A HANDS-ON APPROACH TO TUNING PYTHON APPLICATIONS FOR PERFORMANCE

David Liu, Python Technical Consultant Engineer (Intel)
Dr. Javier Conejero, Barcelona Supercomputing Center

Intel Software

OVERVIEW

- Introduction & Tools of the Trade for optimizing Python performance
- Native Performance libraries
- Performance profilers
- Parallelism tools and other accelerators
- Hands-on activity: Optimizing *Black Scholes* algorithm
- Hands-on activity: *Collaborative Filtering* example
- Real world Application example: *PyCOMPSs* from *Barcelona Supercomputing Center*
- Summary

Intel Software 2

PYTHON PERFORMANCE INTRODUCTION

- How does one obtain addition performance on one's Python code?
- What tools are available to diagnose these issues?
- What types of issues are we looking for?
- What types of fixes are available?

PYTHON PERFORMANCE INTRODUCTION (CON'T)

- How does one obtain addition performance on one's Python code?
  - Through better usage of correct data structures for a given problem
  - By leveraging the base language's strengths to full advantage
  - By refactoring one's code where inefficiencies are present
  - By moving parts of code to a more native performance library
  - By using specialized tools that get closer to C or JIT the code
  - By leveraging specialized frameworks that are made for accelerated tasks

PYTHON PERFORMANCE INTRODUCTION (CON'T)

- What tools are available to diagnose these issues?
  - Code profilers
    - Code, Memory, Vectorization
      - cProfile, Perf, line_profiler, Intel® VTune ™
      - Memory_profiler, Intel Vtune ™
      - Intel® Advisor, Intel® Inspector
  - Analyzers
    - For MPI and similar messaging protocols
      - Intel® Trace Analyzer and Collector
  - System profilers
    - Full system, OS-level
      - Linux: sysprof

PYTHON PERFORMANCE INTRODUCTION (CON'T)

- What types of issues are we looking for?
  - Improper loop structure
  - Penalty for misuse of a data structure (dict when it should be a list, list when it should be a tuple, etc.)
  - Syntax and coding mistakes
  - Python language bottlenecks
  - Vectorization
  - Tasks ill-fitted for Python that should be in translated to C++

PYTHON PERFORMANCE INTRODUCTION (CON'T)

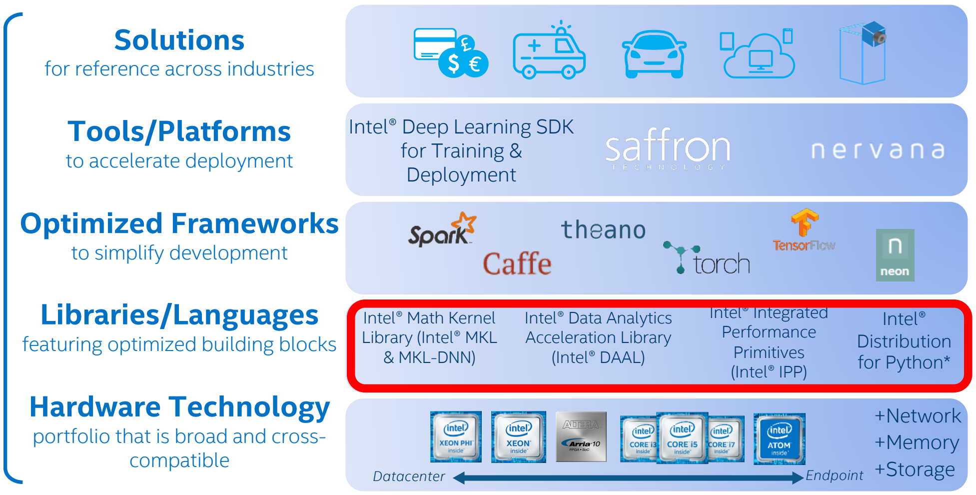
- What types of fixes are available?
  - Syntax and Code fixes at the Python level
  - Syntax and Code fixes at the C++ level
  - Migration of code to the C++ level
  - Refactoring with specialized frameworks
  - Syntax and Code fixes at the Messaging protocol level
  - Refactoring to utilize a distributed framework

intel Software

NATIVE PERFORMANCE LIBRARIES

NATIVE PERFORMANCE LIBRARIES OFFERED BY INTEL

intel strategy provides the foundation for success using AI



NATIVE PERFORMANCE LIBRARIES (CON'T)

- Languages:
  - Intel® Distribution for Python*
- Other performance libraries and tools
  - Cython*
  - Numba*
  - Numexpr*
  - NumPy*

NATIVE PERFORMANCE LIBRARIES (CON'T)

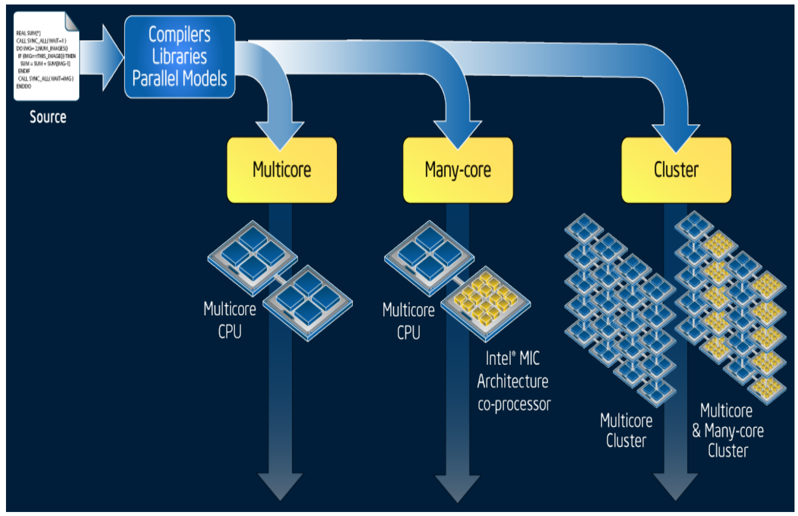
- Numerical and Performance Libraries:
  - **Intel® Math Kernel Library** (Intel® MKL & MKL-DNN)
  - Intel® Integrated Performance Primitives (Intel® IPP)
  - **Intel® Data Analytics Acceleration Library** (Intel® DAAL)
  - **Intel® C++ Compiler**
  - Intel® Threading Building Blocks
  - Intel® MPI Library

NATIVE PERFORMANCE LIBRARIES (CON'T)

- Native Libraries help utilize functions with best *vectorization* available for given hardware
- If one's code or parts of the package are in C++, usage of an *Intel® MKL* variant can provide multiplication factors of performance over the stock OpenBLAS implementation
- Placement of certain algorithms in one's code for data analysis can be refactored to be called with *Intel® DAAL*
- Hardware accelerated MPI with *Intel® MPI*
- Use the Intel® Distribution for Python* as a starting point

FROM SINGLE CORE, TO MULTICORE, TO MANY CORE

- Purpose of libraries is to help scaling of code over various types of hardware
- These are some of the ways we've accelerated NumPy*/SciPy* /Scikit-learn*



INTEL® MATH KERNEL LIBRARY (MKL)

- Features highly optimized, threaded, and vectorized math functions that maximize performance on each processor family
- Utilizes industry-standard C and Fortran APIs for compatibility with popular BLAS, LAPACK, and FFTW functions—no code changes required
- Dispatches optimized code for each processor automatically without the need to branch code
- One of the main performance libraries when making numerical optimizations in one's code (mostly at the C/C++ level)
- Is used directly in the optimized NumPy*/SciPy* for *The Intel® Distribution for Python**

INTEL® DATA ANALYTICS ACCELERATION LIBRARY (INTEL® DAAL)

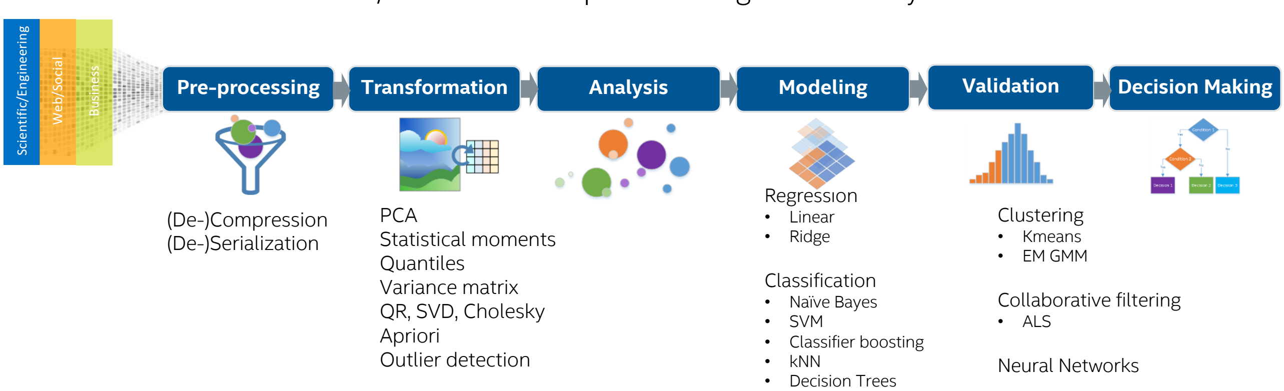
- Features highly tuned functions for deep learning, classical machine learning, and data analytics performance across spectrum of Intel® architecture devices
- Optimizes data ingestion together with algorithmic computation for highest analytics throughput
- Includes Python* (PyDAAL), C++, and Java* APIs and connectors to popular data sources including Spark* and Hadoop*

Software | 15

INTEL® DAAL: HETEROGENEOUS ANALYTICS

Available also in open source:
https://software.intel.com/en-us/articles/opendaal

- Targets both data centers (Intel® Xeon® and Intel® Xeon Phi™) and edge-devices (Intel® Atom™)
- Perform analysis close to data source (sensor/client/server) to optimize response latency, decrease network bandwidth utilization, and maximize security
- Offload data to server/cluster for complex and large-scale analytics



Software | 16

INTEL® DISTRIBUTION FOR PYTHON* 2017

Advancing Python performance closer to native speeds

Easy, out-of-the-box access to high performance Python

- Prebuilt & optimized for numerical computing, HPC, data analytics
- Drop in replacement for your existing Python. No code changes required
- Jupyter* Notebooks, Matplotlib* included
- Compatible with and powered by Anaconda, supports conda and pip

High performance with multiple optimization techniques

- Accelerated NumPy*/SciPy*/Scikit-Learn* with Intel® MKL and Intel® DAAL
- Data analytics with Scikit-learn*, pyDAAL, Caffe*, Theano*
- Numba* and Cython* included for tuning hotspots to scale
- Comes with MPI4Py, works with Dask* and PySpark*

Faster access to latest optimizations for Intel architecture

- Distribution and individual optimized packages available through conda and Anaconda Cloud: anaconda.org/intel
- Optimizations upstreamed back to main Python trunk

intel Software | 17

INSTALLING INTEL® DISTRIBUTION FOR PYTHON* 2017

Stand-alone installer and anaconda.org/intel

Download full installer from
https://software.intel.com/en-us/intel-distribution-for-python

OR

```
> conda config --add channels intel
> conda install intelpython3_core
> conda install intelpython3_full
```

```
docker pull intelpython/intelpython3_full
```

- Linux*
- Windows*
- macOS*

Apt/Yum also available

intel Software | 18

Software

PERFORMANCE PROFILERS

PYTHON PROFILERS

- Profiling one's code is the initial step of investigation for performance tuning
- Many options exist to get large and small granularity insights to one's code
- All profilers have certain characteristics that one need to take into account—using the one that best suits the nature of one's workflow is best
- Insights from profiling lead to direction of optimizations to follow, or possible refactoring path

SHORT OVERVIEW OF PYTHON PROFILERS

Tool	Description	Platforms	Profile level	Avg. overhead *
Intel® VTune™ Amplifier	• Rich GUI viewer • Mixed C/C++/Python code	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 (package)	• Pure Python • Open Source • Text-only viewer	Any	Line	Up to 10x or more
VMProf	• Mixed C++/Python mode • CPython and PyPy • Open Source	Linux, limited Windows (32-bit)	Line	N/A

* Measured against Grand Unified Python Benchmark

Machine specs: HP EliteBook 850 G1; Intel® Core™ i5-4300U @1.90 Ghz (4 cores with HT on) CPU; 16 GB RAM; Windows 8.1 x86_64

CPROFILE AND LINE_PROFILIER

- **CProfile** is C extension variant of *profile* (all Python), has decent overhead for usage
- **Line_profiler** has a much deeper granularity at much higher price
- Easy to instantiate from REPL and Jupyter Notebooks
- Function level vs line-level will depend on what type of Python code is being profiled—single function? Full Program?
- From top level, even simple %timeit or *timeit* might be good enough
- Continuum's *accelerate* module has a bokeh visualization of cProfile if needed

INTEL® VTUNE™ AMPLIFIER

- Profile one's source code to check for hotspots, measure utilization
- Determine optimal vectorization for Intel® processors (C/C++)
- Take advantage of non-uniform memory architectures and cache (C/C++)
- Helps one's code translate from multi-core to many-core systems, such as Xeon Phi™
- Determine IO and CPU-bound behaviors
- Useful even if one's code is non-numerical (such as Django, Buildbot, etc.)

MIXED C/PYTHON EXAMPLE TO PROFILE: CORE.PYX (CYTHON-BASED)

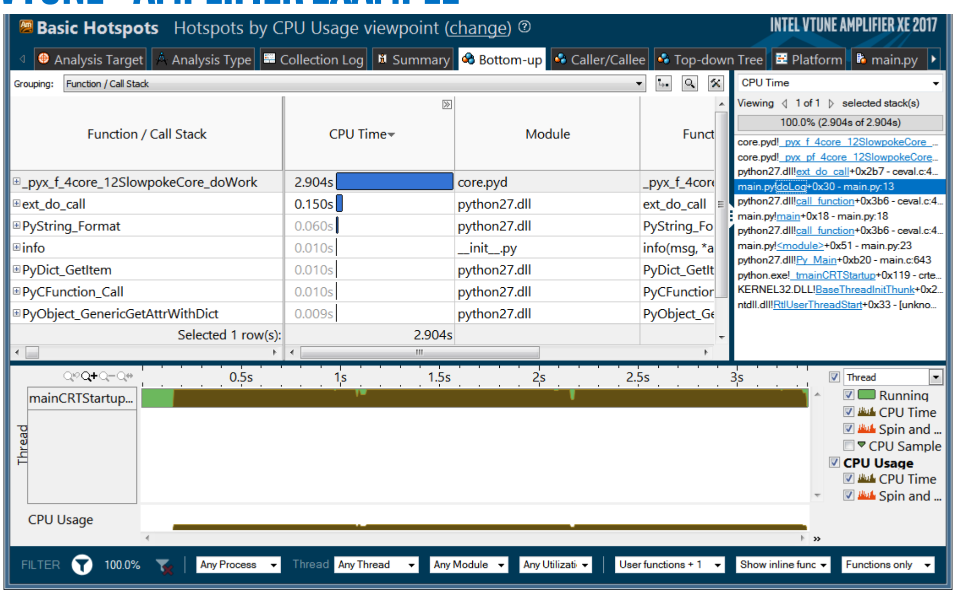
```
import math
cdef class SlowpokeCore:
    cdef public object N
    def __init__(self, N):
        self.N = N

    cdef double doWork(self, int N) except *:
        cdef int i, j, k
        cdef double res
        res = 0
        for j in range(N):
            k = 0
            for i in range(N):
                k += 1
            res += k
        return math.log(res)

    def __str__(self):
        return 'SlowpokeCore: %f' % self.doWork(self.N)
```

© 2017 Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.
intel Software

MIXED C/PYTHON EXAMPLE TO PROFILE: MAIN.PY

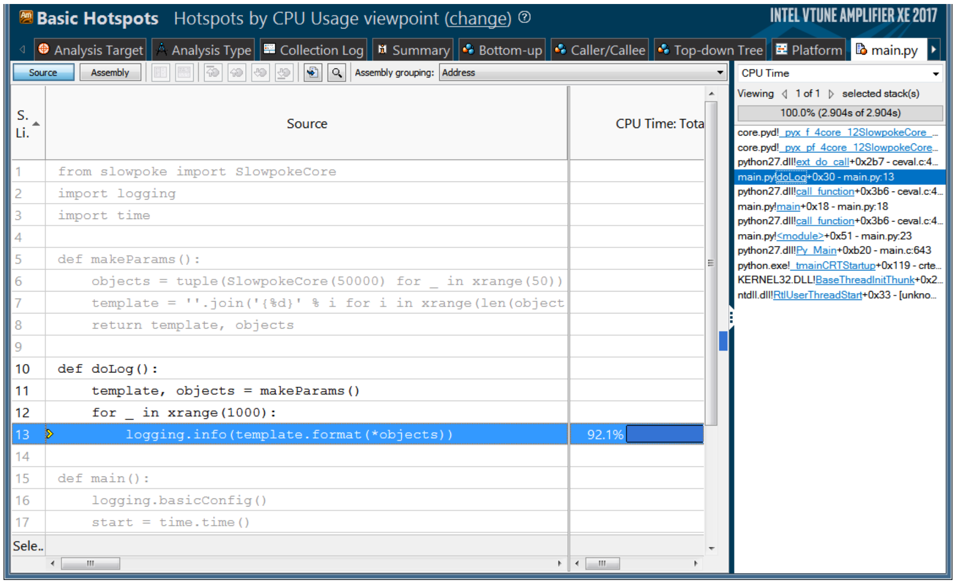
```python
from slowpoke import SlowpokeCore
import logging
import time

def makeParams():
    objects = tuple(SlowpokeCore(50000) for _ in xrange(50))
    template = ''.join('{%d}' % i for i in xrange(len(objects)))
    return template, objects

def calc_pi():
   # removed for readability; pure-Python function was here

def doLog():
    template, objects = makeParams()
    for _ in xrange(1000):
        calc_pi()
        logging.info(template.format(*objects))

def main():
    logging.basicConfig()
    start = time.time()
    doLog()
    stop = time.time()
    print('run took: %.3f' % (stop - start))

if __name__ == '__main__':
    main()
```


INTEL® VTUNE™ AMPLIFIER EXAMPLE



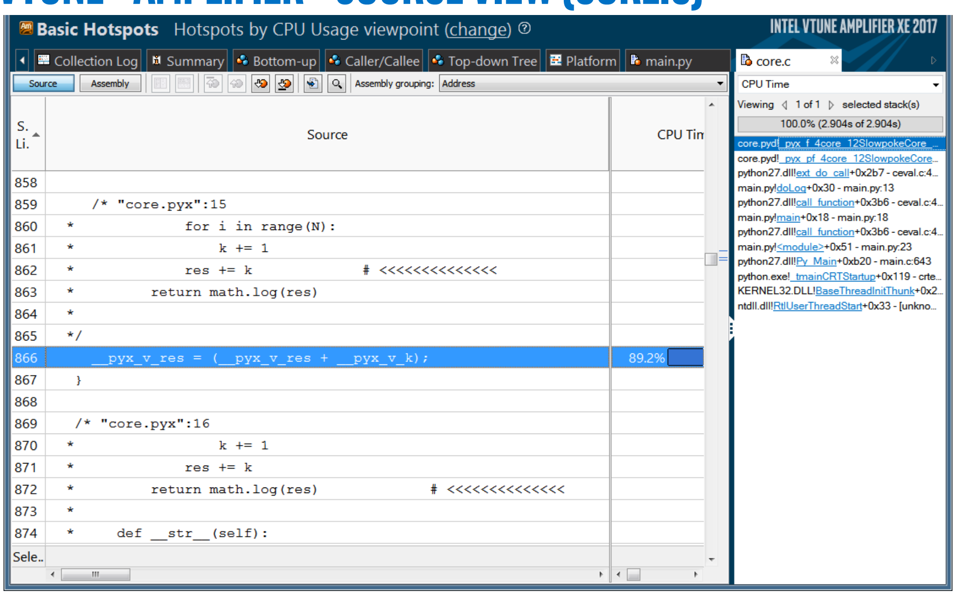
Machine specs: HP EliteBook 850 G1; Intel® Core™ i5-4300U @ 1.90 GHz (4 cores with HT on) CPU; 16 GB RAM; Windows 8.1 x86_64

INTEL® VTUNE™ AMPLIFIER – SOURCE VIEW (MAIN.PY)



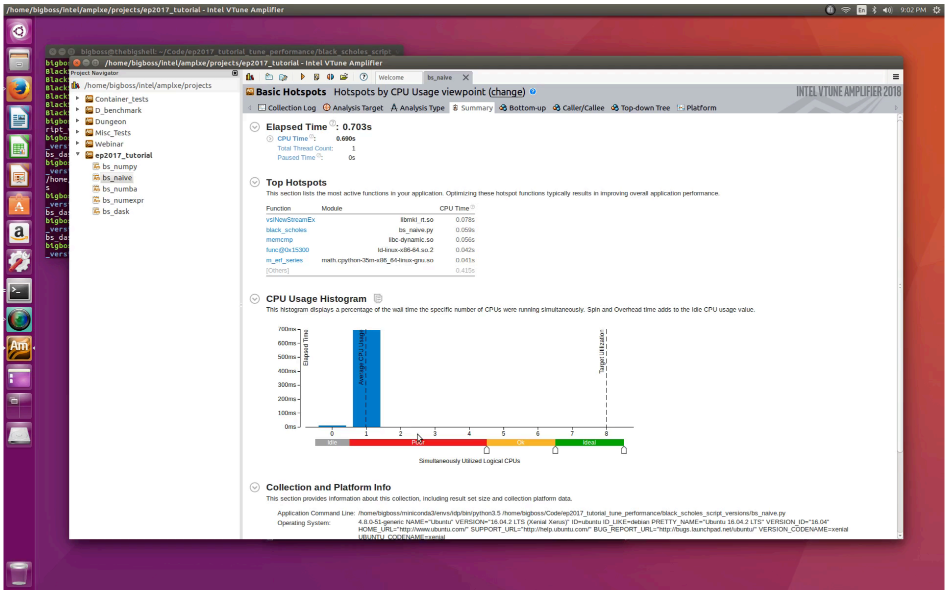
Machine specs: HP EliteBook 850 G1; Intel® Core™ i5-4300U @1.90GHz (4 cores with HT on) CPU; 16 GB RAM; Windows 8.1 x86_64
intel Software

INTEL® VTUNE™ AMPLIFIER – SOURCE VIEW (CORE.C)



Machine specs: HP EliteBook 850 G1; Intel® Core™ i5-4300U @1.90GHz (4 cores with HT on) CPU; 16 GB RAM; Windows 8.1 x86_64
intel Software

VTUNE EXAMPLE



intel Software | 29

INTEL® VTUNE™ AMPLIFIER DETAILS

- Line-level profiling details:
  - Uses sampling profiling technique
  - Average overhead ~1.1x-1.6x (on certain benchmarks *)
- Cross-platform:
  - Windows and Linux (Viewer-only on OSX)
  - Python 32- and 64-bit; 2.7.x, 3.5.X versions (3.6 with 2018 Beta)

* Measured against Grand Unified Python Benchmark

Machine specs: HP EliteBook 850 G1; Intel® Core™ i5-4300U @1.90 Ghz (4 cores with HT on) CPU; 16 GB RAM; Windows 8.1 x86_64

intel Software | 30

PROFILER SUMMARY

- Profilers should be the first step when after a visual inspection does not net performance advantages
- Without Code Profilers, one is pretty much lost without the insight provided by them, especially with the complexity of Python
- Each of the open source profilers have different aspects they are good at (or that they can see), so use accordingly
- Tools such as VTune™ provide source, function, and hardware level information if the open source profilers aren't enough
- Test often, and if in doubt profile your code!

Software

PARALLELISM AND OTHER ACCELERATORS

MANY TYPES OF PARALLELISM

- Parallelism is the best way to achieve performance gains in Python
- Examples:
  - *Message passing*
    - MPI4Py*, Dask*
  - *General parallelism*
    - multiprocessing, Dask*
  - *Multi-format parallelism*
    - Cython*, Numba*
    - TBB, OpenMP are backends/runtimes
    - Numexpr*, NumPy*, et al.
- *At lower levels:* OpenMP, TBB, and MKL, DAAL calls

DISTRIBUTED COMPUTING LANDSCAPE

mpi4py | pySpark | Dask/distributed | . . .

- New distributed computing technologies appear almost every year
- These technologies help Python achieve task-based parallelism and mitigate the issues that many people have with Python

TWO DIFFERENT FLAVORS OF DISTRIBUTED: DASK AND MPI4PY

- MPI4PY*
  - Access to the MPI Library at the Python level
  - Accelerated with Intel® MPI Library
  - Best for composing things that have complex relationships
- Dask*
  - Framework that uses distributed futures to construct tasks graphs and execute via a scheduler
  - Specialized for computational workloads (numerical Python parallelism), and comes with a lot of built-in functionality

MPI4PY

- Allows one to utilize the Message Passing Interface (MPI) with the Python language
- Designed for the parallel computing world
- Can handle very complex relationships that don't necessarily fit "templates" of other distributed task frameworks

```
from mpi4py import MPI
import numpy

def matvec(comm, A, x):
    m = A.shape[0] # local rows
    p = comm.Get_size()
    xg = numpy.zeros(m*p, dtype='d')
    comm.Allgather([x,  MPI.DOUBLE],
                   [xg, MPI.DOUBLE])
    y = numpy.dot(A, xg)
    return y
```

Image From MPI readthedocs

DASK

- Easy way of accessing distributed task-parallelism in the NumPy*/SciPy* ecosystem
- Comes with Task Graphs, Delayed wrappers, diagnostic server
- Can scale up and down quickly depending on needs (local computer, full cluster)

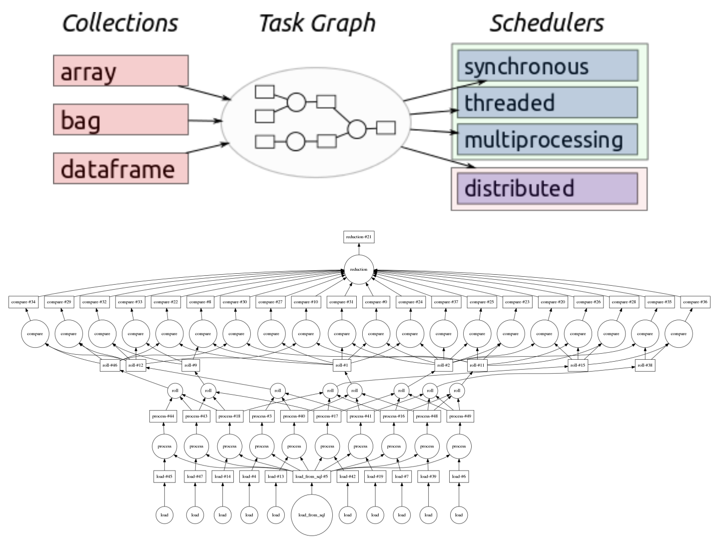


Image From Dask documentation*

DASK (CON'T)

- Extremely easy to integrate in places where NumPy* and SciPy* already exist
- Is a bit "heavier" of a solution than MPI, so use accordingly
- Works best when tasks have little intercommunication between workers

OTHER PYTHON-LEVEL ACCELERATORS

- Cython*
  - Optimizing static compiler
  - Similar syntax to Python
  - Can interact with NumPy* pretty well
  - Supports calling C/C++ well
- Numba*
  - Just-in-time (JIT) certain functions in Python
  - Optimizes down to Low Level Virtual Machine (LLVM) code
  - Useful for code that can be instantiated once and reused

NUMBA

- Accessed by using the @jit decorator
- May need special compilation options to get best out of it
- Can cache the function with cache=True
- Access vectorization with @vectorization decorator

```
from numba import jit


@jit
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255
```

Code snipit from the Numba documentation

CYTHON

- Can statically compile native code
- Can utilize static typing for faster code
- Compiles to C files
- Can pre-compile and import Cython code/modules
- Accessed with a package or via the %%cython in Jupyter notebooks

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Code from the Cython documentation

CAVEATS

From the Cython docs:

- *"The general recommendation is that you should only try to compile the critical paths in your code. If you have a piece of performance-critical computational code amongst some higher-level code, you may factor out the performance-critical code in a separate function and compile the separate function with Numba. Letting Numba focus on that small piece of performance-critical code has several advantages:*
  - *it reduces the risk of hitting unsupported features;*
  - *it reduces the compilation times;*
  - *it allows you to evolve the higher-level code which is outside of the compiled function much easier."*

© 2017 Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

VECTORIZATION

- Special form of parallelism converted from an initial scalar form
- Hardware supported parallelism of SIMD which can greatly assist numerical pipelines
- Main two components are numexpr* and the NumPy* that use vectorization
- Intel® Distribution for Python* does this for you with changes to NumPy*, SciPy*, Scikit-learn* etc.
- Occasionally using the raw numexpr* might fit one's use case

NUMEXPR: THE NUMERICAL EVALUATOR

- Multi-core, multi-threaded vectorization performance through Vector Math Library (VML), part of the Intel® MKL
- Best on large array size calculations, and transcendent expressions
- Callable from the Python-level
- Great for making changes that could call down to vectorization code without moving one's code to C++ level

```
In [1]: import numpy as np

In [2]: import numexpr as ne

In [3]: a = np.random.rand(1e6)

In [4]: b = np.random.rand(1e6)

In [5]: timeit 2*a + 3*b
10 loops, best of 3: 18.9 ms per loop

In [6]: timeit ne.evaluate("2*a + 3*b")
100 loops, best of 3: 5.83 ms per loop   # 3.2x: medi

In [7]: timeit 2*a + b**10
10 loops, best of 3: 158 ms per loop

In [8]: timeit ne.evaluate("2*a + b**10")
100 loops, best of 3: 7.59 ms per loop   # 20x: large
```

NUMEXPR (CON'T)

- Easy to intermix with NumPy* and SciPy* code
- Requires that you understand the numerical implications of your code
- This was one of the methods we accelerated NumPy* and SciPy* in our optimized IDP Package

```
>>> import numpy as np
>>> import numexpr as ne

>>> a = np.arange(1e6)   # Choose large arrays for better speedups
>>> b = np.arange(1e6)

>>> ne.evaluate("a + 1")   # a simple expression
array([  1.00000000e+00,   2.00000000e+00,   3.00000000e+00, ...,
         9.99998000e+05,   9.99999000e+05,   1.00000000e+06])

>>> ne.evaluate('a*b-4.1*a > 2.5*b')   # a more complex one
array([False, False, False, ...,  True,  True,  True], dtype=bool)

>>> ne.evaluate("sin(a) + arcsinh(a/b)")   # you can also use functions
array([        NaN,  1.72284457,  1.79067101, ...,  1.09567006,
        0.17523598, -0.09597844])

>>> s = np.array(['abba', 'abbb', 'abbcdef'])
>>> ne.evaluate("'abba' == s")   # string arrays are supported too
array([ True, False, False], dtype=bool)
```


PARALLELISM AND OTHER TOOLS: USAGE DETAILS

- Clearly understand one's workload and algorithms before implementing anything with these tools
- Profile one's code to more accurately understand where to make code changes
- Try different strategies and mixes of optimization to see where balance point is
- Documentation is you friend: many of these technologies have lots of gotchas and implementation quirks

OPTIMIZING THE BLACK SCHOLES* ALGORITHM

THE BLACK SCHOLES* ALGORITHM

- A financial options trading formula used for investment price estimates
- The formula calculates the price of a *European 'put'* and *'call'* options
- Is a partial differential equation (PDE) which describes the *price of the option over time*
- Is a great example of some of the optimization problems that exist in real-world

BLACK-SCHOLES* (CON'T)

- Algorithm is a PDE in general form
- Solvable for Call and Put options
- Goal is to solve for Call and Put options
- Putting it into Python is next step

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

$$C(S_t, t) = N(d_1)S_t - N(d_2)Ke^{-r(T-t)}$$

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right]$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

The price of a corresponding put option based on put-call parity is:

$$P(S_t, t) = Ke^{-r(T-t)} - S_t + C(S_t, t) = N(-d_2)Ke^{-r(T-t)} - N(-d_1)S_t$$

For both, as above:

- $N(\cdot)$ is the cumulative distribution function of the standard normal distribution
- $T-t$ is the time to maturity (expressed in years)
- S_t is the spot price of the underlying asset
- K is the strike price
- r is the risk free rate (annual rate, expressed in terms of continuous compounding)
- σ is the volatility of returns of the underlying asset

BLACK-SCHOLES* (CON'T)

- Code generates the intermediates of the formula, and gives the corresponding call/put
- Generates for as many options that exist (nopt)
- Calculates final call/put at the last two lines

```
from math import log, sqrt, exp, erf
import numpy as np
invsqrt = lambda x: 1.0/sqrt(x)

def black_scholes ( nopt, price, strike, t, rate, vol, call, put )
    mr = -rate
    sig_sig_two = vol * vol * 2

    for i in range(nopt):
        P = float( price [i] )
        S = strike [i]
        T = t [i]

        a = log(P / S)
        b = T * mr

        z = T * sig_sig_two
        c = 0.25 * z
        y = invsqrt(z)

        w1 = (a - b + c) * y
        w2 = (a - b - c) * y

        d1 = 0.5 + 0.5 * erf(w1)
        d2 = 0.5 + 0.5 * erf(w2)

        Se = exp(b) * S

        call [i] = P * d1 - Se * d2
        put [i] = call [i] - P + Se
```

BLACK SCHOLES* INITIAL ANALYSIS

- Where do you think the problems are in the code?
- What methods are you going to use to hunt them down?
- How much of this code is using performance libraries?

- Exercise: Come up with a game plan
  - Code is at: https://github.com/triskadecaepyon/ep2017_tutorial_tune_performance
  - Or just search Github for “ep2017_tutorial_tune_performance”

BLACK SCHOLES* INITIAL ANALYSIS (SETUP)

- You’ll need:
  - cProfile (included)
  - Line_profiler (conda install line_profiler)
  - Numexpr
  - Numba
  - Dask
  - Cython
  - Jupyter and Jupyter notebook
- Optional:
  - VTune Amplifier2017 XE or later

BLACK SCHOLES* INITIAL ANALYSIS

- What did you find?
- How did cProfile help?
- What did line_profiler do?

- Notes about profiling:
- cProfile:
  - use the import cProfile command, then cProfile.run('command')
- Line_profiler:
  - use "%load_ext line_profiler" in Jupyter
  - %lprun -f function function(args)

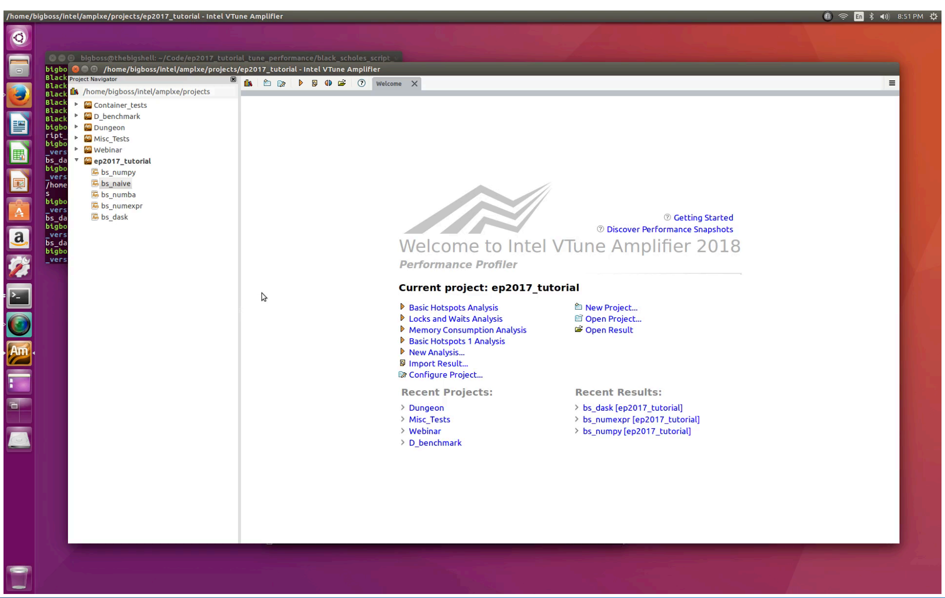
BLACK SCHOLES* INITIAL ANALYSIS (CPROFILE)

```
Fri Jun 16 15:58:01 2017    restats

         60004 function calls in 0.039 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    10000    0.003    0.000    0.004    0.000 <ipython-input-48-2d252d67ac99>:3(<lambda>)
        1    0.026    0.026    0.039    0.039 <ipython-input-48-2d252d67ac99>:5(black_scholes)
        1    0.000    0.000    0.039    0.039 <string>:1(<module>)
        1    0.000    0.000    0.039    0.039 {built-in method builtins.exec}
    20000    0.006    0.000    0.006    0.000 {built-in method math.erf}
    10000    0.001    0.000    0.001    0.000 {built-in method math.exp}
    10000    0.001    0.000    0.001    0.000 {built-in method math.log}
    10000    0.001    0.000    0.001    0.000 {built-in method math.sqrt}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

BLACK SCHOLES* INITIAL ANALYSIS (LINE_PROFILER)

```
Timer unit: 1e-06 s

Total time: 0.186871 s
File: <ipython-input-13-2d252d67ac99>
Function: black_scholes at line 5

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           def black_scholes ( nopt, price, strike, t, rate, vol, call, put ):
     6         1            2      2.0      0.0      mr = -rate
     7         1            2      2.0      0.0      sig_sig_two = vol * vol * 2
     8
     9     10001         8906      0.9      4.8      for i in range(nopt):
    10     10000        11370      1.1      6.1          P = float( price [i] )
    11     10000         9257      0.9      5.0          S = strike [i]
    12     10000         9262      0.9      5.0          T = t [i]
    13
    14     10000        11753      1.2      6.3          a = log(P / S)
    15     10000        10216      1.0      5.5          b = T * mr
    16
    17     10000        10405      1.0      5.6          z = T * sig_sig_two
    18     10000        10443      1.0      5.6          c = 0.25 * z
    19     10000        15951      1.6      8.5          y = invsqrt(z)
    20
    21     10000        13279      1.3      7.1          w1 = (a - b + c) * y
    22     10000        12288      1.2      6.6          w2 = (a - b - c) * y
    23
    24     10000        13464      1.3      7.2          d1 = 0.5 + 0.5 * erf(w1)
    25     10000        13741      1.4      7.4          d2 = 0.5 + 0.5 * erf(w2)
    26
    27     10000        11917      1.2      6.4          Se = exp(b) * S
    28
    29     10000        12540      1.3      6.7          call [i] = P * d1 - Se * d2
    30     10000        12075      1.2      6.5          put [i] = call [i] - P + Se
```


WHAT VTUNE SHOWS FROM THE EXAMPLE



ONE FORM OF OPTIMIZATION: NUMPY*-SPECIFIC MATH CALLS

- **Exercise:** In this example, replace the functions from the math library with NumPy* equivalents:
  - log
  - exp
  - erf
  - invsqrt
- Re-run the profiling to see what you can find
  - Total time?
  - A change in what the bottlenecks were?

BLACK SCHOLES*: NUMPY VARIANT

- Test out changes with NumPy* to the Naïve implementation of Black Scholes*
- Test with same methods: timeit, cProfile, line_profiler
- What works? What doesn't work?

BLACK SCHOLES*: NUMPY VARIANT (VECTORIZED)

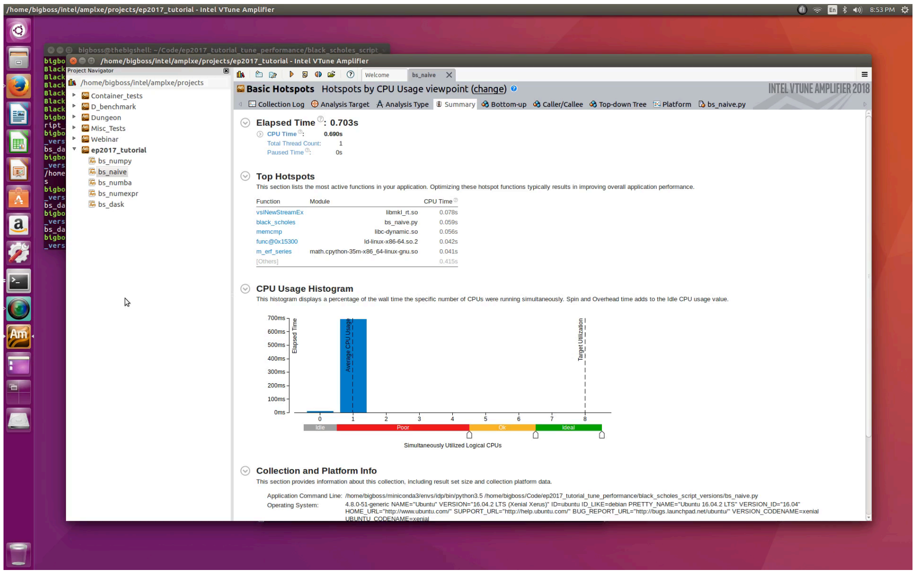
- Test out changes with NumPy* to the vectorized implementation of Black Scholes*
- Test with same methods: timeit, cProfile, line_profiler
- What works? What doesn't work?

BLACK SCHOLES*: NUMPY* VARIANT (VECTORIZED)

- Loop removal helps by allowing use of NumPy's native array capabilities
- Individually going through loops, even with NumPy* arrays is VERY expensive
- Loop-parallel has a few options, and this is one of them: vectorization!
- On line_profiler, how many times did the code hits changes in this new version?

© 2017 Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

VTUNE ANALYSIS OF BLACK SCHOLES* WITH NUMPY*



BLACK SCHOLES*: NUMEXPR*

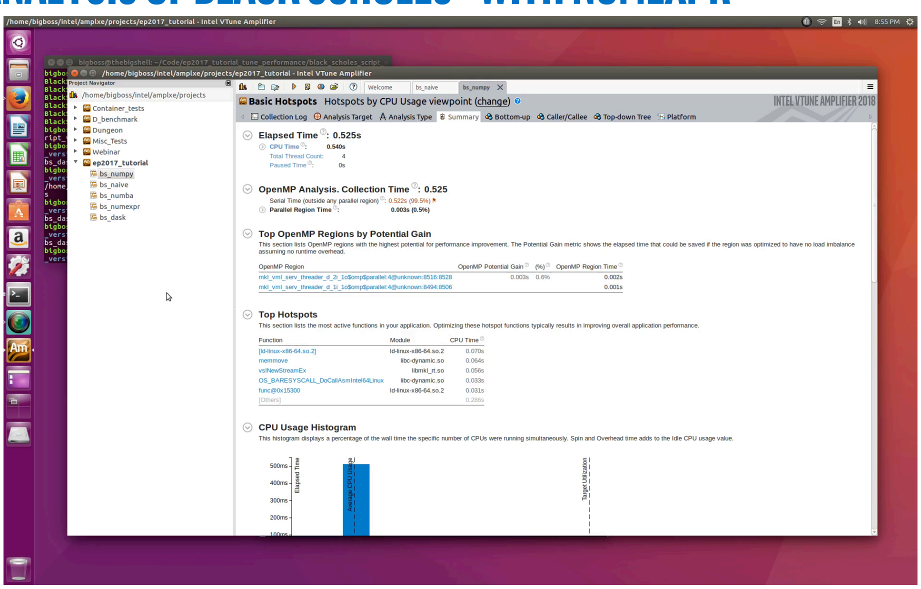
- Exercise: Modify the Black Scholes* algorithm to utilize numexpr*, and re-run the same tests
- Test with same methods: timeit, cProfile, line_profiler
- What works? What doesn't work?

- What about the condensed version? How well does that work?

BLACK SCHOLES*: NUMEXPR*

- By interacting directly with numexpr*, you are calling out to the vectorization capabilities without going through the NumPy* layer
- By compressing the entire vectorization command of one's calculation in one expression, the vectorization engine can do significantly more
- This is one of the ways we did some of our optimization work on NumPy* itself for the Intel® Distribution for Python*!

VTUNE ANALYSIS OF BLACK SCHOLES* WITH NUMEXPR



BLACK SCHOLES*: NUMBA*

- **Exercise**: Using the Numba example, test with same methods: timeit, cProfile, line_profiler
- What do you notice about the functions being imported?
- Why do you think it uses the "nopython=True" option?
- What works? What doesn't work?

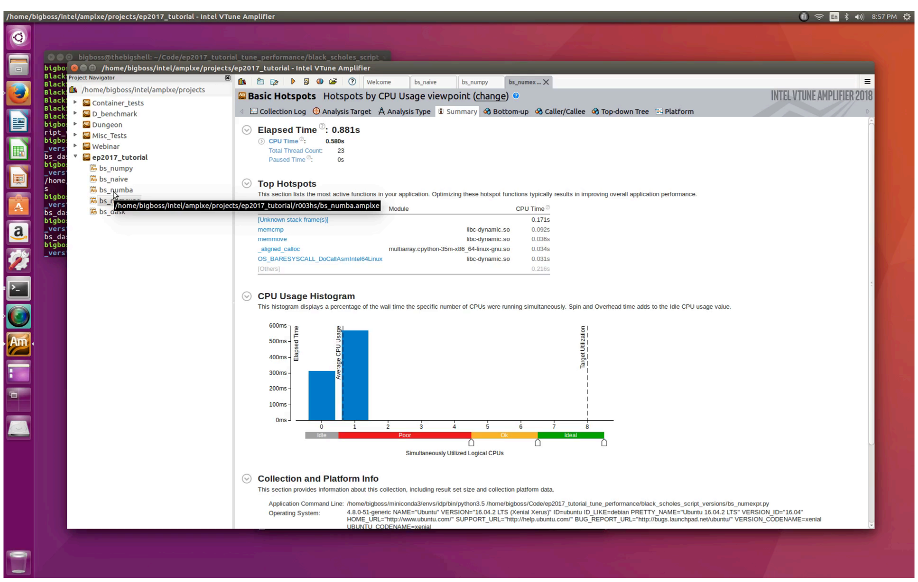
BLACK SCHOLES*: NUMBA* (VARIANT 2)

- What is different in this example? What does it change?
- Using the Numba example, test with same methods: timeit, cProfile, line_profiler
- What works? What doesn't work?

BLACK SCHOLES*: NUMBA* (VARIANT 3)

- What is different in this example? What does it change?
- Using the Numba example, test with same methods: timeit, cProfile, line_profiler
- What works? What doesn't work?

VTUNE ANALYSIS OF BLACK SCHOLES* WITH NUMBA*



BLACK SCHOLES*: NUMBA*

- This example uses Just-In-Time(JIT) compiling to achieve performance gains
- Because of this, profiling can become VERY difficult
- The first run is slow because you pay for the compilation time, but the function is cached afterwards
- Many times this require writing in pure Python before utilizing Numba

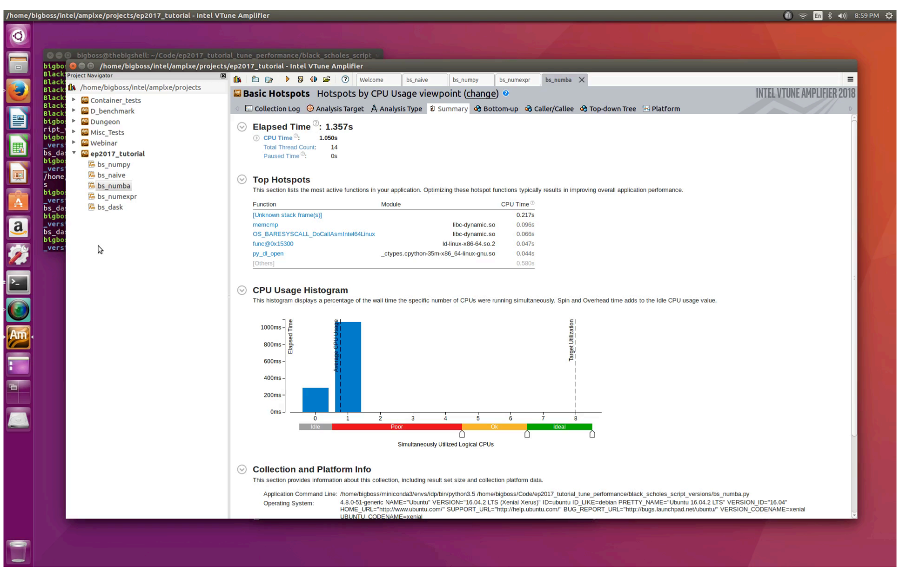
BLACK SCHOLES*: DASK*

- Exercise: What is different in this example? What does it change?
- Using this example, test with same methods: timeit, cProfile, line_profiler
- What works? What doesn't work?

BLACK SCHOLES*: DASK* (NUMPY* MODS)

- What is different in this example? What does it change?
- Using this example, test with same methods: timeit, cProfile, line_profiler
- How does the diagnostic server help?
- What works? What doesn't work?

VTUNE ANALYSIS OF BLACK SCHOLES* WITH DASK



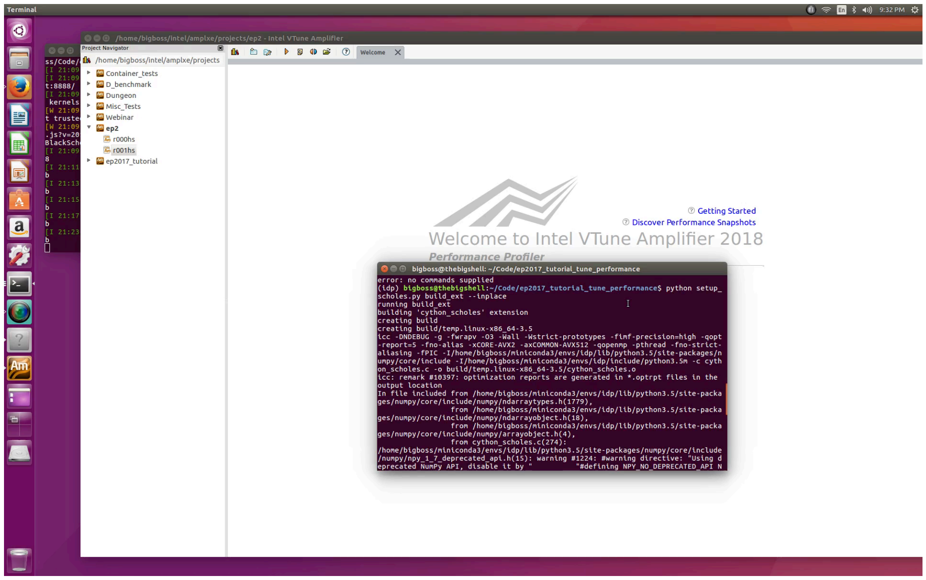
BLACK SCHOLES*: CYTHON*

- Exercise: What is different in this example? What does it change?
- Take a look at the .pyx file provided, then follow the instructions to build the Cython* model
- If you have the Intel® Compiler (icc), the resultant code will be MUCH faster; gcc does not do very good vectorization!
- Using this example, test with same methods: timeit, cProfile, line_profiler

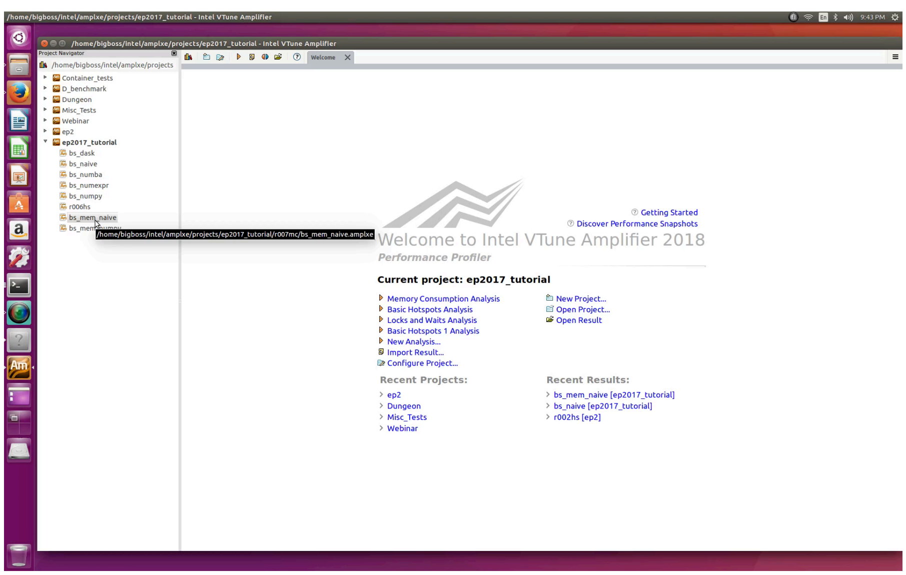
BLACK SCHOLES*: CYTHON

- Cython* is another method of getting performance closer to C that has similar syntax to Python
- Essentially applies some of the rigidity of C to Python in trade for better performance
- Some annoyances on occasion about importing the code, makes testing the code in production a bit difficult (as well as deployment)
- Best performance is achieved with use of a performance compiler, such as icc.

VTUNE ANALYSIS OF BLACK SCHOLES* WITH CYTHON*



VTUNE AND MEMORY CONSUMPTION ANALYSIS



BLACK SCHOLES*: A SUMMARY

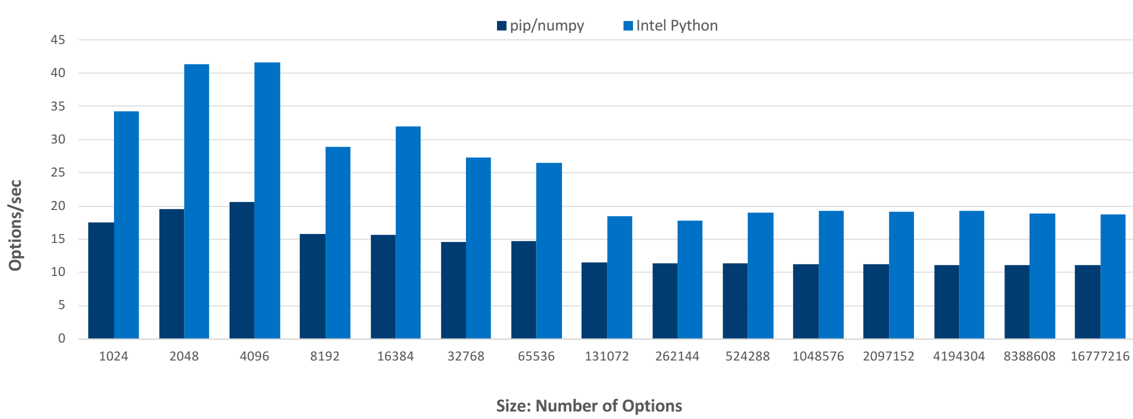
- With these examples, a proper strategy and methodical testing w/ tools can properly accelerate one's code properly
- Understanding which technologies are good for what purposes can help with selecting the best optimization technique for one's code
- Use of proper code profilers for the job can also help significantly
- Advanced profilers such as VTune can reveal much more about how a problem should be optimized (and what tools to use)
- Remember that parallelism is something that takes much effort to achieve, but the benefits can be tremendous

BLACK SCHOLES* BENCHMARKS

i5

Black Scholes algorithm on i5 processors (2017 Update 2)

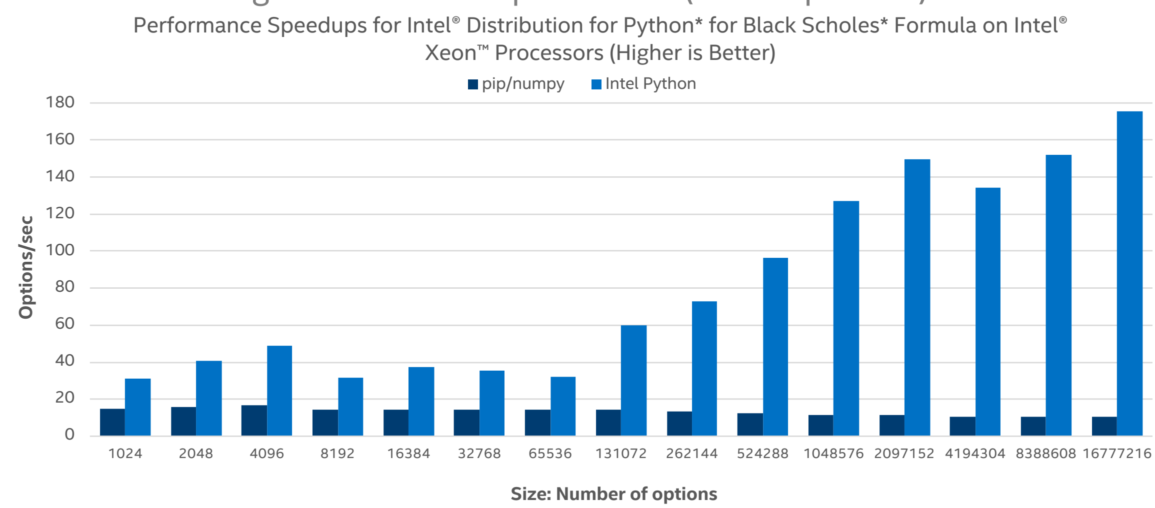
Performance Speedups for Intel® Distribution for Python* for Black Scholes* Formula on Intel® Core™ i5 Processor (Higher is Better)



BLACK SCHOLES* BENCHMARKS

Xeon

Black Scholes algorithm on Xeon processors (2017 Update 2)

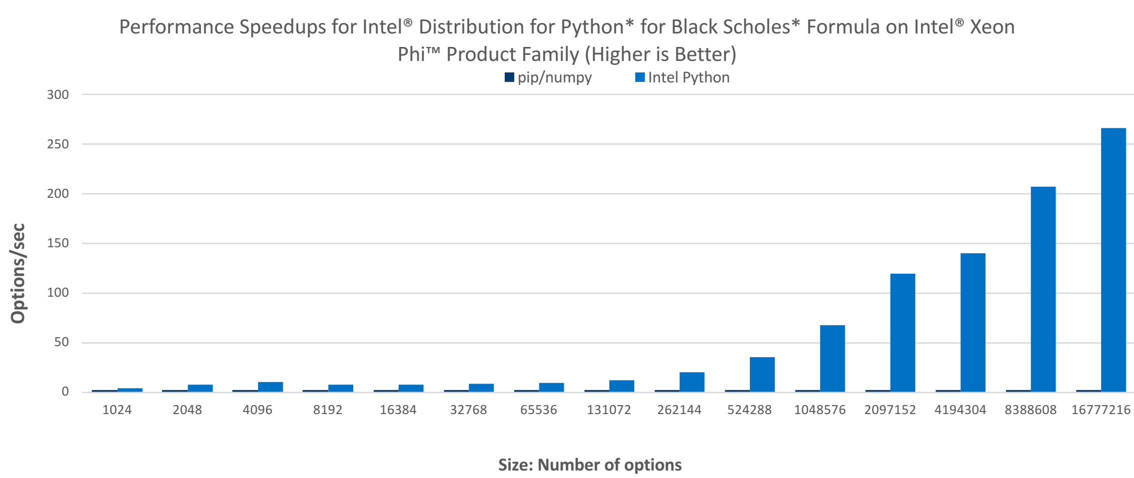


intel Software | 79

BLACK SCHOLES* BENCHMARKS

Xeon Phi

Black Scholes algorithm on Xeon Phi processors (2017 Update 2)



intel Software | 80

CONFIGURATION INFORMATION

Software

- Pip*/NumPy*: Installed with Pip, Ubuntu*, Python* 3.5.2, NumPy=1.12.1, scikit-learn*=0.18.1
- Windows*, Python 3.5.2, Pip/NumPy=1.12.1, scikit-learn=0.18.1
- Intel® Distribution for Python 2017, Update 2

Hardware

- Intel® Core™ i5-4300M processor @ 2.60 GHz 2.59 GHz, (1 socket, 2 cores, 2 threads per core), 8GB DRAM
- Intel® Xeon® E5-2698 v3 processor @ 2.30 GHz (2 sockets, 16 cores each, 1 thread per core), 64GB of DRAM
- Intel® Xeon Phi™ processor 7210 @ 1.30 GHz (1 socket, 64 cores, 4 threads per core), DRAM 32 GB, MCDRAM (Flat mode enabled) 16GB

Modifications

- Scikit-learn: conda installed NumPy with Intel® Math Kernel Library (Intel® MKL) on Windows (pip install scipy on Windows contains Intel® MKL dependency)
- Black Scholes* on Intel Core i5 processor/Windows: Pip installed NumPy and conda installed SciPy

HOW WERE THESE OPTIMIZATIONS DONE?

- Many of the changes initially leverage research on NumPy* vectorization code
- Changes were made at the numexpr* level (such as the ones that were shown), in NumPy's source
- Additional changes were done at the C level with the Intel MKL
- Notice that even with these changes that should help the stock pip version, it does not scale very well
- Advanced vectorization through AVX 2.0 and AVX512 really help the algorithm scale out on hardware

BLACK SCHOLES EXAMPLE REFERENCES

- https://github.com/IntelPython/BlackScholes_bench
- https://en.wikipedia.org/wiki/Black–Scholes_model
- Multiprocessing, MPI Variants
  - https://github.com/IntelPython/BlackScholes_bench/tree/parallel2017

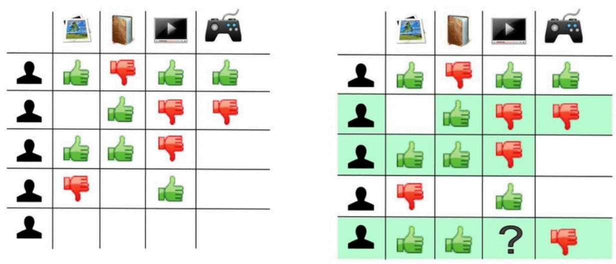
intel Software | 83

intel Software

COLLABORATIVE FILTERING EXAMPLE

COLLABORATIVE FILTERING EXAMPLE

- Exercise: optimize Collaborative filtering
- Collaborative filtering is used by *recommender systems*.
- Uses dot product/cosine similarity to generate similarity calculation (memory-based)



$$\text{simil}(x,y) = \frac{\sum\limits_{i \in I_{xy}} (r_{x,i} - \bar{r_x})(r_{y,i} - \bar{r_y})}{\sqrt{\sum\limits_{i \in I_{xy}} (r_{x,i} - \bar{r_x})^2 \sum\limits_{i \in I_{xy}} (r_{y,i} - \bar{r_y})^2}}$$

where I_{xy} is the set of items rated by both user x and user y.

The cosine-based approach defines the cosine-similarity between two users x and y as:[4]

$$\text{simil}(x,y) = \cos(\vec{x},\vec{y}) = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}|| \times ||\vec{y}||} = \frac{\sum\limits_{i \in I_{xy}} r_{x,i} r_{y,i}}{\sqrt{\sum\limits_{i \in I_x} r_{x,i}^2} \sqrt{\sum\limits_{i \in I_y} r_{y,i}^2}}$$

intel Software

COLLABORATIVE FILTERING EXAMPLE: METHODS

- Similar to Black Scholes*, utilize timeit, cProfile, line_profiler to determine how the algorithms perform and what can be seen
- Several examples to demonstrate parallelism methods:
  - NumPy*
  - Dask*
  - Numba*
  - NumPy*+Numba*
  - Dask*+Numba*

intel Software | 86

COLLABORATIVE FILTERING EXAMPLE: ANALYSIS

- What can you see about the example?
- How do the different variants fair against each other?
- How do the composable variants compare?
- Why do you think the composable variants work well?
- What method(s) would you use?

COLLABORATIVE FILTERING REFERENCES

- https://github.com/IntelPython/composability_bench/blob/master/collab_filt.py
- https://github.com/IntelPython/composability_bench
- https://en.wikipedia.org/wiki/Collaborative_filtering

CODE PROFILING EXAMPLES SUMMARY

- Profiling code as a starting point helps guide what methods one decides to look for optimization
- Developing one's ability to see inherent parallelism, and composable parallelism levels can help as one develops future codebases
- Use of the correct profiler for the job will help validate one's changes to performance code
- Knowledge and increased usage of performance libraries+vectorization will ensure one's tuning efforts are realized
- Parallelism is a diverse space; lots of things happening in the Python world!

intel Software

NEXT UP: PYCOMPSS FROM BARCELONA SUPERCOMPUTING CENTER

ADDITIONAL INFORMATION

- Intel® Distribution for Python* Documentation
  - https://software.intel.com/en-us/intel-distribution-for-python-support/documentation
- 2018 Beta information:
  - https://software.intel.com/en-us/articles/intel-parallel-studio-xe-2018-beta
- cProfile:
  - https://docs.python.org/3.5/library/profile.html
- Line_profiler:
  - https://github.com/rkern/line_profiler

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.

For more complete information about compiler optimizations, see our Optimization Notice at https://software.intel.com/en-us/articles/optimization-notice#opt-en.


intel
Software