Tuning Python Applications Can Dramatically Increase Performance

Vasilij Litvinov

Software Engineer, Intel

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Agenda

Why do we need Python optimization?

How one finds the code to tune?

Overview of existing tools

An example

Intel® VTune Amplifier capabilities and comparison

Q & A

Why do we need Python optimization?

- Python is used to power wide range of software, including those where application performance matters
- Some Python code may not scale well, but you won't know it unless you give it enough workload to chew on
- Sometimes you are just not happy with the speed of your code

All in all, there are times when you want to make your code run faster, be more responsive, *(insert your favorite buzzword here)*.

So, you need to **optimize** (or tune) your code.

How one finds the code to tune – measuring vs guessing

- **Hard stare** = Often wrong

- **Logging** = Analysis is tedious

- **Profile** = Accurate, Easy

Not All Profilers Are Equal

There are different profiling techniques, for example:

- **Event-based**
  - Example: built-in Python cProfile profiler

- **Instrumentation-based**
  - Usually requires modifying the target application (source code, compiler support, etc.)
  - Example: line_profiler

- **Statistical**
  - Accurate enough & less intrusive
  - Example: vmstat, statprof

Most Profilers – High Overhead, No Line Info

Tool	Description	Platforms	Profile level	Avg. overhead
cProfile (built-in)	• Text interactive mode: “pstats” (built-in) • GUI viewer: RunSnakeRun • Open Source	Any	Function	1.3x-5x
Python Tools	• Visual Studio (2010+) • Open Source	Windows	Function	~2x
PyCharm	• Not free • cProfile/yappi based	Any	Function	1.3x-5x (same as cProfile)
line_profiler	• Pure Python • Open Source • Text-only viewer	Any	Line	Up to 10x or more

Example performance hogs

Task	Slow way	Faster way

Concatenate a list	`s = ''`
	`for ch in some_lst:`
	`    s += ch`
		`s = ''.join(some_lst)`

reason		

Remove some value from a list	`while some_value in lst:`
	`    lst.remove(some_value)`
		`while True:`
		`    try:`
		`        lst.remove(some_value)`
		`    except ValueError:`
		`        break`

reason		

Python example to profile: **demo.py**

```
class Encoder:
    CHAR_MAP = {'a': 'b', 'b': 'c'}
    def __init__(self, input):
        self.input = input

    def process_slow(self):
        result = ''
        for ch in self.input:
            result += self.CHAR_MAP.get(ch, ch)
        return result

    def process_fast(self):
        result = []
        for ch in self.input:
            result.append(self.CHAR_MAP.get(ch, ch))
        return ''.join(result)
```

Python sample to profile: **run.py**

```
import demo
import time

def slow_encode(input):
    return demo.Encoder(input).process_slow()

def fast_encode(input):
    return demo.Encoder(input).process_fast()

if __name__ == '__main__':
    input = 'a' * 10000000 # 10 millions of 'a'
    start = time.time()
    s1 = slow_encode(input)
    slow_stop = time.time()
    print 'slow: %.2f sec' % (slow_stop - start)
    s2 = fast_encode(input)
    print 'fast: %.2f sec' % (time.time() - slow_stop)
```

slow: 9.15 sec = 1.00x
fast: 3.16 sec = 1.00x

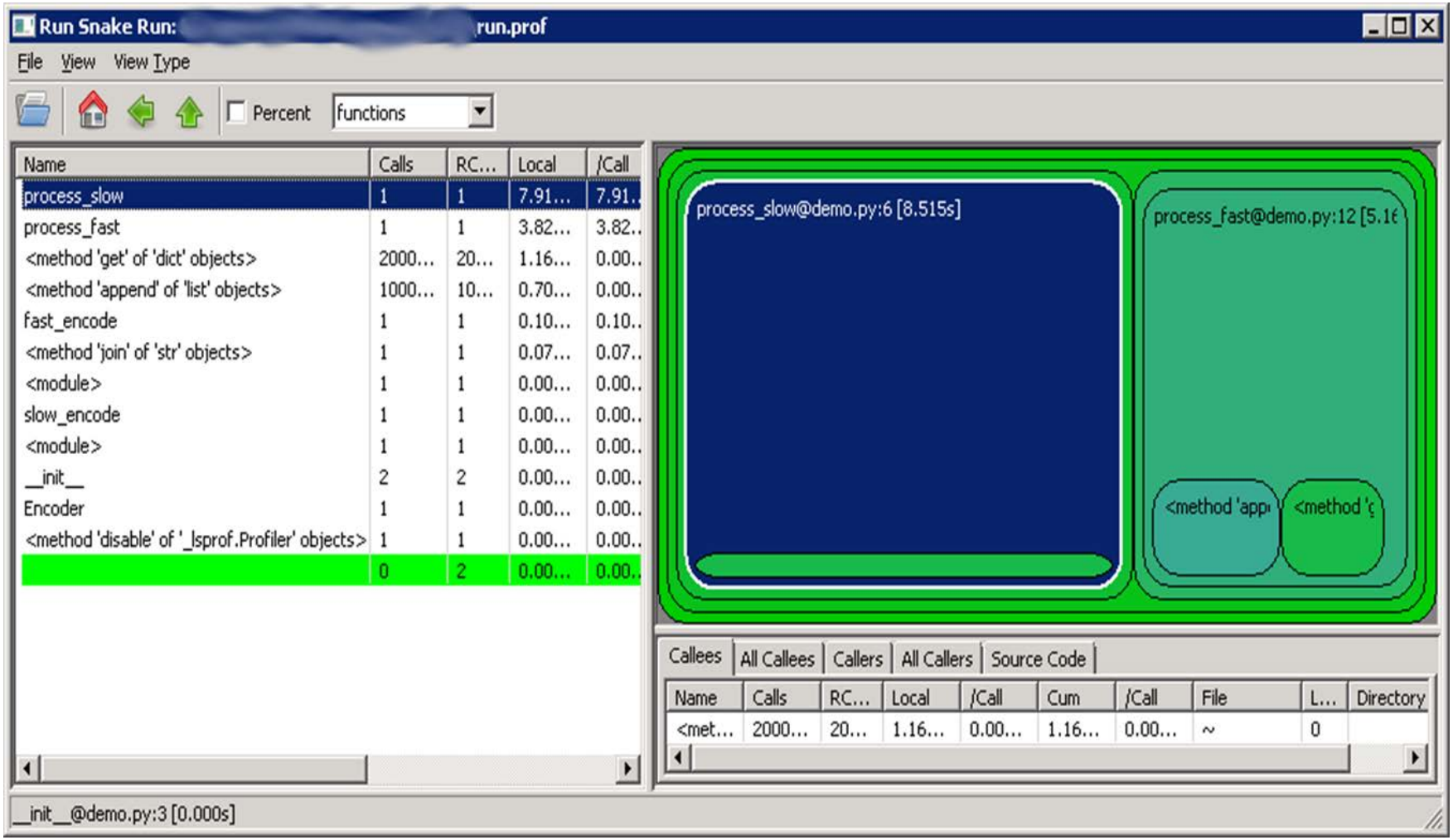
No profiling overhead - a baseline for tools' overhead comparison

cProfile + pstats UI example

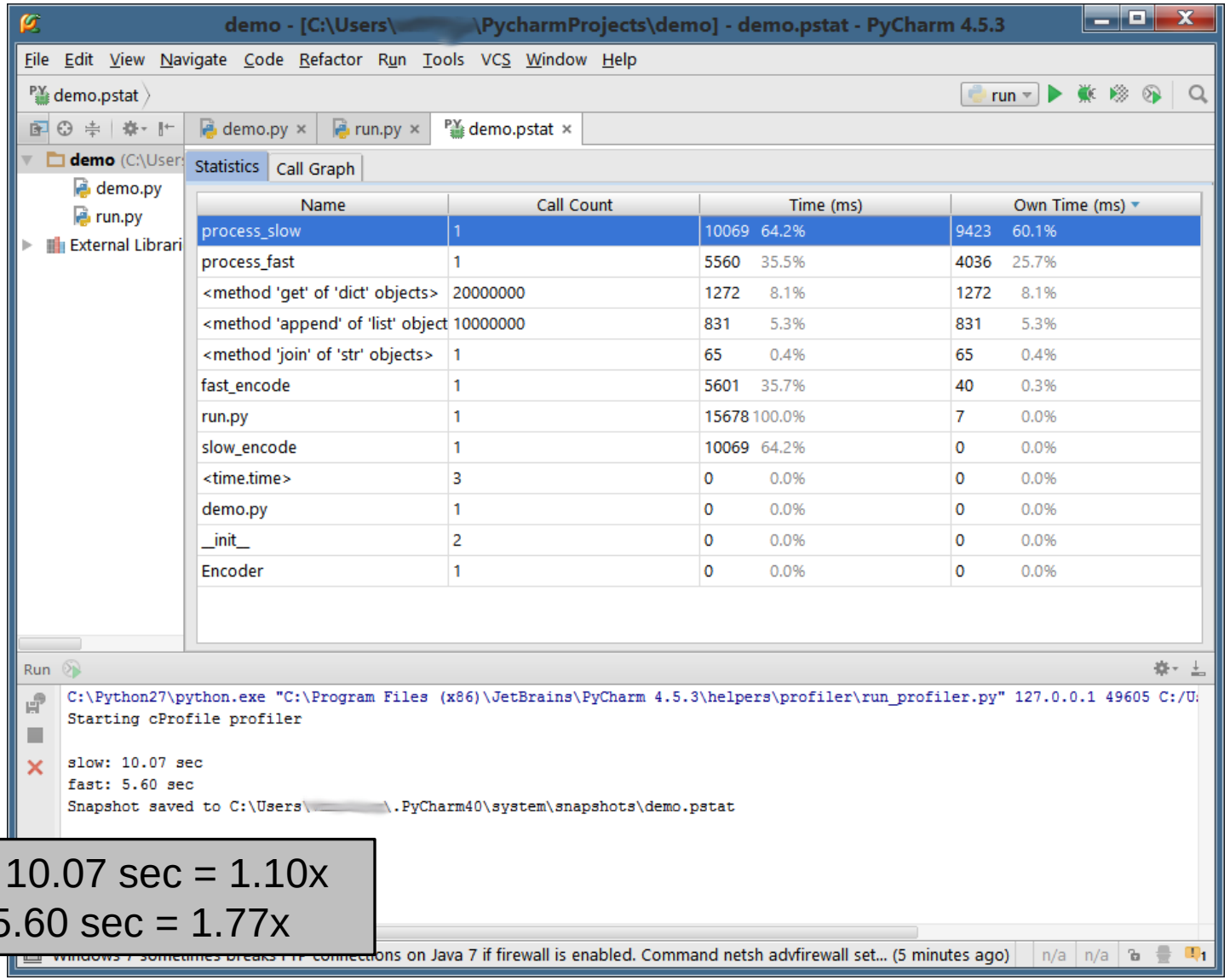
```
> python -m cProfile -o run.prof run.py
> python -m pstats run.prof

run.prof% sort time
run.prof% stats
Tue Jun 30 18:43:53 2015    run.prof

         30000014 function calls in 15.617 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    9.597    9.597   10.268   10.268 demo.py:6(process_slow)
        1    3.850    3.850    5.302    5.302 demo.py:12(process_fast)
 20000000    1.267    0.000    1.267    0.000 {method 'get' of 'dict' objects}
 10000000    0.790    0.000    0.790    0.000 {method 'append' of 'list' objects}
        1    0.066    0.066    0.066    0.066 {method 'join' of 'str' objects}
        1    0.038    0.038    5.340    5.340 run.py:7(fast_encode)
        1    0.009    0.009   15.617   15.617 run.py:1(<module>)
        1    0.000    0.000   10.268   10.268 run.py:4(slow_encode)
```

slow: 10.27 sec = 1.12x
fast: 5.34 sec = 1.69x

cProfile + RunSnakeRun



cProfile in PyCharm



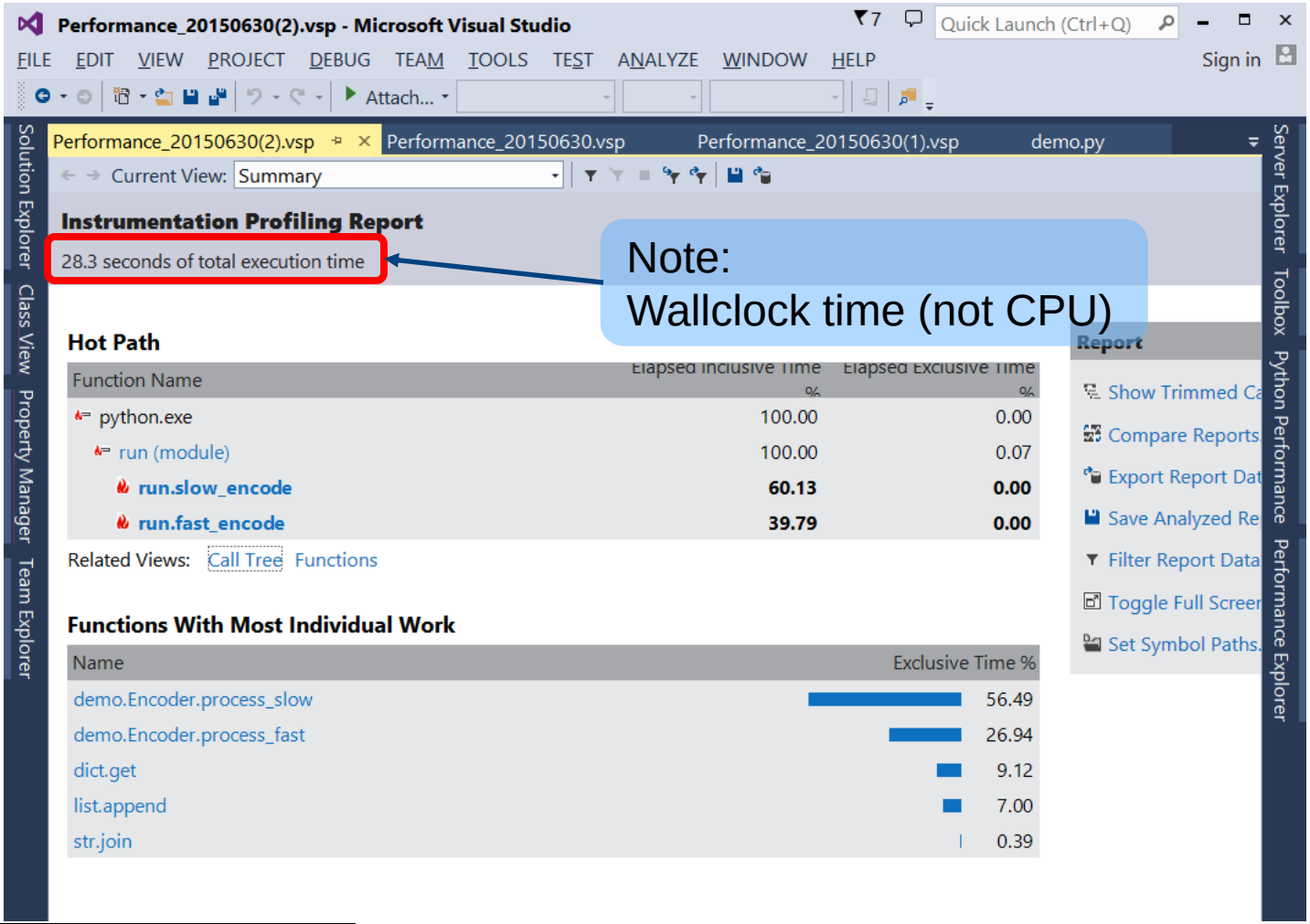
slow: 10.07 sec = 1.10x
fast: 5.60 sec = 1.77x

line_profiler results

```
Total time: 18.095 s
File: demo_lp.py
Function: process_slow at line 6
Line #      Hits     Time  Per Hit   % Time  Line Contents
==============================================================
     6                                           @profile
     7                                           def process_slow(self):
     8         1       14     14.0      0.0      result = ''
     9  10000001 10260548      1.0     23.3      for ch in self.input:
    10  10000000 33814644      3.4     76.7          result += self.CHAR_MAP.get(...
    11         1        4      4.0      0.0      return result

Total time: 16.8512 s
File: demo_lp.py
Function: process_fast at line 13
Line #      Hits     Time  Per Hit   % Time  Line Contents
==============================================================
    13                                           @profile
    14                                           def process_fast(self):
    15         1        7      7.0      0.0      result = []
    16  10000001 13684785      1.4     33.3      for ch in self.input:
    17  10000000 27048146      2.7     65.9          result.append(self.CHAR_MAP.get(...
    18         1   312611 312611.0      0.8      return ''.join(result)
```

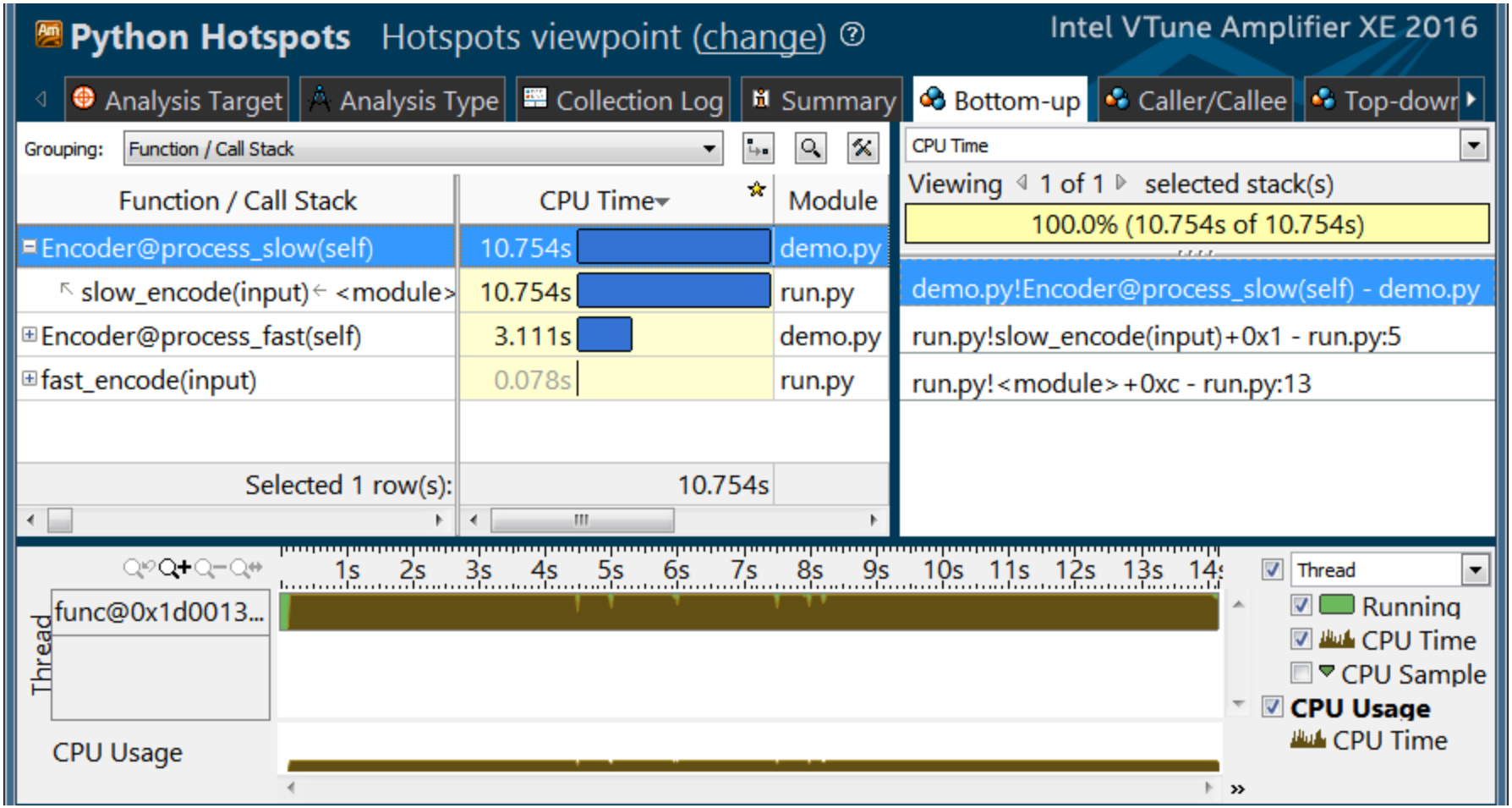
slow: 24.32 sec = 2.66x
fast: 25.37 sec = 8.03x

Python Tools GUI example



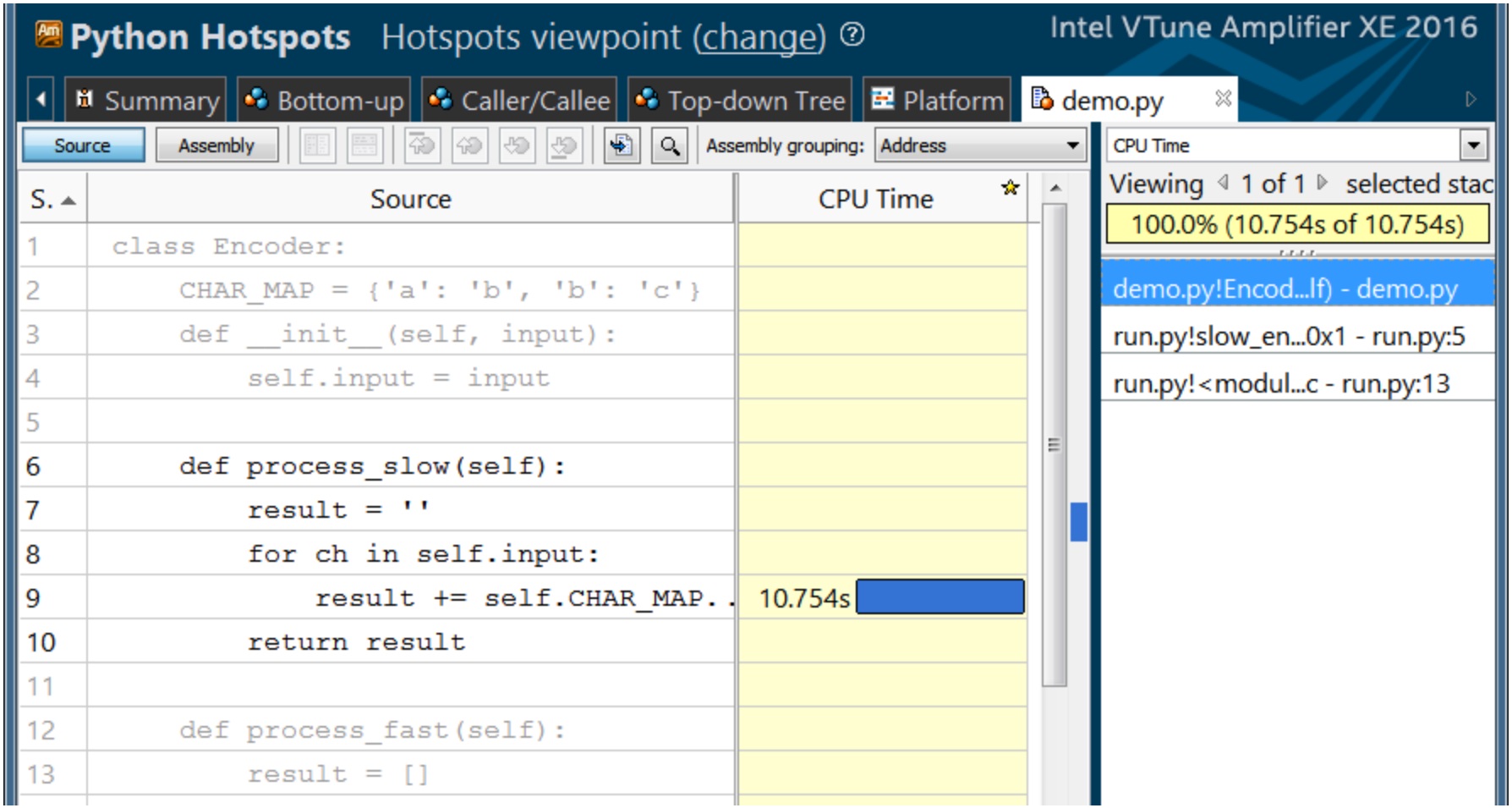
slow: 17.40 sec = 1.90x
fast: 12.08 sec = 3.82x

Intel® VTune Amplifier example



slow: 10.85 sec = 1.19x
fast: 3.30 sec = 1.05x

Intel® VTune Amplifier – source view



Intel® VTune Amplifier: Accurate & Easy

Line-level profiling details:

- Uses sampling profiling technique
- Average overhead ~1.1x-1.6x (on certain benchmarks)

Cross-platform:

- Windows and Linux
- Python 32- and 64-bit; 2.7.x, 3.4.x, 3.5.0 versions

Rich Graphical UI

Supported workflows:

- Start application, wait for it to finish
- Attach to application, profile for a bit, detach

Low Overhead *and* Line-Level Info

Tool	Description	Platforms	Profile level	Avg. overhead
Intel® VTune Amplifier	• Rich GUI viewer	Windows Linux	Line	~1.1-1.6x
cProfile (built-in)	• Text interactive mode: “pstats” (built-in) • GUI viewer: RunSnakeRun (Open Source) • PyCharm	Any	Function	1.3x-5x
Python Tools	• Visual Studio (2010+) • Open Source	Windows	Function	~2x
line_profiler	• Pure Python • Open Source • Text-only viewer	Any	Line	Up to 10x or more

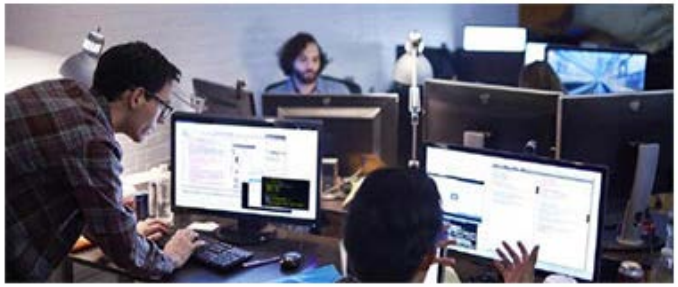
We've Had Success Tuning Our Python Code

- One widely-used web page in our internally set up Buildbot service: 3x speed up (from 90 seconds to 28)
- Report generator – from 350 sec to <2 sec for 1MB log file
  - Distilled version was the base for demo.py
- Internal SCons-based build system: several places sped up 2x or more
  - Loading all configs from scratch tuned from 6 minutes to 3 minutes

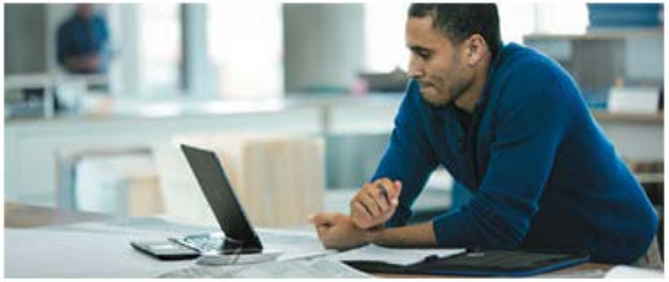
Sign Up with Us to Give the Profiler a Try & Check out Intel® Software Development Tools

- Technical Preview & Beta Participation – email us at scripting@intel.com
  - We're also working on Intel-accelerated Python (e.g. NumPy/SciPy, etc.), which is currently in Tech Preview. Sign up!
- Check out Intel Developer Zone – software.intel.com
  - Check out Intel® Software Development tools
  - Qualify for Free Intel® Software Development tools
- Catch me on the conference if you have some questions, feature suggestions, etc.

Free Intel® Software Development Tools



Academic Researcher ›
Intel Performance Libraries for academic research



Student ›
For current students at degree-granting institutions.

Educator ›
For use in teaching curriculum.

Open Source Contributor ›
For developers actively contributing to open source projects.

Visit us at https://software.intel.com/en-us/qualify-for-free-software

Q & A

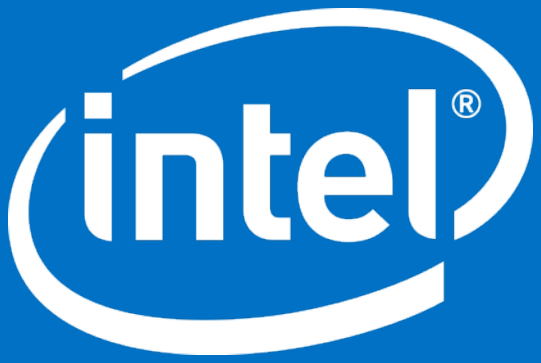
… and again:

- For Tech Preview and Beta, drop us an email at scripting@intel.com
- Check out free Intel® software – just google for “free intel tools” to see if you’re qualified

Performance Starts Here!

You are a:	Products Available++	Support Model	Price
Commercial Developer or Academic Researcher	Intel® Parallel Studio XE (Compilers, Performance Libraries & Analyzers)	Intel® Premier Support	$699** - $2949**
Academic Researcher+	**Intel® Performance Libraries** Intel® Math Kernel Library Intel® MPI Library Intel® Threading Building Blocks Intel® Integrated Performance Primitives	Forum only support	Free!
Student+	Intel® Parallel Studio XE Cluster Edition		
Educator+			
Open Source Contributor+	Intel® Parallel Studio XE Professional Edition		

+Subject to qualification ++OS Support varies by product **Single Seat Pricing


intel®