text{sizeof(1)} & \rightarrow 4 \\
\text{sizeof(list<int>)} & \rightarrow 24 \\
\text{sizeof(map<int, int>)} & \rightarrow 24 \\
\end{align*}
\]

```python
>>> sys.getsizeof(1)
28
```

```python
>>> sys.getsizeof([])
56
```

```python
>>> sys.getsizeof({})
240
```

(3.6)
HOW MANY BYTES IN ...(INT, LIST, DICT)?
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
Matrix-Multiply Speedup vs. Pure Python

60,000X slowdown!
≈2010: THE RIDE IS OVER
The Ride Is Over

Transistors (millions)

Clock Speed (MHz)

Too Hot!

Transistor counts still increased

(Moore’s Law)

(Dennard Scaling)
```python
import numpy as np

def main():
    for i in range(10):
        x = np.array(range(10**7))

76999 function calls (74718 primitive calls) in 6.307 seconds

Ordered by: cumulative time

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>433/1</td>
<td>0.000</td>
<td>0.000</td>
<td>6.307</td>
<td>6.307</td>
<td>test2-2.py:2(&lt;module&gt;)</td>
</tr>
<tr>
<td>1</td>
<td>0.022</td>
<td>0.022</td>
<td>6.307</td>
<td>6.307</td>
<td>test2-2.py:4(main)</td>
</tr>
<tr>
<td>13</td>
<td>0.155</td>
<td>0.155</td>
<td>6.216</td>
<td>6.216</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>3.191</td>
<td>0.024</td>
<td>3.191</td>
<td>0.024</td>
<td>{built-in method numpy.array}</td>
</tr>
<tr>
<td>6</td>
<td>2.870</td>
<td>0.478</td>
<td>2.870</td>
<td>0.478</td>
<td>{method 'uniform' of 'numpy.random.mtrand.RandomState' objects}</td>
</tr>
<tr>
<td>156/1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.069</td>
<td>0.069</td>
<td>&lt;frozen importlib._bootstrap&gt;:1002(_find_and_load)</td>
</tr>
<tr>
<td>156/1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.069</td>
<td>0.069</td>
<td>&lt;frozen importlib._bootstrap&gt;:967(_find_and_load_unlocked)</td>
</tr>
<tr>
<td>145/1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.069</td>
<td>0.069</td>
<td>&lt;frozen importlib._bootstrap&gt;:659(_load_unlocked)</td>
</tr>
<tr>
<td>112/1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.069</td>
<td>0.069</td>
<td>&lt;frozen importlib._bootstrap_external&gt;:784(exec_module)</td>
</tr>
<tr>
<td>224/1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.069</td>
<td>0.069</td>
<td>&lt;frozen importlib._bootstrap&gt;:220(_call_with_frames_removed)</td>
</tr>
<tr>
<td>179/16</td>
<td>0.000</td>
<td>0.000</td>
<td>0.066</td>
<td>0.066</td>
<td>&lt;frozen importlib._bootstrap&gt;:1033(_handle_fromlist)</td>
</tr>
<tr>
<td>326/9</td>
<td>0.000</td>
<td>0.000</td>
<td>0.066</td>
<td>0.066</td>
<td>{built-in method builtins.<strong>import</strong>}</td>
</tr>
<tr>
<td>112</td>
<td>0.000</td>
<td>0.000</td>
<td>0.024</td>
<td>0.024</td>
<td>&lt;frozen importlib._bootstrap_external&gt;:856(get_code)</td>
</tr>
<tr>
<td>112</td>
<td>0.000</td>
<td>0.000</td>
<td>0.016</td>
<td>0.016</td>
<td>&lt;frozen importlib._bootstrap_external&gt;:976(get_data)</td>
</tr>
<tr>
<td>112</td>
<td>0.014</td>
<td>0.000</td>
<td>0.014</td>
<td>0.014</td>
<td>{method 'read' of '_io.BufferedReader' objects}</td>
</tr>
<tr>
<td>145/142</td>
<td>0.000</td>
<td>0.000</td>
<td>0.011</td>
<td>0.011</td>
<td>&lt;frozen importlib._bootstrap&gt;:558(module_from_spec)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.010</td>
<td>0.010</td>
<td>0.010</td>
<td>multiarray.py:1(&lt;module&gt;)</td>
</tr>
<tr>
<td>320</td>
<td>0.000</td>
<td>0.000</td>
<td>0.010</td>
<td>0.010</td>
<td>overrides.py:187(decorator)</td>
</tr>
<tr>
<td>32/30</td>
<td>0.008</td>
<td>0.000</td>
<td>0.010</td>
<td>0.010</td>
<td>&lt;frozen importlib._bootstrap_external&gt;:1106(create_module)</td>
</tr>
<tr>
<td>32/30</td>
<td>0.008</td>
<td>0.000</td>
<td>0.010</td>
<td>0.010</td>
<td>{built-in method _imp.create_dynamic}</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.009</td>
<td>0.009</td>
<td>overrides.py:1(&lt;module&gt;)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.008</td>
<td>0.008</td>
<td>_pickle.py:1(&lt;module&gt;)</td>
</tr>
<tr>
<td>153</td>
<td>0.000</td>
<td>0.000</td>
<td>0.007</td>
<td>0.007</td>
<td>&lt;frozen importlib._bootstrap&gt;:901(_find_spec)</td>
</tr>
</tbody>
</table>
```

THE "C"

PYTHON
THE "C"

PYTHON

numpy  scikit  tensorflow

THE "C"
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)))
./test2-2.py: % of time = 97.8% out of 30.1s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
<th>LINE PROFILE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.//test2-2.py</td>
</tr>
</tbody>
</table>

2 import numpy as np
4 def main():
  5     for i in range(10):
  6     x = np.array(range(10**7))
  7     y = np.array(np.random.uniform(0, 100, size=(10**8)))
```
./test2-2.py: % of time = 97.8% out of 30.1s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY</th>
<th>MEMORY</th>
<th>MEMORY</th>
<th>COPY</th>
<th>GPU</th>
<th>GPU</th>
<th>LINE PROFILE (click to reset order)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>peak</td>
<td>timeline</td>
<td>activity</td>
<td>(MB/s)</td>
<td>util.</td>
<td>memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17%</td>
<td>83%</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>import</td>
<td>numpy</td>
<td>as np</td>
<td></td>
<td></td>
<td></td>
<td>2 import numpy as np</td>
</tr>
<tr>
<td>4</td>
<td>def</td>
<td>main()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 def main():</td>
</tr>
<tr>
<td>5</td>
<td>for</td>
<td>i in</td>
<td>range(10):</td>
<td></td>
<td></td>
<td></td>
<td>5 for i in range(10):</td>
</tr>
<tr>
<td>6</td>
<td>x = np.array(range(10**7))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>y = np.array(np.random.uniform(0, 100, size=(10**8)))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
AVERAGE & PEAK MEMORY PYTHON NATIVE SYS%

//test2-2.py: % of time = 97.8% out of 30.1s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
<th>LINE PROFILE (click to reset order)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>./test2-2.py</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 import numpy as np</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 def main():</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5 for i in range(10):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6 x = numpy.array(range(10**7))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 y = numpy.array(random.uniform(0, 100, size=(10**8)))</td>
</tr>
</tbody>
</table>
./test2-2.py: % of time = 97.8% out of 30.1s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
<th>LINE PROFILE (click to reset order)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>./test2-2.py</td>
</tr>
</tbody>
</table>

2) import numpy as np
4) def main():
5)    for i in range(10):
6)        x = np.array(range(10**7))
7)        y = np.array(np.random.uniform(0, 100, size=(10**8)))

CPU

PYTHON
NATIVE
SYS%

MEMORY
PYTHON
NATIVE
AVERAGE & PEAK

MEMORY
PYTHON
NATIVE
OVER TIME, % OF MEM ALLOCATED

MEMORY
PYTHON
NATIVE
AVEAGE & PEAK

COPY
GPU
GPU
MEMORY USAGE

MEMORY
PYTHON
NATIVE
OVER TIME, % OF MEM ALLOCATED

COPY
GPU
MEMORY

Time: Python | native | system
Memory: Python | native
Memory timeline: (max: 3135.8MB, growth: 3.1%)

hover over bars to see breakdowns; click on COLUMN HEADERS to sort.
AVERAGE & PEAK OVER TIME, % OF MEM ALLOCATED

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MEMORY USAGE OVER TIME

CPU PYTHON NATIVE
MEMORY USAGE OVER TIME

PYTHON NATIVE AVERAGE & PEAK

CPU PYTHON NATIVE SYS%
MEMORY USAGE

/test2-2.py: % of time = 97.8% out of 30.1s.

```python
2 import numpy as np
4 def main():
5     for i in range(10):
6         x = np.array(range(10**7))
7         y = np.array(np.random.uniform(0, 100, size=(10**8)))
```
./test2.py: % of time = 97.8% out of 30.1s.

```python
2 import numpy as np
4 def main():
5     for i in range(10):
6         x = np.array(range(10**3))
7         y = np.array(np.random.uniform(0, 100, size=(10**3)))
```
```python
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)))
```

Time: Python | native | system
Memory: Python | native | Memory timeline: (max: 3136.8MB, growth: 3.1%

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

MEMORY USAGE OVER TIME
/test2-2.py: % of time = 97.8% out of 30.1s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util%</th>
<th>GPU memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU

PYTHON

NATIVE

SYS%

MEMORY

PYTHON

NATIVE

AVERAGE & PEAK

MEMORY

USAGE

OVER TIME, % OF MEM ALLOCATED

COPY

VOLUME

(MB/s)

GPU

UTIL%, PEAK MEMORY

MEMORY USAGE OVER TIME

LINE PROFILE (click to reset order)

/test2-2.py
```python
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)))

./test2-2.py
```
58% of runtime in native code

```python
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)))
```

Time: Python | native | system
Memory: Python | native | system

Memory timeline: (max: 3135.8MB, growth: 3.1%)

hover over bars to see breakdowns; click on COLUMN HEADERS to sort.
58% of runtime in native code

2GB allocated in native code

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
<th>CPU (%)</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```python
import numpy as np

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

/test2-2.py: % of time = 97.8% out of 30.1s.
58% of runtime in native code

2GB allocated in native code

83% of memory activity "sawtooth" pattern

./test2-2.py: % of time = 97.8% out of 30.1s.

```python
copy

2 import numpy as np
4 def main():
5     for i in range(10):
6         x = np.array(range(10**7))
7         y = np.array(np.random.uniform(0, 100, size=(10**8)))
```
58% of runtime in native code

2GB allocated in native code

83% of memory activity "sawtooth" pattern

250MB/s copying

./test2-2.py: % of time = 97.8% out of 30.1s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
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)))
```
58% of runtime in native code

2GB allocated in native code

83% of memory activity "sawtooth" pattern

250MB/s copying

/test2-2.py: % of time = 97.8% out of 30.1s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
<th>LINE PROFILE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>./test2-2.py</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 import numpy as np</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 def main():</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5 for i in range(10):</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6 x = np.array(range(10**7))</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 y = np.array(np.random.uniform(0, 100, size=(10**8)))</td>
</tr>
</tbody>
</table>

converts to numpy array
58% of runtime in native code

2GB allocated in native code

83% of memory activity "sawtooth" pattern

250MB/s copying

converts to numpy array

already a numpy array!
```python
test2-2.py: % of time = 97.8% out of 30.1s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
</tr>
</thead>
</table>

2  import numpy as np
4  def main():
5     for i in range(10):
6         x = np.arange(10**7)
7     y = np.array([np.random.uniform(0, 100, size=(10**8))])
```
The code snippet for `test2-2.py` and `test2-2-optimized.py` shows the use of NumPy for array manipulation.

### Test2-2.py
```python
import numpy as np

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

### Test2-2-Optimized.py
```python
import numpy as np

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

The diagrams indicate the CPU and memory usage for both files, with `test2-2-optimized.py` showing a more efficient memory usage compared to `test2-2.py`.
### ./test2-2.py: % of time = 97.8% out of 30.1s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### LINE PROFILE

```
2 import numpy as np
4 def main():
5   for i in range(10):
6     x = np.array(range(10**7))
7     y = np.array(np.random.uniform(0, 100, size=(10**8)))
```

### ./test2-2-optimized.py: % of time = 95.2% out of 23.4s.

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### LINE PROFILE

```
2 import numpy as np
4 def main():
5   for i in range(10):
6     x = np.array(range(10**7))
7     y = np.random.uniform(0, 100, size=(10**8))
```
memory_profiler: ~300X slower
(from 5s to 20 minutes!)
NATIVE VS. PYTHON TIME

LOW-OVERHEAD MEMORY PROFILING
SAMPLING-BASED PROFILING
SAMPLING-BASED PROFILING

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>1</td>
</tr>
</tbody>
</table>
SAMPLING-BASED PROFILING

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>1</td>
</tr>
<tr>
<td>bar</td>
<td>1</td>
</tr>
</tbody>
</table>
SAMPLING-BASED PROFILING

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>1</td>
</tr>
<tr>
<td>bar</td>
<td>2</td>
</tr>
</tbody>
</table>
SAMPLING-BASED PROFILING

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>2</td>
</tr>
<tr>
<td>bar</td>
<td>2</td>
</tr>
</tbody>
</table>
DEFERRED SIGNAL DELIVERY

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>1</td>
</tr>
</tbody>
</table>
DEFERRED SIGNAL DELIVERY

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>1</td>
</tr>
</tbody>
</table>
DEFERRED SIGNAL DELIVERY

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>1</td>
</tr>
</tbody>
</table>
DEFERRED SIGNAL DELIVERY

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>2</td>
</tr>
</tbody>
</table>
DEFERRED SIGNAL DELIVERY

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>2</td>
</tr>
</tbody>
</table>
DEFERRED SIGNAL DELIVERY

python™ python™ python™ python™

python™ C++ C++ python™

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>2</td>
</tr>
</tbody>
</table>

(pprofile)
INFERRING EXECUTION TIME

(VIRTUAL TIME)
INFERRING EXECUTION TIME

(VIRTUAL TIME)

\[
\begin{align*}
\text{python} & \quad \text{time} \\
\text{delay} & \\
\text{c} & \quad \text{time} \\
\text{foo} & \quad 2 \\
\text{bar} & \quad 2
\end{align*}
\]
INFERRING EXECUTION TIME

(VIRTUAL TIME)

python_time += interval

c_time += delay - interval

<table>
<thead>
<tr>
<th>function</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>2</td>
</tr>
<tr>
<td>bar</td>
<td>2</td>
</tr>
</tbody>
</table>
memory_profiler:
memory_profiler: ~300x slower
memory_profiler: ~300x slower

tracks every malloc/free
memory_profiler: ~300x slower

tracks every malloc/free

invokes getrusage!
memory_profiler: \(\sim 300x\) slower

tracks every malloc/free
memory_profiler: ~300x slower

tracks every malloc/free
memory_profiler: ~300x slower

tracks every malloc/free
memory_profiler: ~300x slower

tracks every malloc/free

threshold-based sampling
memory_profiler: ~300x slower

tracks every malloc/free

threshold-based sampling
memory_profiler: ~300x slower

tracks every malloc/free

threshold-based sampling
memory_profiler: ~300x slower

tracks every malloc/free

threshold-based sampling
memory_profiler: ~300x slower

tracks every malloc/free

threshold-based sampling
memory_profiler: ~300x slower

tracks every malloc/free
memory_profiler: ~300x slower

- tracks every malloc/free
- only tracks every \( \Delta \geq 1\text{MB} \)
- threshold-based sampling

record malloc info
record malloc info
record free info
record malloc info

~1MB

~1MB
memory_profiler: ~300x slower

threshold-based sampling

only tracks every \( \Delta \geq 1\text{MB} \)

tracks every malloc/free

~1MB

~1MB
memory_profiler: ~300x slower

tracks every malloc/free

only tracks every \( \Delta \geq 1\text{MB} \)

threshold-based sampling

~1MB

record malloc info

record malloc info

record malloc info

record free info

record malloc info

record malloc info
memory_profiler: ~300x slower

tracks every malloc/free

only tracks every \( \Delta \geq 1\text{MB} \)

threshold-based sampling

~1MB
memory_profiler: ~300x slower

tracks every malloc/free

only tracks every \( \Delta \geq 1\text{MB} \)

~1MB

~1MB

threshold-based sampling

only tracks every 
\( \Delta \geq 1\text{MB} \)
memory_profiler: ~300x slower

tracks every malloc/free

only tracks every \( \Delta \geq 1\text{MB} \)

~1MB

~1MB

threshold-based sampling
memory_profiler: ~300x slower

tracks every malloc/free

only tracks every $\Delta \geq 1\text{MB}$
memory_profiler: ~300x slower

tracks every malloc/free

only tracks every \( \Delta \geq 1\text{MB} \)

~1MB

\(~1\text{MB}\)
memory_profiler: \(~300\times\) slower

threshold-based sampling

only tracks every \(\Delta \geq 1\text{MB}\)

\(~1\text{MB}\)

tracks every malloc/free

\(~1\text{MB}\)
tracks every malloc/free

~1MB

Δ ≥ 1MB

only tracks

~1MB

HIGHER ACCURACY + ~no overhead!

memory_profiler: ~300x slower
reports leak volume (MB/s) per line

<table>
<thead>
<tr>
<th>TIME</th>
<th>MEMORY average</th>
<th>MEMORY peak</th>
<th>MEMORY timeline</th>
<th>MEMORY activity</th>
<th>COPY (MB/s)</th>
<th>GPU util.</th>
<th>GPU memory</th>
<th>LINE PROFILE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100%</td>
<td>430</td>
<td>562</td>
<td>562</td>
<td>3 for i in range(1000000):</td>
<td>leaky/test-leaky.py</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

memory: 1596.0MB (@ 3s); possible leak (0.3 MB/s)
% pip install -U scalene

downloads 715k  downloads/month 27k

GitHub.com/plasma-umass/scalene
% pip install -U scalene

GitHub.com/plasma-umass/scalene
% pip install -U scalene

GitHub.com/plasma-umass/scalene
We've started using scalene over at Semantic Scholar (www.semanticscholar.org) as part of our tool suite for operationalizing machine learning models. Recently we found a model of ours was cost prohibitive and put an entire product direction in jeopardy. We generated a set of test data and ran our models with Scalene mounted -- the html output was able to pin point our squeakiest wheels and help us validate our changes were having an impact. The process was iterative, precise and repeatable. In the end, we were able to reduce costs by a staggering 92%.

With these models, there is also always the question of whether things would be more cost effective running inference services on GPUs rather than CPU. Scalene allowed us to quickly ascertain what fraction of our runtime would benefit from the hardware acceleration, and what CPU-bound code we'd need to pare down to achieve our goals.
writing your code in Python
writing your code in Python

profiling your Python code with Scalene
writing your code in Python

profiling your Python code with Scalene

getting Scalene to optimize your code!
Select a profile (.json)

Advanced options
Proposed optimizations
Enter an OpenAI key to enable:

- Optimize runtime performance
- Optimize memory efficiency
- Include GPU optimizations

Click on an explosion (恸) to see proposed optimizations for a region of code, or on a lightning bolt (掣) to propose optimizations for a specific line. Click again to generate a different one.

Note that optimizations are AI-generated and may not be correct.

Time: Python | native | system
Memory: Python | native
Memory timeline: (max: 1.653 GB, growth: 21.0%)

Hover over bars to see breakdowns; click on COLUMN HEAEDERS to sort.
# Proposed optimization:
# Vectorize the code to reduce the number of loops and improve performance.

```python
x = np.arange(10**7)
y = np.random.uniform(0, 100, size=10**8)
```
90x speedup

#58 (comment) presents the following code:

```python
for i in range(n_features):
    for n in range(n_samples):
        subgrad[i] += (- y[n] * X[n][i]) if y[n] * (np.dot(X[n], w) + b) < 1 else 0
        subgrad[i] += self.lambda1 * (-1 if w[i] < 0 else 1) + 2 * self.lambda2 * w[i]
```

Scalene proposes the following optimization:

```python
# Vectorized operations to replace for loops
subgrad[1:-1] = np.sum(-y[:, None] * X * (y * (X.dot(w) + b) < 1)[:, None], axis=0)
subgrad[1:-1] += self.lambda1 * np.sign(w) + 2 * self.lambda2 * w
subgrad[-1] = np.sum(-y * (y * (X.dot(w) + b) < 1))
```

Scalene's proposed optimization accelerates the original code by at least 90x (89 seconds to 1 second, when running 500 iterations), and takes full advantage of multiple cores.
% pip install -U scalene

GitHub.com/plasma-umass/scalene
% pip install -U scalene

GitHub.com/plasma-umass/scalene