MORE PERFORMANCE USING INTEL® DISTRIBUTION FOR PYTHON

Frank Schlimbach
Michael Steyer
Intel Corporation

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

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

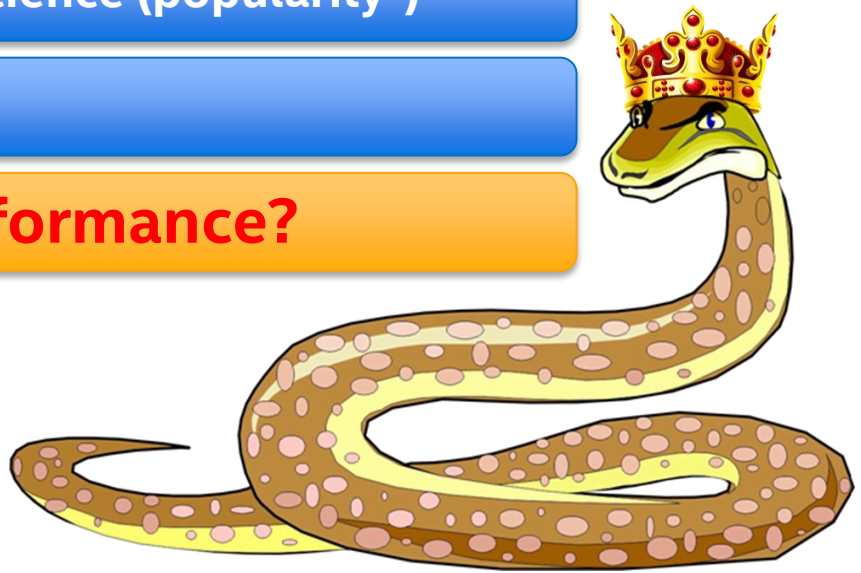
Most Wanted: Python

~7,8 Million out of ~21 Million developers use python (EDC 2016)

#1 language for machine learning/data science (popularity[1])

>100000 python packages on PyPI

Often slow. **Can we get better performance?**



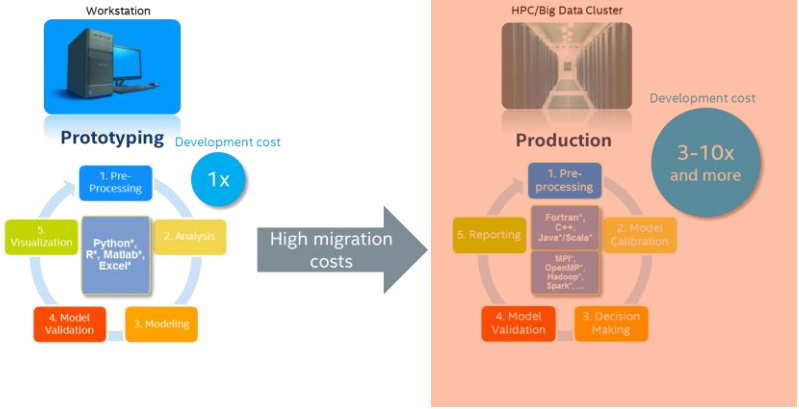
[1] https://www.kdnuggets.com/2017/01/most-popular-language-machine-learning-data-science.html

Faster Python* with Intel® Distribution for Python 2018

High Performance Python Distribution

- Accelerated NumPy, SciPy, scikit-learn well suited for scientific computing, machine learning & data analytics
- Drop-in replacement for existing Python. No code changes required
- Highly optimized for latest Intel processors
- Take advantage of Priority Support – connect direct to Intel engineers for technical questions[1]

Get sufficient performance without an extra development cycle



[1]Paid versions only.

What's Inside Intel® Distribution for Python

High Performance Python* for Scientific Computing, Data Analytics, Machine Learning

FASTER PERFORMANCE	GREATER PRODUCTIVITY	ECOSYSTEM COMPATIBILITY
Performance Libraries, Parallelism, Multithreading, Language Extensions	**Prebuilt & Accelerated Packages**	**Supports Python 2.7 & 3.6, conda, pip**
Accelerated NumPy/SciPy/scikit-learn with Intel® MKL[1] & Intel® DAAL[2]	Prebuilt & optimized packages for numerical computing, machine/deep learning, HPC, & data analytics	Compatible & powered by Anaconda*, supports conda & pip
Data analytics, machine learning & deep learning with scikit-learn, pyDAAL	Drop in replacement for existing Python - No code changes required	Distribution & individual optimized packages also available at conda & Anaconda.org, YUM/APT, Docker image on DockerHub
Scale with Numba* & Cython*	Jupyter* notebooks, Matplotlib included	Optimizations upstreamed to main Python trunk
Includes optimized mpi4py, works with Dask* & PySpark*	Conda build recipes included in packages	Commercial support through Intel® Parallel Studio XE 2017
Optimized for latest Intel® architecture	Free download & free for all uses including commercial deployment	

Intel® Architecture Platforms

Operating System: Windows*, Linux*, MacOS[1]*

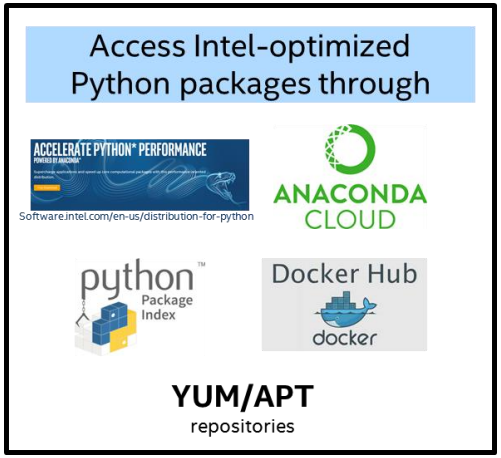
[1]Intel® Math Kernel Library
[2]Intel® Data Analytics Acceleration Library

[1] Available only in Intel® Parallel Studio Composer Edition.

What's New? Intel® Distribution for Python*

What's New in 2018 version

- Updated to latest version of Python 3.6
- Optimized scikit-learn for machine learning speedups
- Conda build recipes for custom infrastructure



What's new in 2019 beta?

Faster Machine learning with Scikit-learn functions

- Support Vector Machine (SVM) and K-means prediction, accelerated with Intel® DAAL

Built-in access to XGBoost library for Machine Learning

- Access to Distributed Gradient Boosting algorithms

Ease of access installation

- Now integrated into Intel® Parallel Studio XE installer.

Day 1 of the new CPU launch

Time

Intel® Distribution for Python*

yum apt

Working with Python vendors

Python Vendor 1

Python Vendor 2

Python Vendor 3

Anaconda Cloud*

Intel conda packages with build recipes & optimization patches

Working with community to upstream

Wheels for Intel runtimes and development packages (MKL, DAAL, TBB, etc.) and core packages (intel-numpy, intel-scipy,...)

python™ Package Index

GitHub

PRs with optimization patches

Installing Intel® Distribution for Python* 2018

Standalone Installer

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

ANACONDA CLOUD

anaconda.org
anaconda.org/intel channel

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

python™ Package Index

pip
Intel-optimized packages

```
> pip install intel-<pkg-name>
```

Docker Hub
docker

Docker Hub

```
> docker pull intelpython/intelpython3_full
```

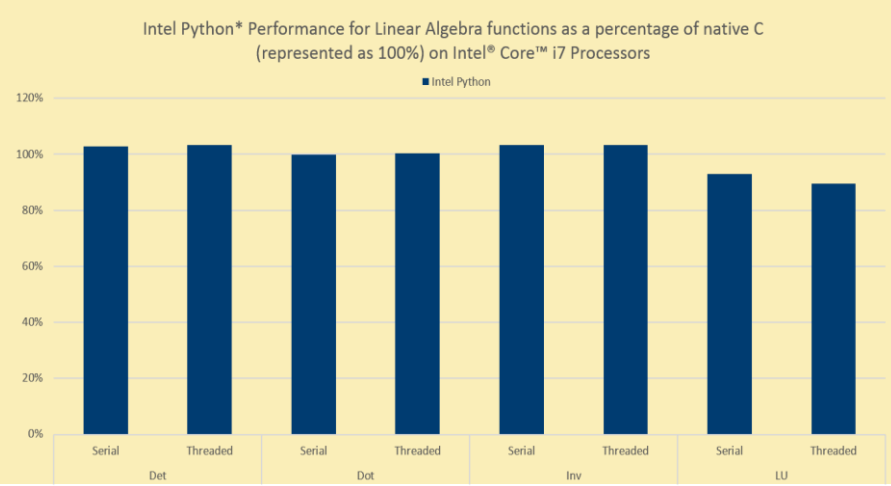
YUM/APT
repositories

yum/apt

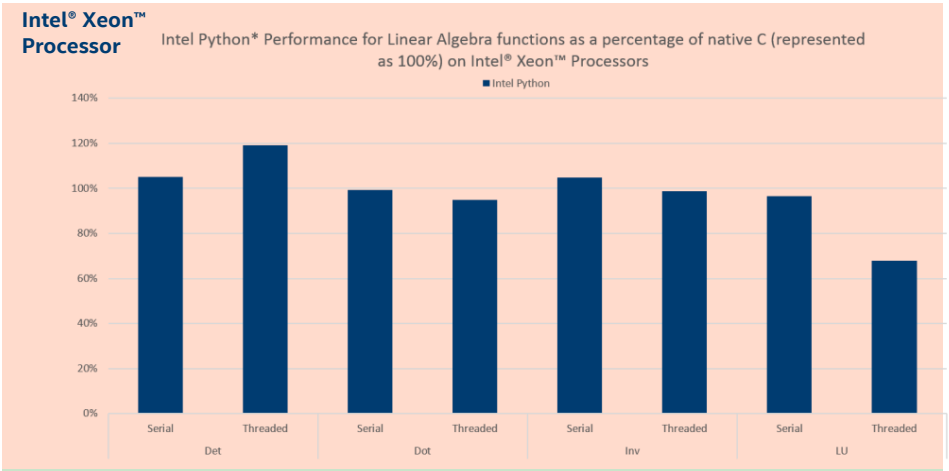
Access for yum/apt:
https://software.intel.com/en-us/articles/installing-intel-free-libs-and-python

Native Code like Performance Speeds for Linear Algebra

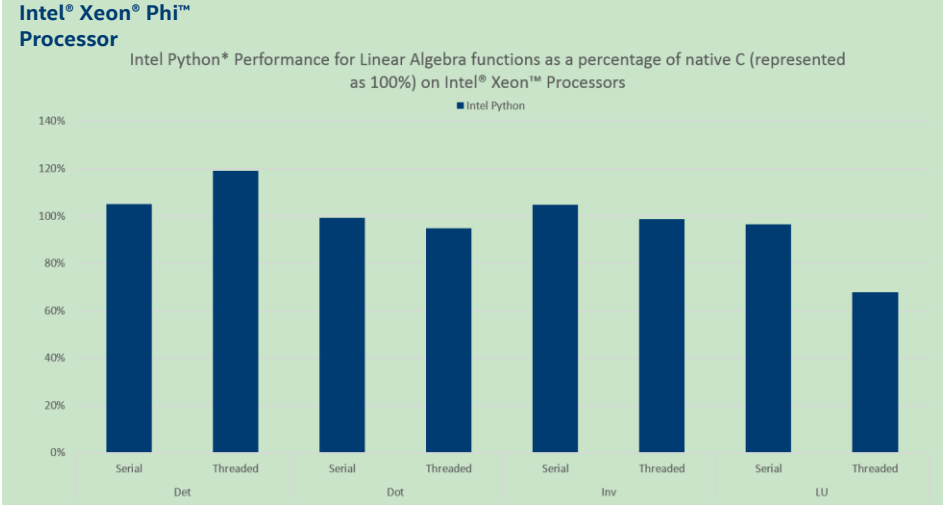
Intel® Core™ i7 Processor



Intel® Xeon™ Processor

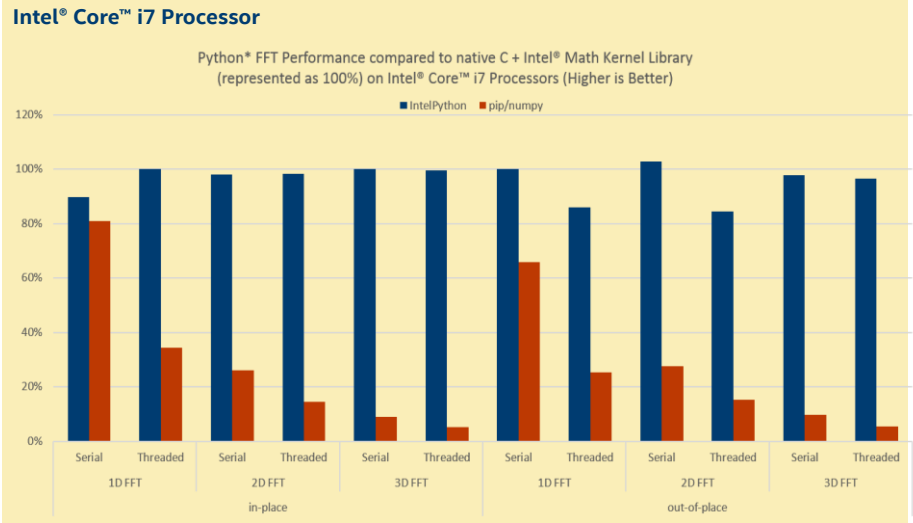


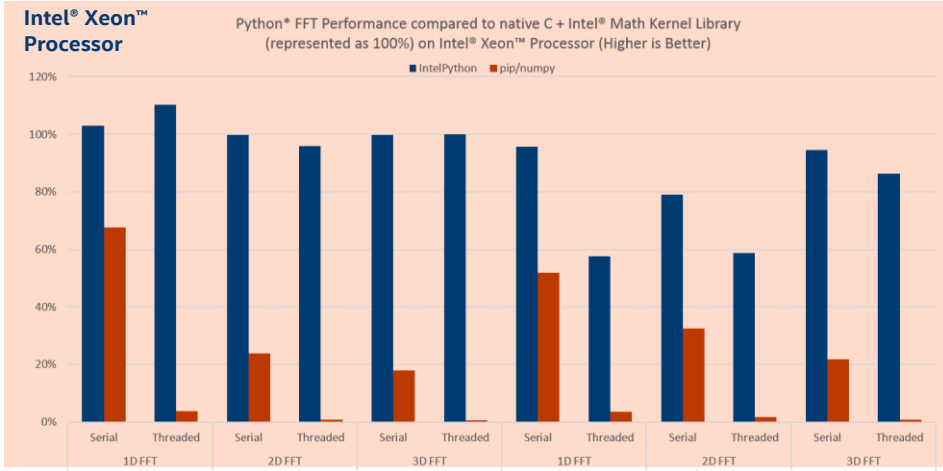
Intel® Xeon® Phi™ Processor

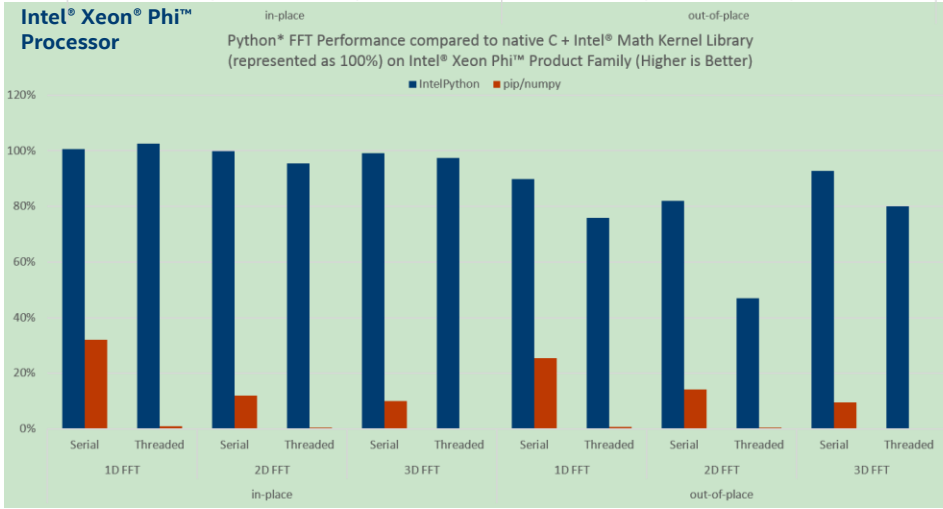


Hardware: Intel® Core™ i7-7567U CPU@3.50GHz (1 socket, 2 cores per socket, 2 threads per core), 32GB DDR4 @ 2133MHz. Intel® Xeon® CPU E5-2699 v4@2.20GHz (2 sockets, 22 cores per socket, 1 thread per core-HT is off), 256GB DDR4@2400MHz. Intel® Xeon Phi™ CPU 7250@1.40GHz (1 socket, 68 cores per socket, 4 threads per core), 192GB DDR4 @1200MHz, 16GB MCDRAM@7200MHz in cache mode
Software: Stock: CentOS Linux release 7.3.1611 (Core), python 3.6.2, pip 9.0.1, numpy 1.13.1, scipy 0.19.1, scikit-learn 0.19.0
Intel® Distribution for Python 2018 Gold packages: mkl 2018.0.0 intel_4, daal 2018.0.0.20170814, numpy 1.13.1 py36_intel_15, openmp 2018.0.0 intel_7, scipy 0.19.1 np113py36_intel_11, scikit-learn 0.18.2 np113py36_intel_3

Native C-like Performance Speedups for Fast Fourier Transforms



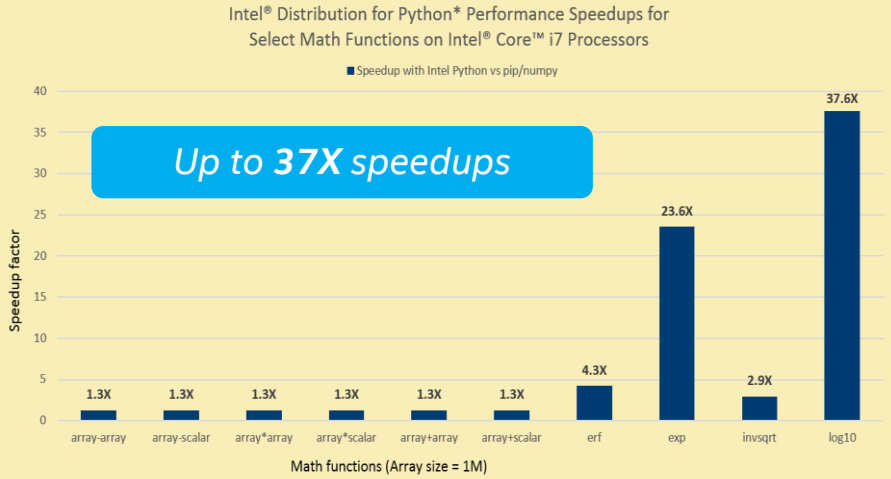




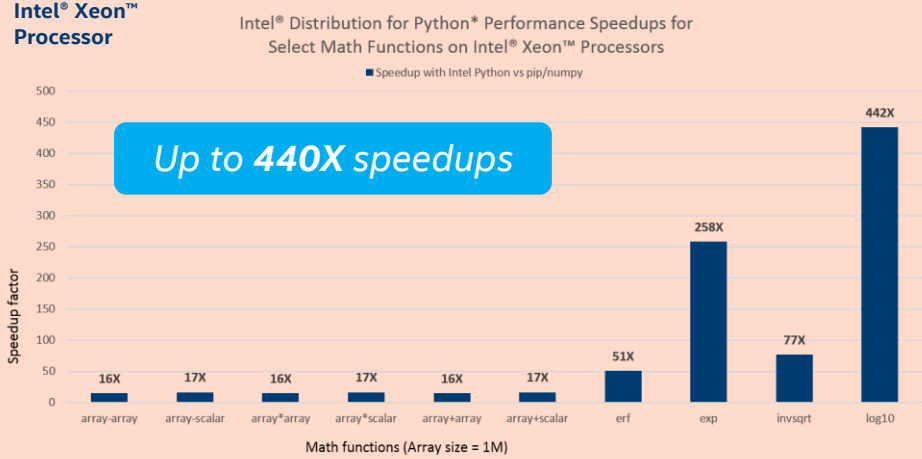
Hardware: Intel® Core™ i7-7567U CPU@3.50GHz (1 socket, 2 cores per socket, 2 threads per core), 32GB DDR4 @ 2133MHz. Intel® Xeon® CPU E5-2699 v4@2.20GHz (2 sockets, 22 cores per socket, 1 thread per core-HT is off), 256GB DDR4@2400MHz. Intel® Xeon Phi™ CPU 7250@1.40GHz (1 socket, 68 cores per socket, 4 threads per core), 192GB DDR4 @1200MHz, 16GB MCDRAM@7200MHz in cache mode
Software: Stock: CentOS Linux release 7.3.1611 (Core), python 3.6.2, pip 9.0.1, numpy 1.13.1, scipy 0.19.1, scikit-learn 0.19.0
Intel® Distribution for Python 2018 Gold packages: mkl 2018.0.0 intel_4, daal 2018.0.0.20170814, numpy 1.13.1 py36_intel_15, openmp 2018.0.0 intel_7, scipy 0.19.1 np113py36_intel_11, scikit-learn 0.18.2 np113py36_intel_3

UMath Optimizations & Vectorization to Utilize Multiple Cores, Memory Management

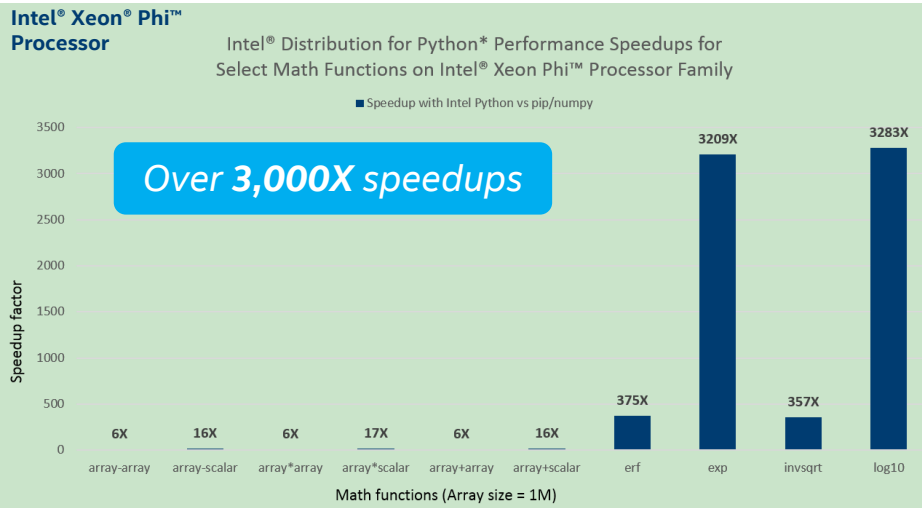
Intel® Core™ i7 Processor



Intel® Xeon™ Processor



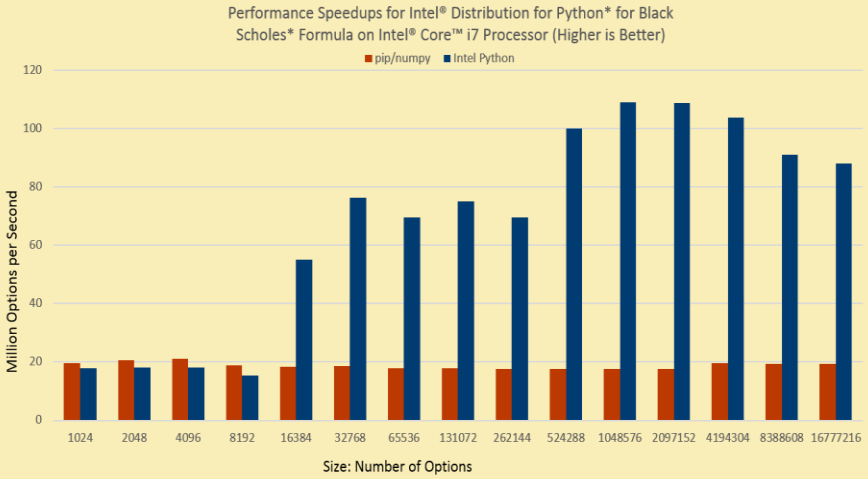
Intel® Xeon® Phi™ Processor



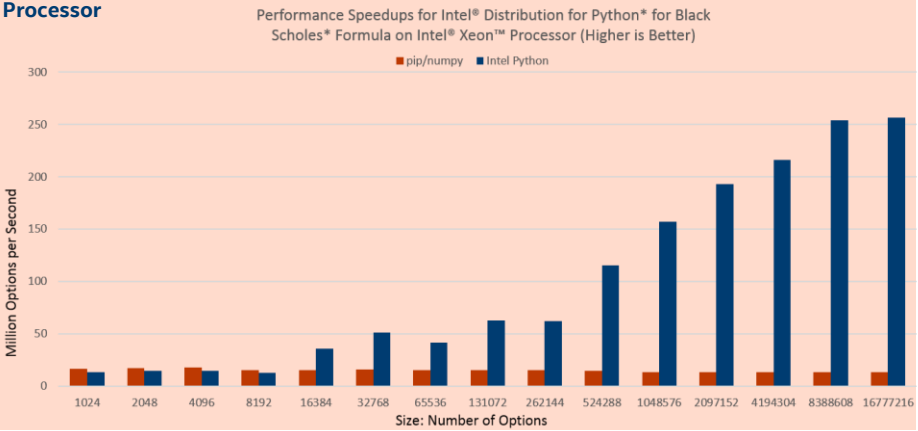
Hardware: Intel® Core™ i7-7567U CPU@3.50GHz (1 socket, 2 cores per socket, 2 threads per core), 32GB DDR4 @ 2133MHz. Intel® Xeon® CPU E5-2699 v4@2.20GHz (2 sockets, 22 cores per socket, 1 thread per core-HT is off), 256GB DDR4@2400MHz. Intel® Xeon Phi™ CPU 7250@1.40GHz (1 socket, 68 cores per socket, 4 threads per core), 192GB DDR4 @1200MHz, 16GB MCDRAM@7200MHz in cache mode
Software: Stock: CentOS Linux release 7.3.1611 (Core), python 3.6.2, pip 9.0.1, numpy 1.13.1, scipy 0.19.1, scikit-learn 0.19.0
Intel® Distribution for Python 2018 Gold packages: mkl 2018.0.0 intel_4, daal 2018.0.0.20170814, numpy 1.13.1 py36_intel_15, openmp 2018.0.0 intel_7, scipy 0.19.1 np113py36_intel_11, scikit-learn 0.18.2 np113py36_intel_3

Performance Speedups for Black Scholes Formula

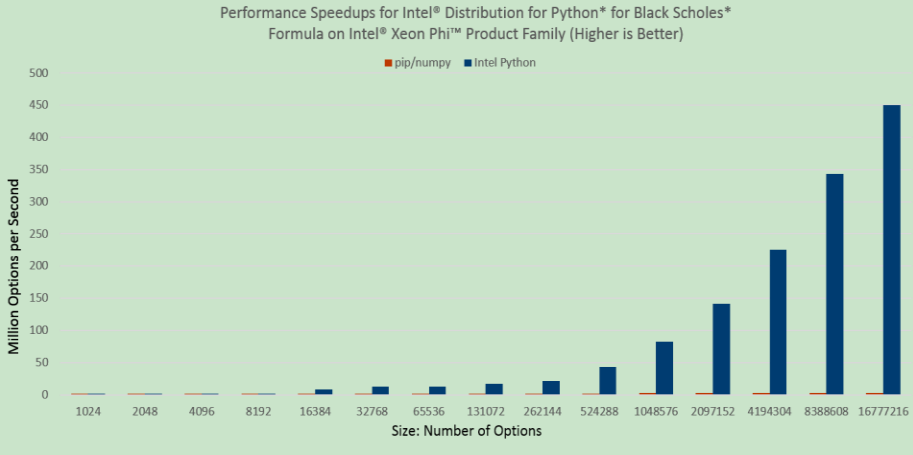
Intel® Core™ i7 Processor



Intel® Xeon® Processor



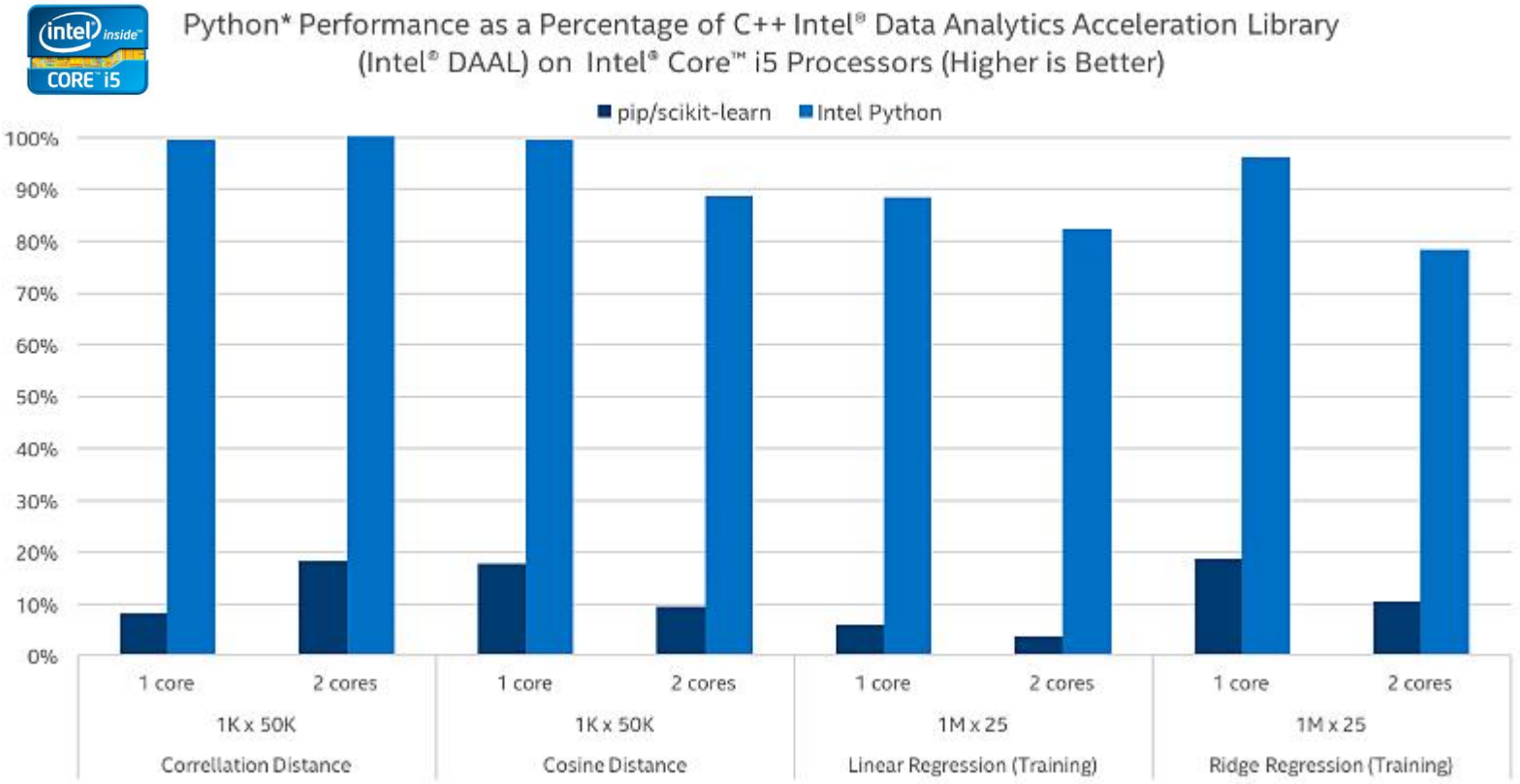
Intel® Xeon® Phi™ Processor



Hardware: Intel® Core™ i7-7567U CPU@3.50GHz (1 socket, 2 cores per socket, 2 threads per core), 32GB DDR4 @ 2133MHz. Intel® Xeon® CPU E5-2699 v4@2.20GHz (2 sockets, 22 cores per socket, 1 thread per core-HT is off), 256GB DDR4@2400MHz. Intel® Xeon Phi™ CPU 7250@1.40GHz (1 socket, 68 cores per socket, 4 threads per core), 192GB DDR4 @1200MHz, 16GB MCDRAM@7200MHz in cache mode
Software: Stock: CentOS Linux release 7.3.1611 (Core), python 3.6.2, pip 9.0.1, numpy 1.13.1, scipy 0.19.1, scikit-learn 0.19.0
Intel® Distribution for Python 2018 Gold packages: mkl 2018.0.0 intel_4, daal 2018.0.0.20170814, numpy 1.13.1 py36_intel_15, openmp 2018.0.0 intel_7, scipy 0.19.1 np113py36_intel_11, scikit-learn 0.18.2 np113py36_intel_3

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

Scikit-Learn* optimizations



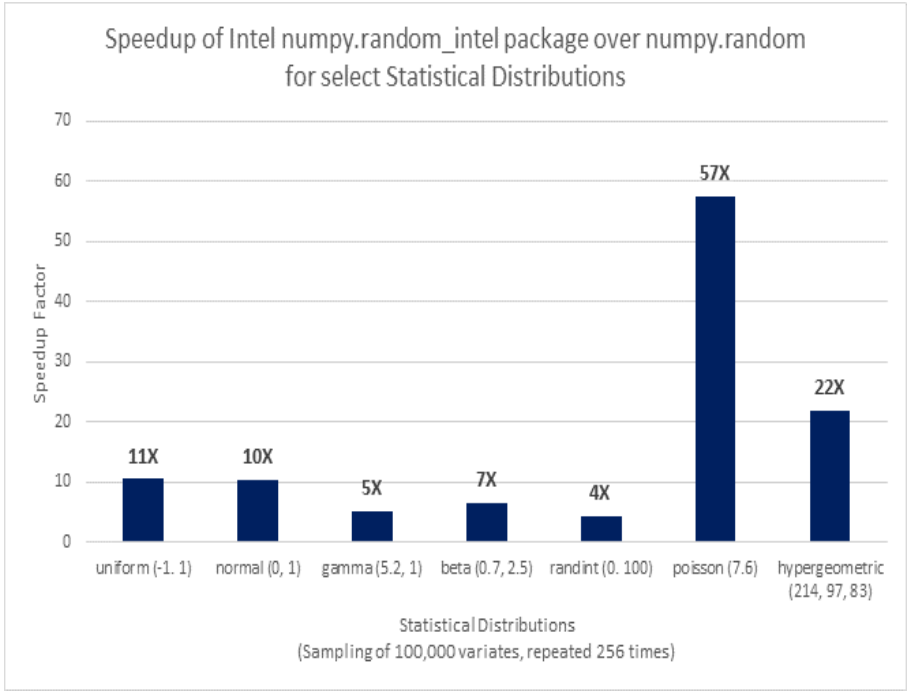
https://github.com/IntelPython/mkl_random

MKL-based RANDOM NUMBER GENERATION

Added numpy subpackage numpy.random_intel that mirrors numpy.random in the scope

Exposes all basic random number generators provided in Intel® MKL.

```
In [3]: import numpy as np, numpy.random_intel as irnd, numpy.random as vrnd

In [4]: irnd.seed(1234,brng='SFMT19937')

In [5]: %time x1 = irnd.randn(10**6)
        CPU times: user 4 ms, sys: 4 ms, total: 8 ms
        Wall time: 7.73 ms

In [6]: %time x2 = vrnd.randn(10**6)
        CPU times: user 44 ms, sys: 0 ns, total: 44 ms
        Wall time: 44.8 ms
```



Hardware: Intel® Xeon® CPU E5-2699 v4@2.20GHz (2 sockets, 22 cores per socket, 1 thread per core-HT is off), 256GB DDR4@2400MHz.
Software: Stock: CentOS Linux release 7.3.1611 (Core), python 3.6.2, pip 9.0.1, numpy 1.13.1, scipy 0.19.1, scikit-learn 0.19.0
Intel® Distribution for Python 2018 Gold packages: mkl 2018.0.0 intel_4, daal 2018.0.0.20170814, numpy 1.13.1 py36_intel_15, openmp 2018.0.0 intel_7, scipy 0.19.1 np113py36_intel_11, scikit-learn 0.18.2 np113py36_intel_3

But Wait…..There's More!

Outside of optimized Python*, how efficient is your Python/C/C++ application code?

Are there any non-obvious sources of performance loss?

Performance analysis gives the answer!

Tune Python* + Native Code for Better Performance

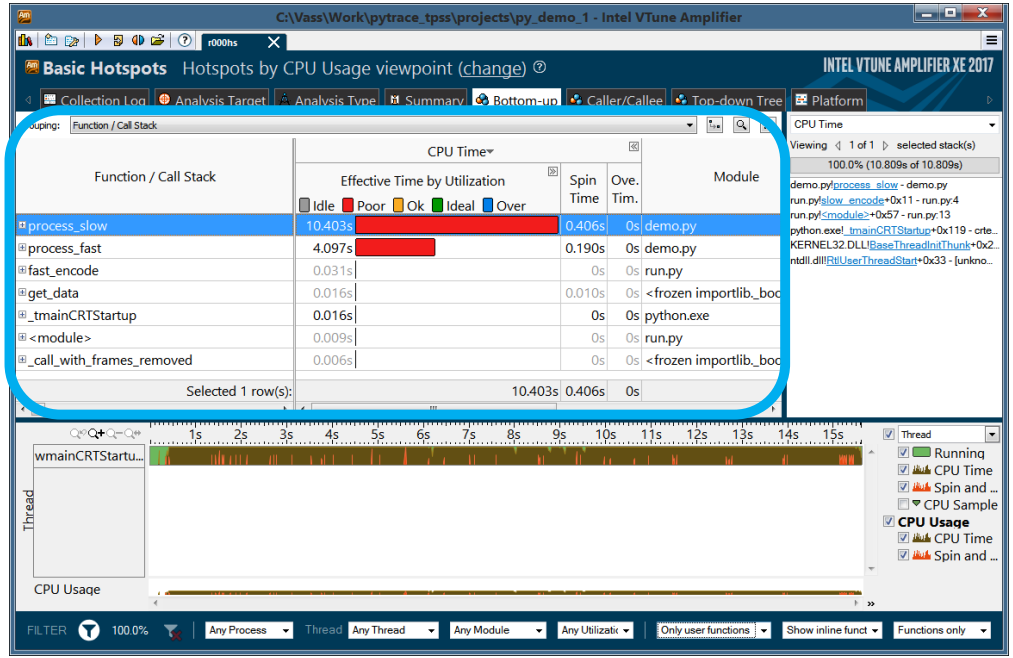
Analyze Performance with Intel® VTune™ Amplifier (available in Intel® Parallel Studio XE)

Challenge

- Single tool that profiles Python + native mixed code applications
- Detection of inefficient runtime execution

Solution

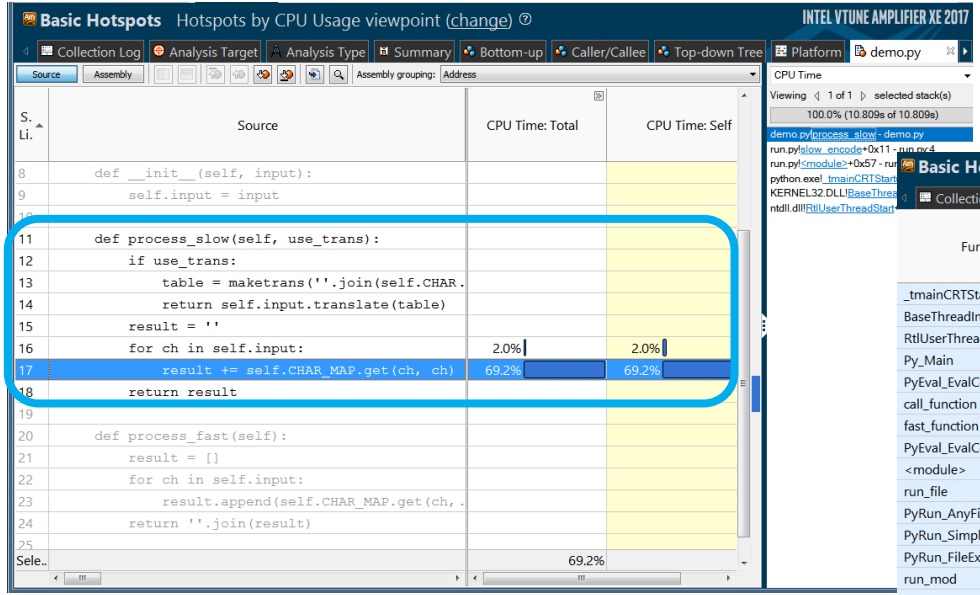
- Auto-detect mixed Python/C/C++ code & extensions
- Accurately identify performance hotspots at line-level
- Low overhead, attach/detach to running application
- Focus your tuning efforts for most impact on performance



Auto detection & performance analysis of Python & native functions

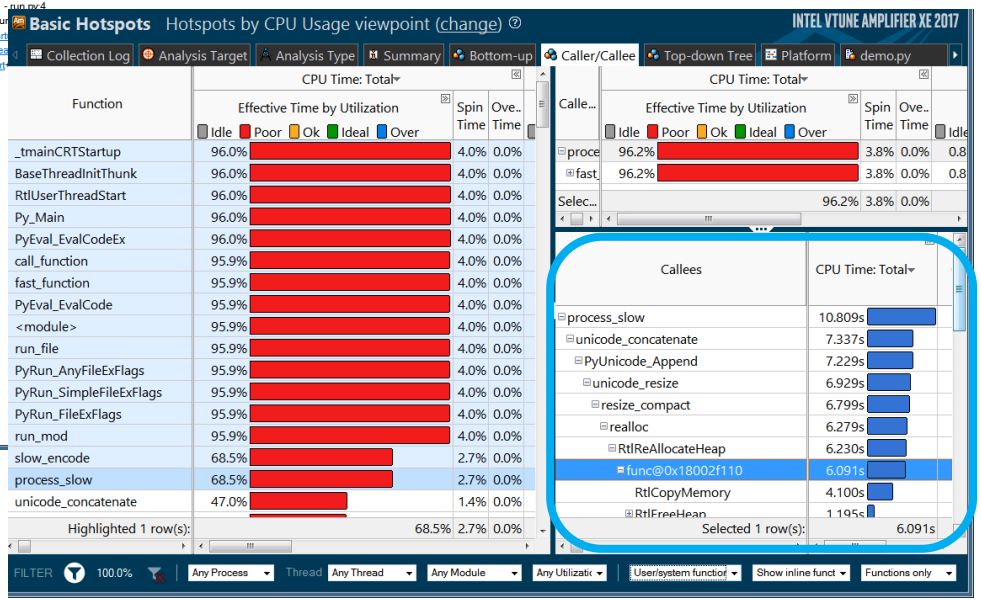
Available in Intel® VTune™ Amplifier & Intel® Parallel Studio XE

Diagnose Problem code quickly & accurately



Identifies exact line of code that is a bottleneck

Details Python* calling into native functions



A 2-prong approach for Faster Python* Performance

High Performance Python Distribution + Performance Profiling

Step 1: Use Intel® Distribution for Python

- Leverage optimized native libraries for performance
- Drop-in replacement for your current Python - no code changes required
- Optimized for multi-core and latest Intel processor

Step 2: Use **Intel® VTune™ Amplifier** for profiling

- Get detailed summary of entire application execution profile
- Auto-detects & profiles Python/C/C++ mixed code & extensions with low overhead
- Accurately detect hotspots - line level analysis helps you make smart optimization decisions fast!
- Available in Intel® Parallel Studio XE Professional & Cluster Edition

More Resources

Intel® Distribution for Python

- Product page – overview, features, FAQs…
- Training materials – movies, tech briefs, documentation, evaluation guides…
- Support – forums, secure support…

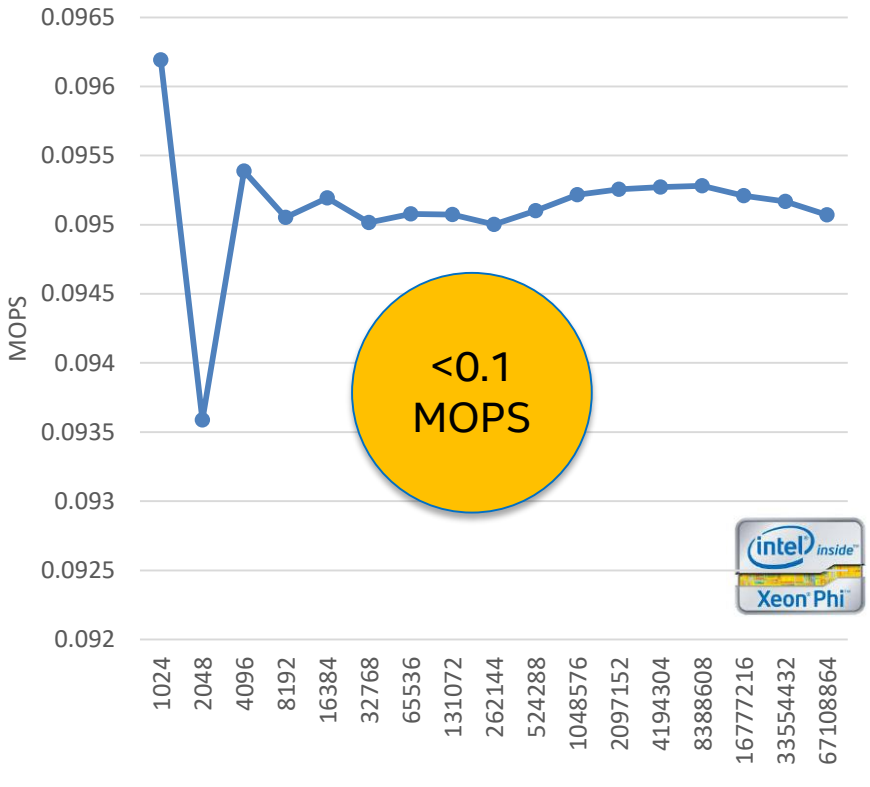
Intel® VTune Amplifier

- Product page – overview, features, FAQs…
- Training materials – movies, tech briefs, documentation, evaluation guides…
- Reviews
- Support – forums, secure support…

intel Software

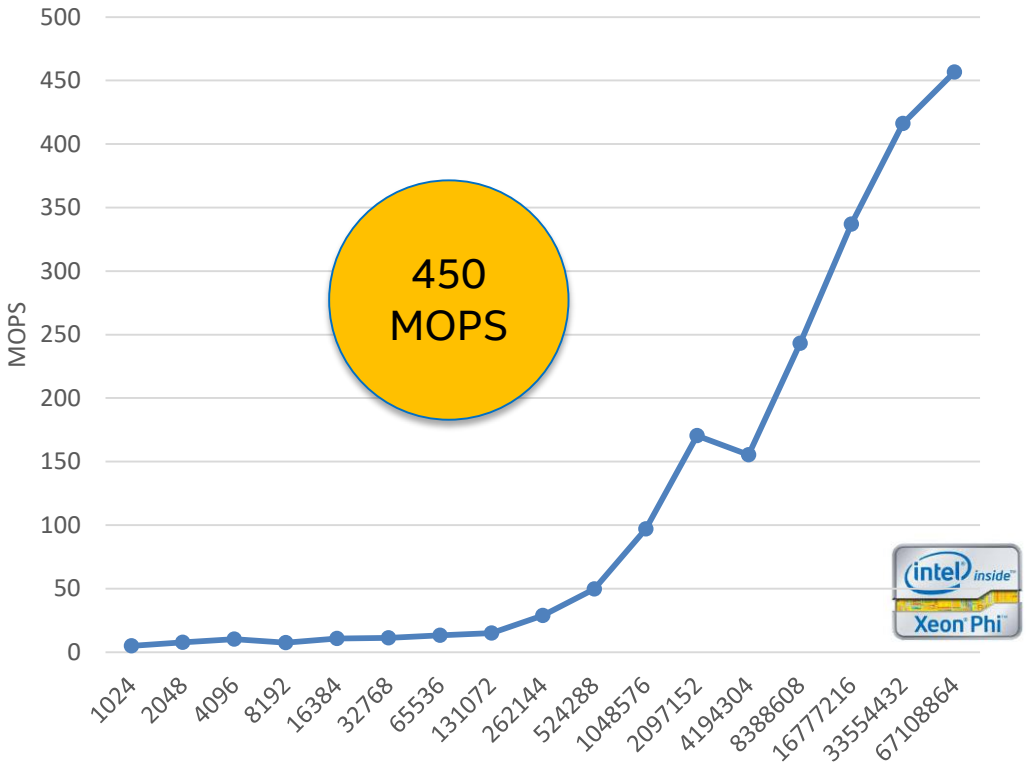
USE CASE
PERFORMANCE TOOLS APPLIED TO BLACK SCHOLES

Variant 1: Plain Python



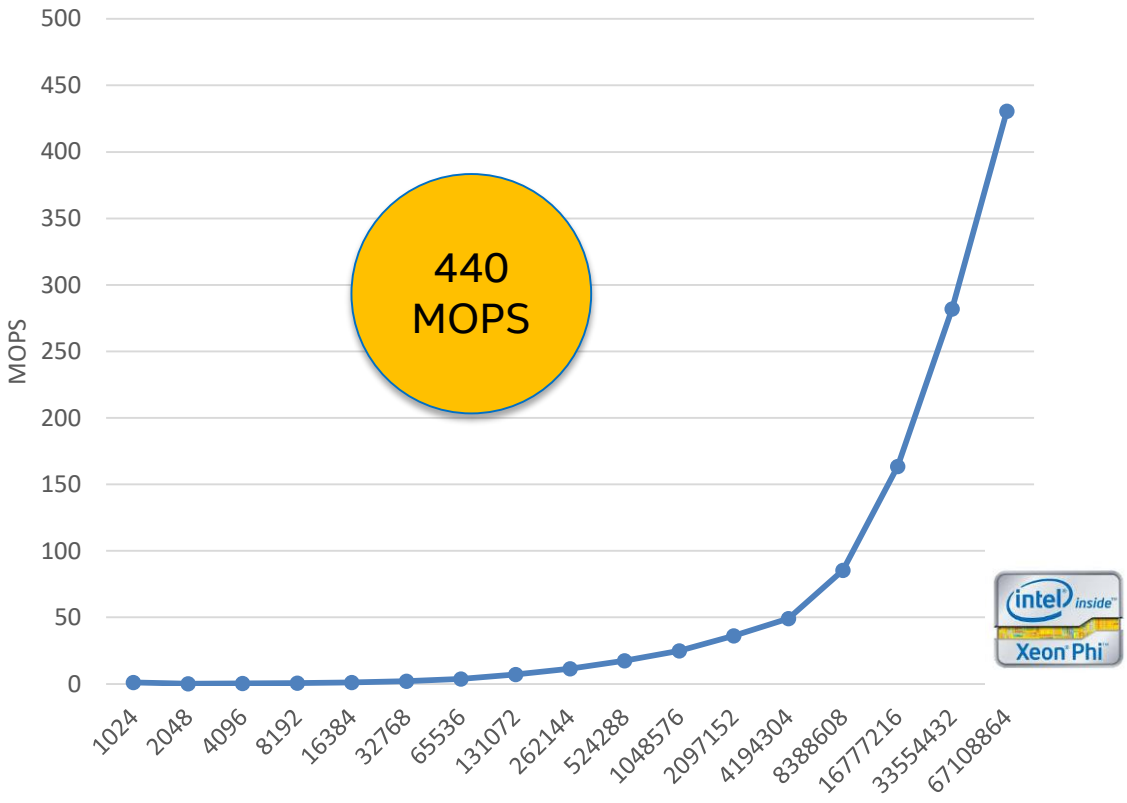

```python
def black_scholes ( nopt, price, strike, t, rate, vol, call, put ):
    mr = -rate
    sig_sig_two = vol * vol * 2

    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 = 1/sqrt(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
```

Variant 2: NumPy* arrays and Umath functions



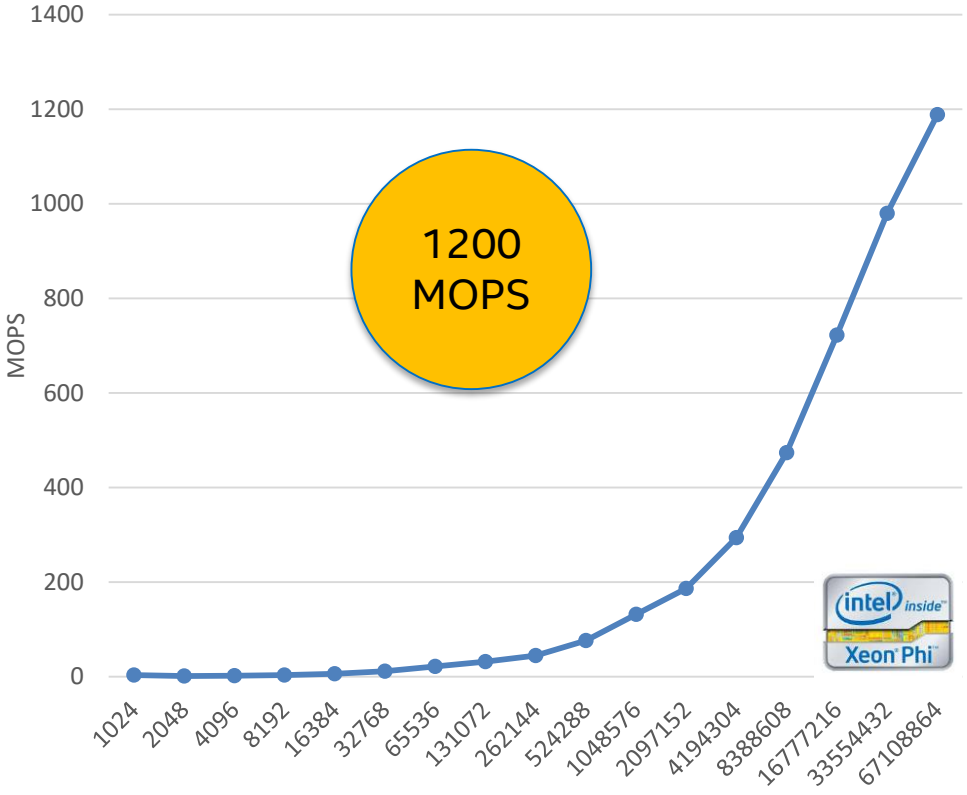

```
6  def black_scholes ( nopt, price, strike, t, rate, vol ):
7      mr = -rate
8      sig_sig_two = vol * vol * 2
9
10     P = price
11     S = strike
12     T = t
13
14     a = log(P / S)
15     b = T * mr
16
17     z = T * sig_sig_two
18     c = 0.25 * z
19     y = invsqrt(z)
20
21     w1 = (a - b + c) * y
22     w2 = (a - b - c) * y
23
24     d1 = 0.5 + 0.5 * erf(w1)
25     d2 = 0.5 + 0.5 * erf(w2)
26
27     Se = exp(b) * S
28
29     call = P * d1 - Se * d2
30     put = call - P + Se
31
32     return call, put
```

Variant 3: NumExpr* (proxy for Umath implementation)



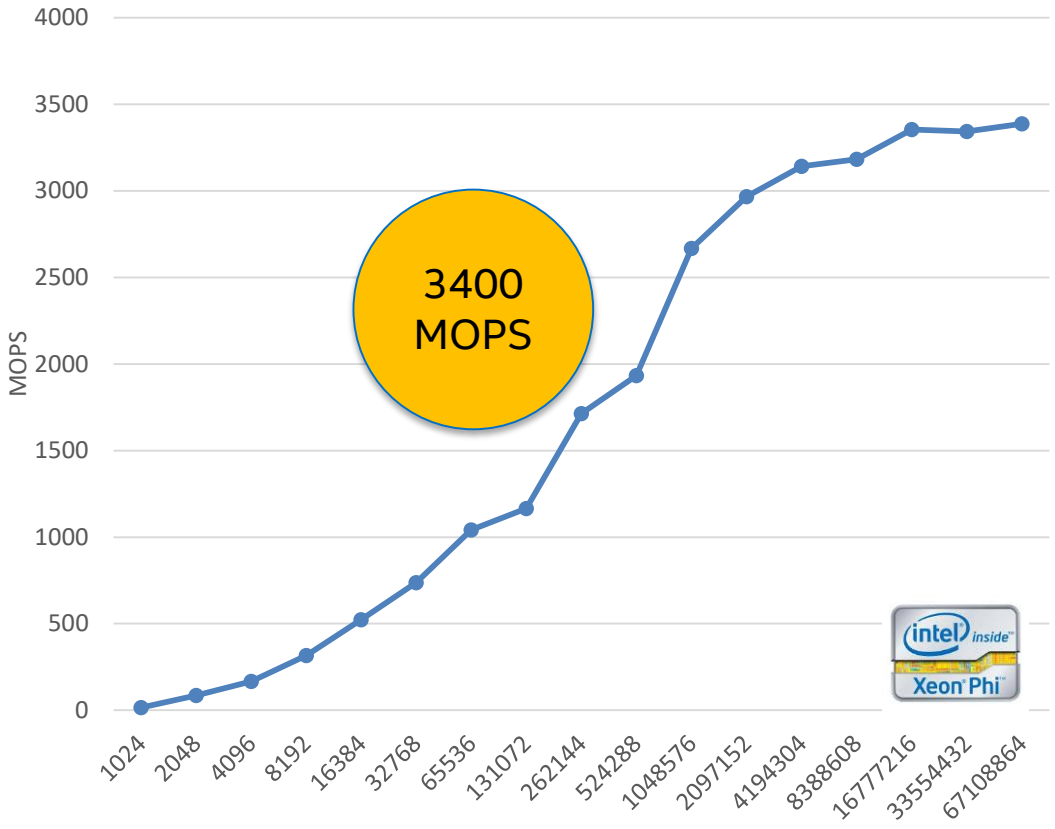

```python
import numexpr as ne

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

      P = price
      S = strike
      T = t

      a = ne.evaluate("log(P / S) ")
      b = ne.evaluate("T * mr ")

      z = ne.evaluate("T * sig_sig_two ")
      c = ne.evaluate("0.25 * z ")
      y = ne.evaluate("1/sqrt(z) ")

      w1 = ne.evaluate("(a - b + c) * y ")
      w2 = ne.evaluate("(a - b - c) * y ")

      d1 = ne.evaluate("0.5 + 0.5 * erf(w1) ")
      d2 = ne.evaluate("0.5 + 0.5 * erf(w2) ")

      Se = ne.evaluate("exp(b) * S ")

      call = ne.evaluate("P * d1 - Se * d2 ")
      put = ne.evaluate("call - P + Se ")

      return call, put

ne.set_num_threads(ne.detect_number_of_cores())
base_bs_erf.run("Numexpr", black_scholes)
```

Variant 4: NumExpr* (most performant)



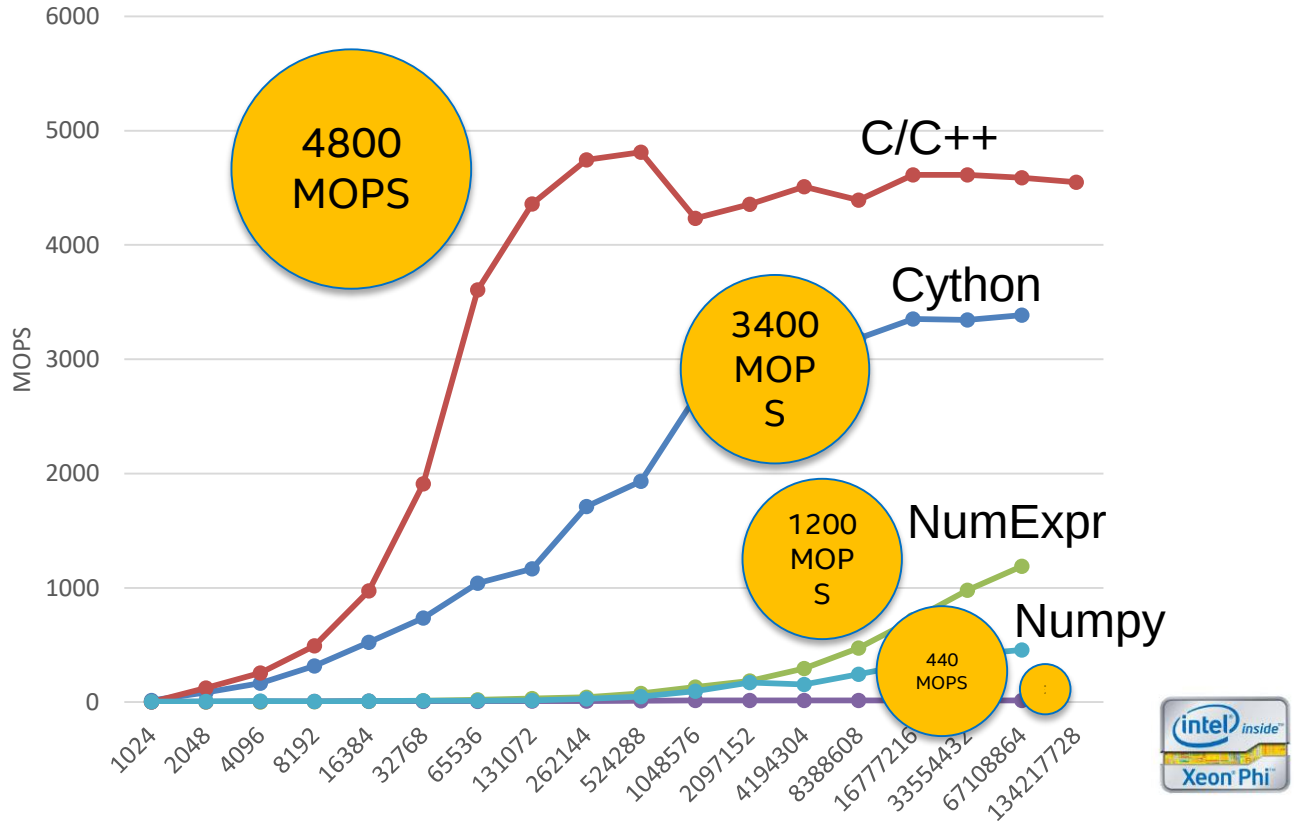

```python
import base_bs_erf
import numexpr as ne

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

    P = price
    S = strike
    T = t

    call = ne.evaluate("P * (0.5 + 0.5 * erf((log(P / S) - T * mr +" +
    "0.25 * T * sig_sig_two) * 1/sqrt(T * sig_sig_two))) - S * exp(T * mr)*" +
    "(0.5 + 0.5 * erf((log(P / S) - T * mr - 0.25 * T * sig_sig_two) *" +
    "1/sqrt(T * sig_sig_two))) ")
    put = ne.evaluate("call - P + S * exp(T * mr) ")

    return call, put
```

Variant 5: Cython*




```
# In order to release GIL for a parallel loop, the code in this block cannot
# manipulate Python objects in any way.
@boundscheck(False)
@wraparound(False)
@cdivision(True)
@initializedcheck(False)
def black_scholes(int nopt,
                  double[:] price,
                  double[:] strike,
                  double[:] t,
                  double rate,
                  double vol,
                  double[:] call,
                  double[:] put):

    cdef int i
    cdef double P, S, a, b, z, c, Se, y, T
    cdef double d1, d2, w1, w2
    cdef double mr = -rate
    cdef double sig_sig_two = vol * vol * 2

    with nogil, parallel():
        for i in prange(nopt):
            P = 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 = 1/sqrt(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
```

Variant 5: Native C/C++ vs. Python variants



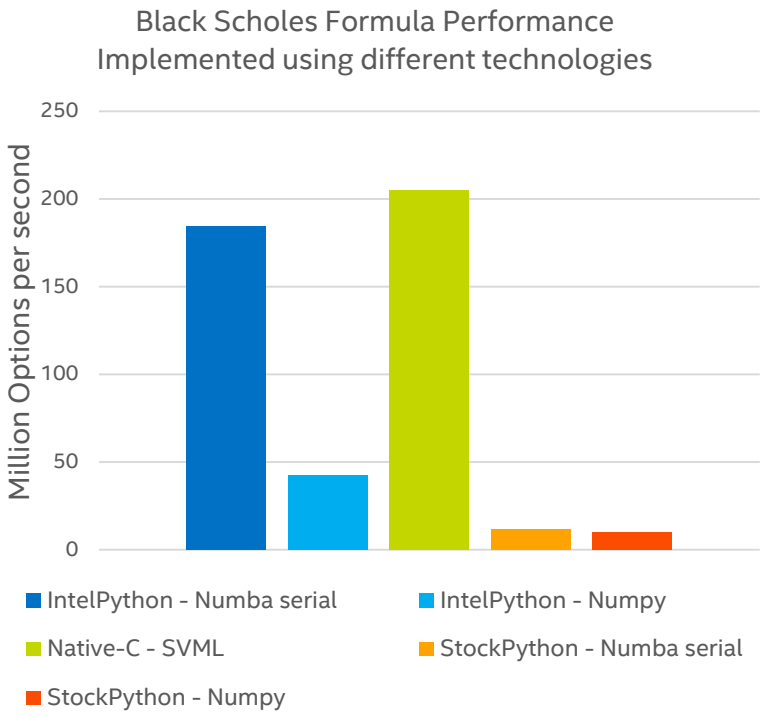
Optimizing NumPy, SciPy, NumExpr to scale

Data Analytics pipelines do not always fully match Machine Learning library functions

- Need to implement custom data transformations
- Need to provide custom optimization functions/kernels
- … And these are performance hotspots sometimes

Pure Python implementation kills performance but there are better alternatives within Python

- NumPy – array programming
- Cython – compiles Python code into native executable
- Numba – JIT compiler to accelerate performance hotspots



Configuration Info: Intel® Xeon® Gold 6140 CPU @ 2.30GHz (2 sockets, 18 cores per socket, 1 thread per core - HT is off), 256GB DDR4. Stock Python: CentOS Linux release 7.3.1611 (Core), python 3.6.4, numpy 1.14.0, numba 0.36.2, llvmlite 0.21.0. Intel® Distribution for Python 2018 Update 2 packages: python 3.6.3 intel_8, numpy 1.14.0 intel_3, numba 0.36.2 intel_2, llvmlite 0.21.1 (intel), mkl 2018.0.2 intel_1, openmp 2018.0.1 intel_0

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

intel Software

DOCKER

Docker – Some CLI basics

docker search - search images

docker pull - download images

docker images - list available images (local)

docker ps [-a] - show running containers

docker rm - delete instance

docker rmi - delete image

docker run - start container (instance) out of an image

docker start - start container

docker stop - stop container

Docker – getting started with Intel Python

Getting the Intel Python image

```
# docker pull intelpython/intelpython3_full

Using default tag: latest

latest: Pulling from intelpython/intelpython3_full


8ad8b3f87b37: Pull complete

e04db1209ac4: Pull complete

edc7ae7e687c: Pull complete

4a7b3487193b: Pull complete

c28d3c606707: Pull complete

Digest: sha256:b6720d28cca1e00fa79a4b99b2f3d58c490c7bef6c3ee95b5d249447cacb18c7

Status: Downloaded newer image for intelpython/intelpython3_full:latest
```

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

intel Software

USING CONDA

Conda

```
module load intelpython/35/2017
```

Package manager

- install/remove packages
- handles dependences
- also non-python packages (such as native libs)

Dnvironments

- isolates different sets of packages/versions
- creates hard-links when possible
- similar to virtualenv

Conda basics

Getting started

- conda --help
- conda list
- conda search numpy
- conda search numpy -c intel -c conda-forge

Environments

- conda env list
- conda create -n sandbox -c intel python=3.6
- source activate sandbox
- conda list

- Package management
  - conda install numpy
  - conda remove numpy

Preparing hands-on sessions

Linux:

- `module load intelpython/35/2017 intel/18`
- `source /cm/shared/apps/intel/vtune_amplifier_2018.2.0.551022/amplxe-vars.sh`
- `source activate idp`
- `conda create -n handson -c intel python=3.6 notebook numpy mpi4py scikit-learn matplotlib pillow numexpr numba line_profiler cython dask distributed`
- `source activate handson`

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

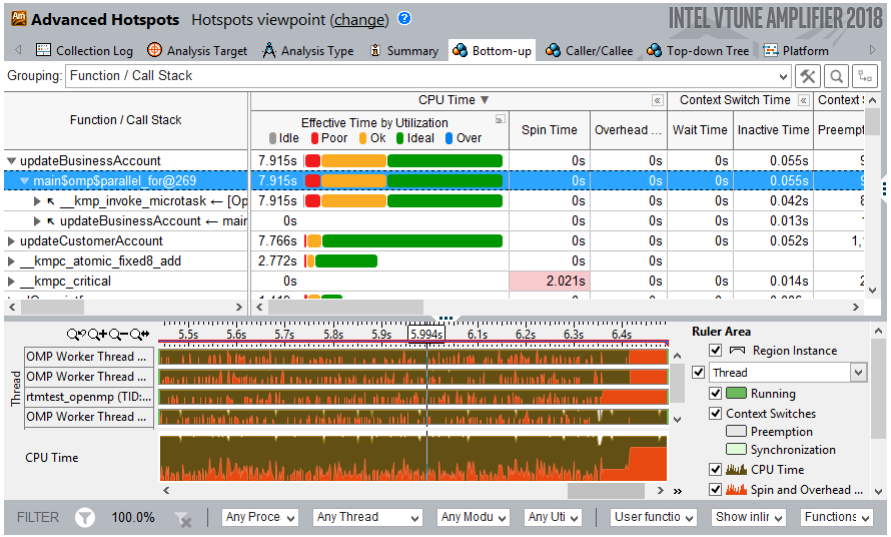
AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

INTRODUCTION TO INTEL® VTUNE™ FOR PYTHON

Intel® VTune™ Amplifier - Performance Profiler

Analyze & Tune Application Performance & Scalability



Faster, Scalable Code, Faster

- Accurately profile C, C++, Fortran*, Python*, Go*, Java*, or any mix
- Optimize CPU/GPU, threading, memory, cache, MPI, storage & more
- Save time: rich analysis leads to insight
- Take advantage of Priority Support
  - Connects customers to Intel engineers for confidential inquiries (paid versions)

New for 2018! (Partial List)

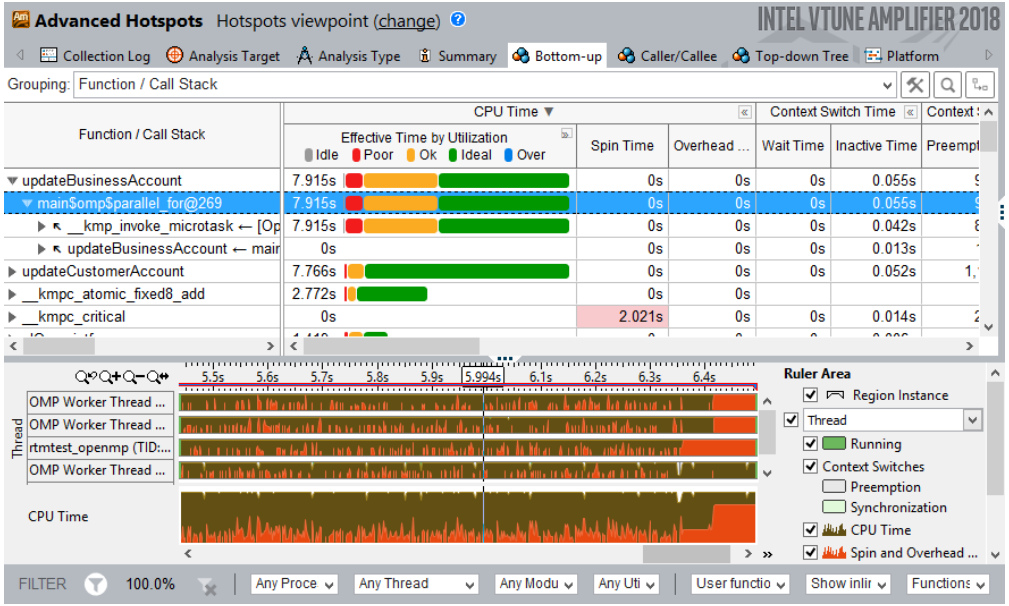
- Better metrics for threaded MPI apps
- OpenCL™ kernel hotspot analysis
- Cross-OS analysis – e.g. analyze Linux* from Windows* or macOS*
- Profile inside containers

Intel® VTune™ Amplifier

Tune Applications for Scalable Multicore Performance

Agenda

- Data Collection –
  Rich set of performance data
- Data Analysis -
  Find answers fast
- Python!



Two Great Ways to Collect Data

Intel® VTune™ Amplifier

Software Collector	Hardware Collector
Uses OS interrupts	Uses the on chip Performance Monitoring Unit (PMU)
Collects from a single process tree	Collect system wide or from a single process tree.
~10ms default resolution	~1ms default resolution (finer granularity - finds small functions)
Either an Intel® or a compatible processor	Requires a genuine Intel® processor for collection
Call stacks show calling sequence	Optionally collect call stacks
Works in virtual environments	Works in a VM only when supported by the VM (e.g., vSphere*, KVM)
No driver required	Requires a driver - Easy to install on Windows - Linux requires root (or use default perf driver)

No special recompiles - C, C++, C#, Fortran, Java, Assembly

A Rich Set of Performance Data

Intel® VTune™ Amplifier

Software Collector	Hardware Collector
Basic Hotspots Which functions use the most time?	**Advanced Hotspots** Which functions use the most time? Where to inline? – Statistical call counts
Concurrency Tune parallelism. Colors show number of cores used.	**General Exploration** Where is the biggest opportunity? Cache misses? Branch mispredictions?
Locks and Waits Tune the #1 cause of slow threaded performance: – waiting with idle cores.	**Advanced Analysis** Dig deep to tune access contention, etc.
Any IA86 processor, any VM, no driver	Higher res., lower overhead, system wide

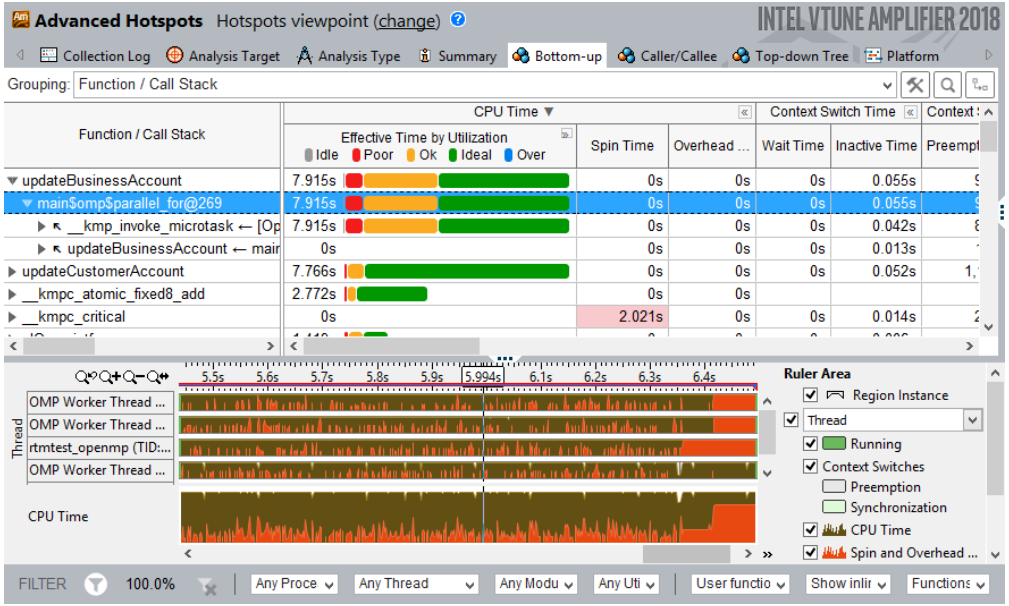
No special recompiles - C, C++, C#, Fortran, Java, Assembly

Intel® VTune™ Amplifier

Tune Applications for Scalable Multicore Performance

Agenda

- Data Collection –
  Rich set of performance data
- Data Analysis -
  Find answers fast
- Python



Find Answers Fast

Intel® VTune™ Amplifier

Adjust Data Grouping

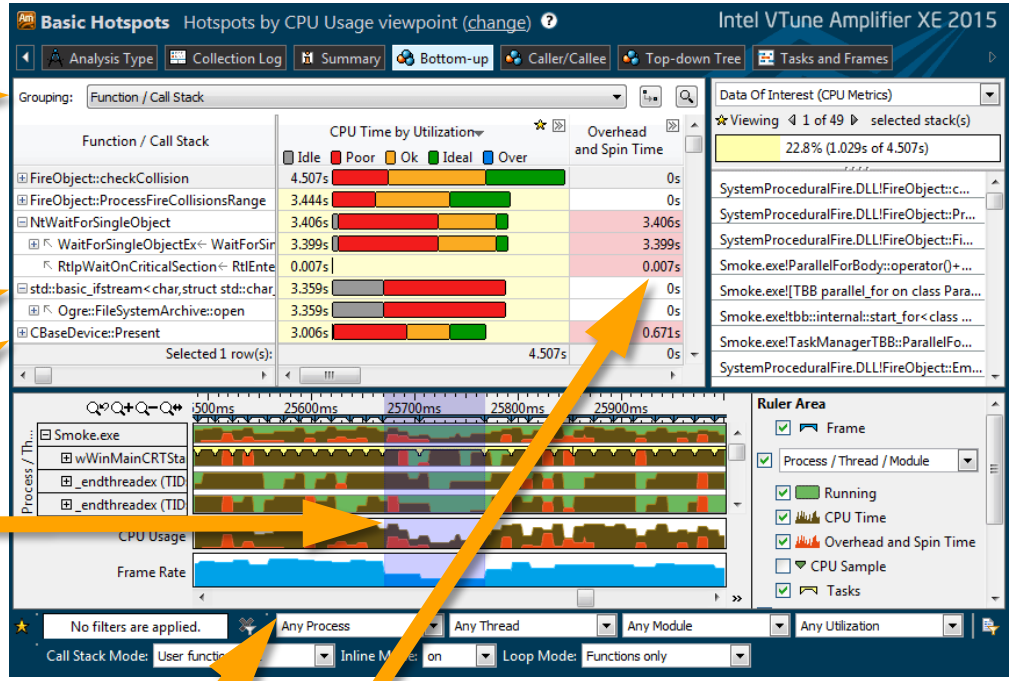
Function - Call Stack
Module - Function - Call Stack
Source File - Function - Call Stack
Thread - Function - Call Stack
... (Partial list shown)

Double Click Function to View Source

Click [+] for Call Stack

Filter by Timeline Selection (or by Grid Selection)

Zoom In And Filter On Selection
Filter In by Selection
Remove All Filters



Filter by Process & Other Controls

Tuning Opportunities Shown in Pink. Hover for Tips

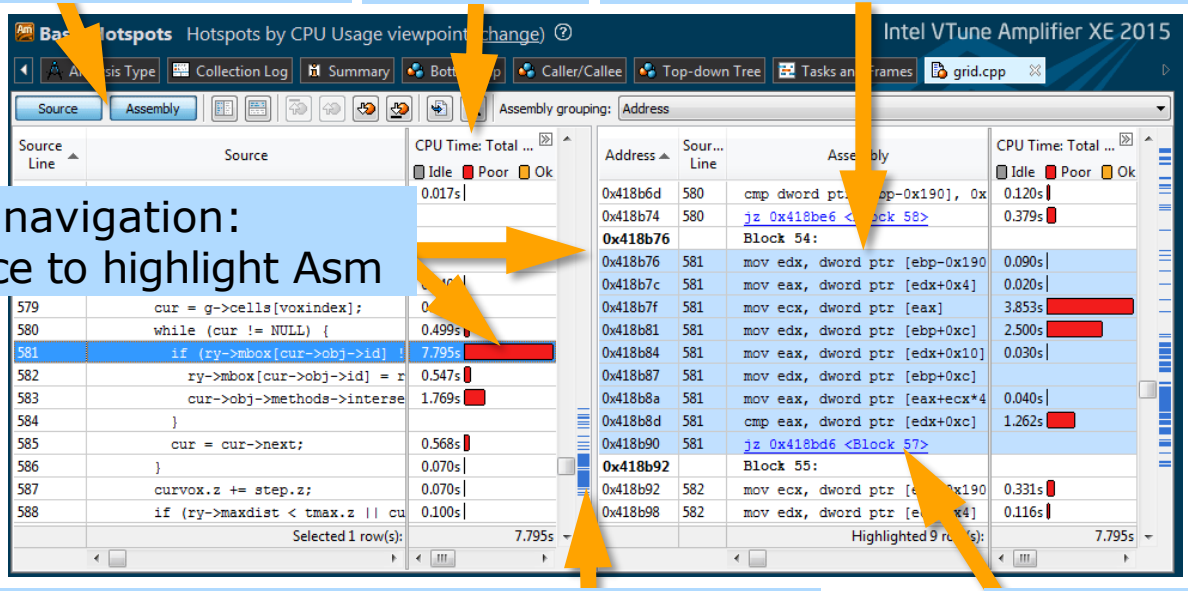
See Profile Data On Source / Asm

Double Click from Grid or Timeline

View Source / Asm or both

CPU Time

Right click for instruction reference manual



Quick Asm navigation:
Select source to highlight Asm

Scroll Bar “Heat Map” is an overview of hot spots

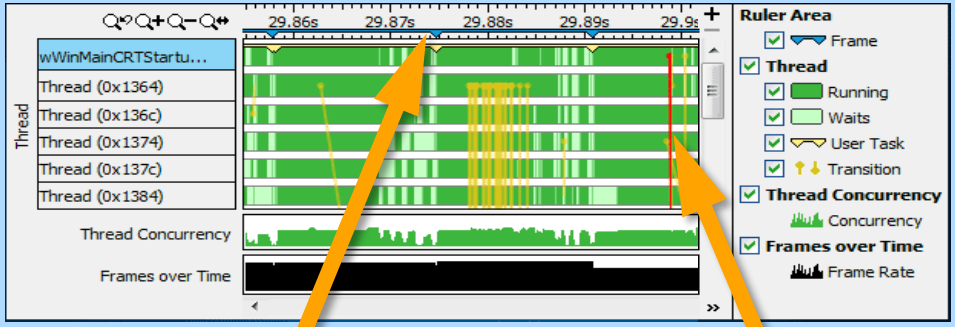
Click jump to scroll Asm

Timeline Visualizes Thread Behavior

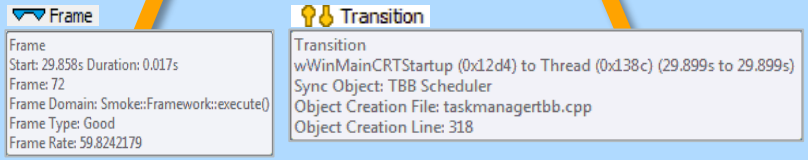
Intel® VTune™ Amplifier

Transitions

Locks & Waits

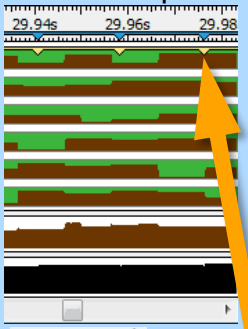


Hovers:

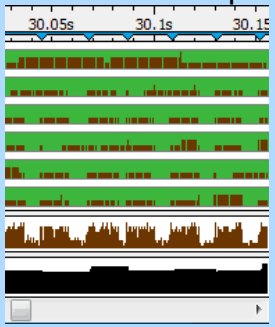


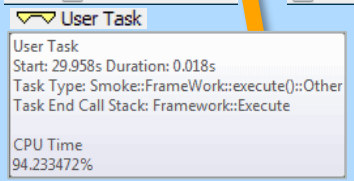
CPU Time

Basic Hotspots



Advanced Hotspots





Optional: Use API to mark frames and user tasks 

Optional: Add a mark during collection 

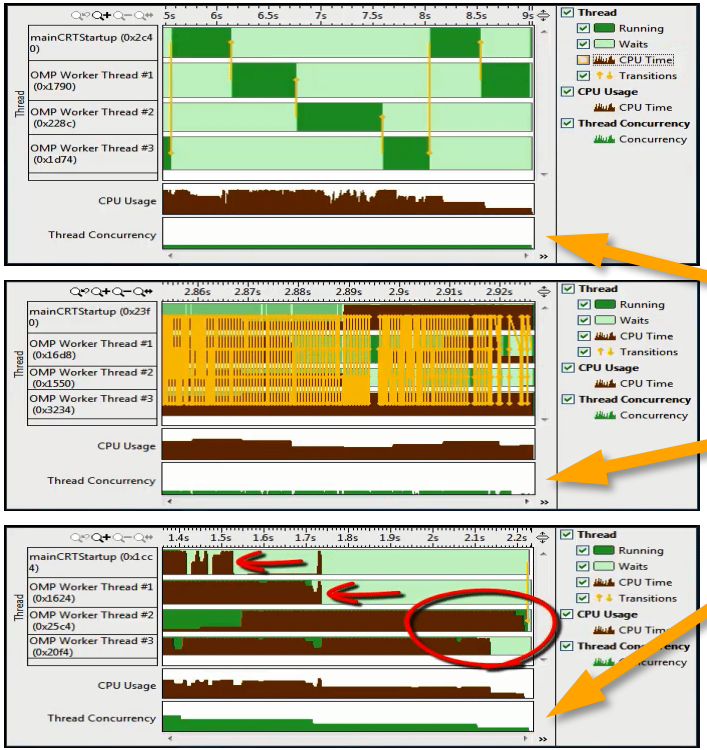
Visualize Parallel Performance Issues

Look for Common Patterns

Coarse Grain Locks

High Lock Contention

Load Imbalance



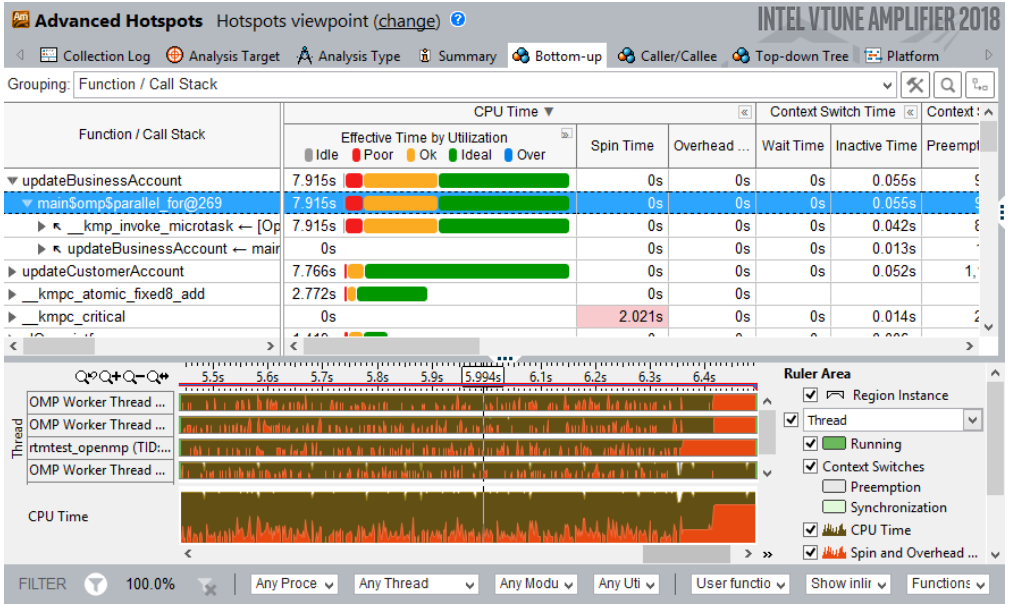
Low Concurrency

Intel® VTune™ Amplifier

Tune Applications for Scalable Multicore Performance

Agenda

- Data Collection –
  Rich set of performance data
- Data Analysis -
  Find answers fast
- Python!



Tune Python* + Native Code for Better Performance

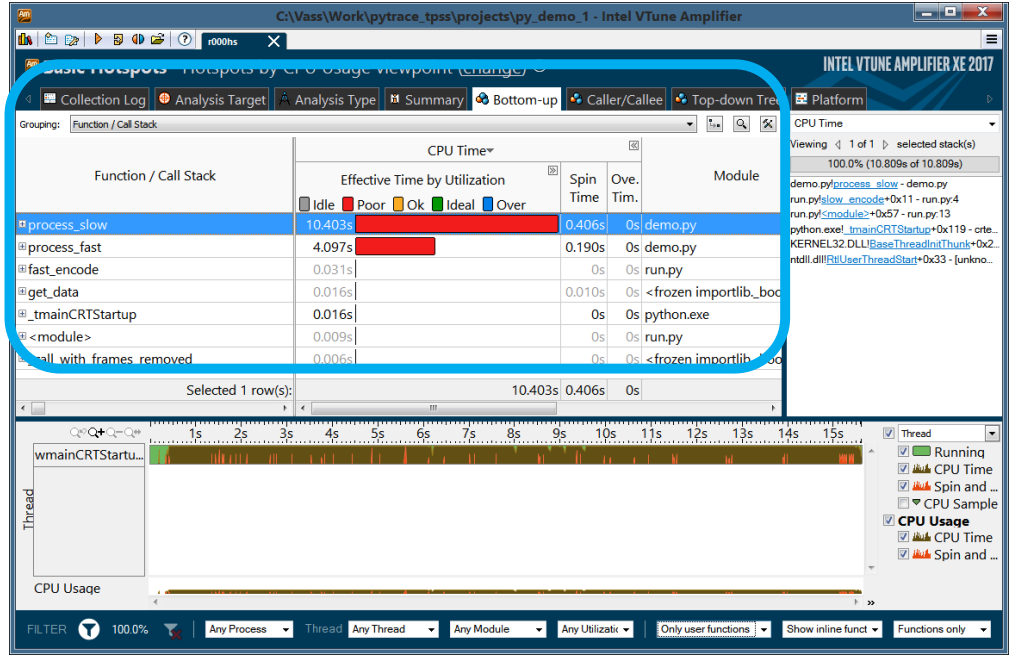
Analyze Performance with Intel® VTune™ Amplifier (available in Intel® Parallel Studio XE)

Challenge

- Single tool that profiles Python + native mixed code applications
- Detection of inefficient runtime execution

Solution

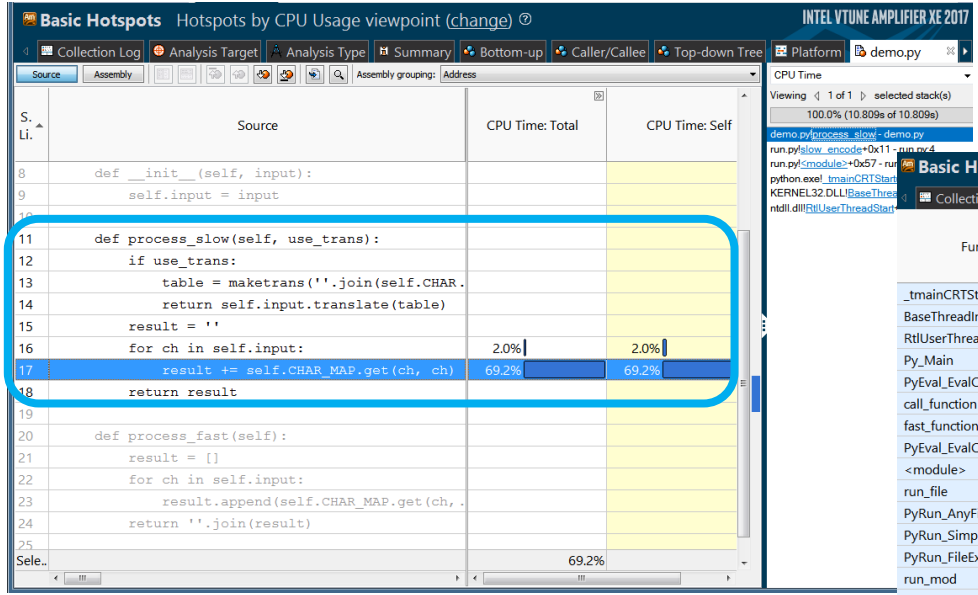
- Auto-detect mixed Python/C/C++ code & extensions
- Accurately identify performance hotspots at line-level
- Low overhead, attach/detach to running application
- Focus your tuning efforts for most impact on performance



Auto detection & performance analysis of Python & native functions

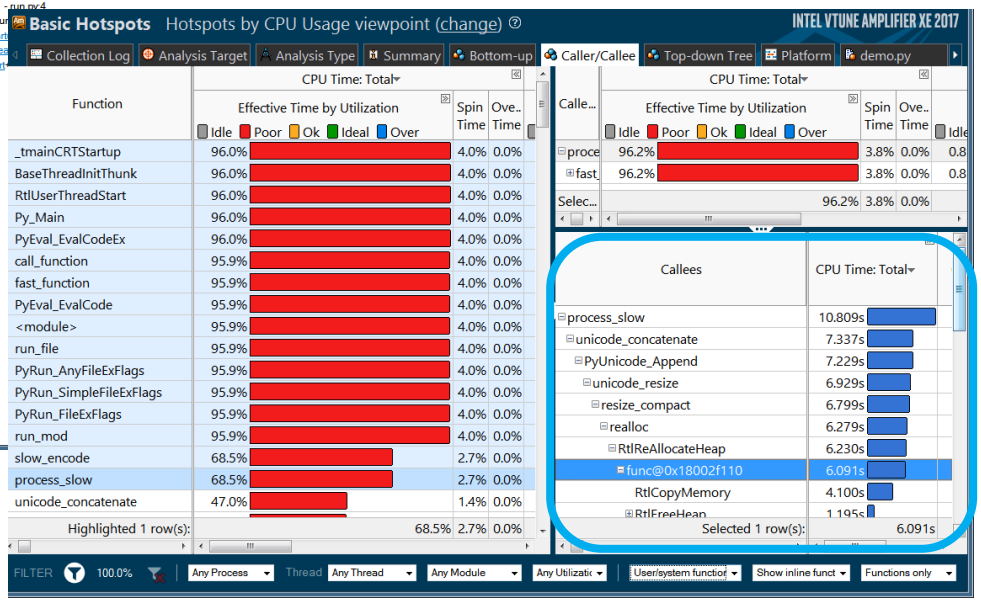
Available in Intel® VTune™ Amplifier & Intel® Parallel Studio XE

Diagnose Problem code quickly & accurately

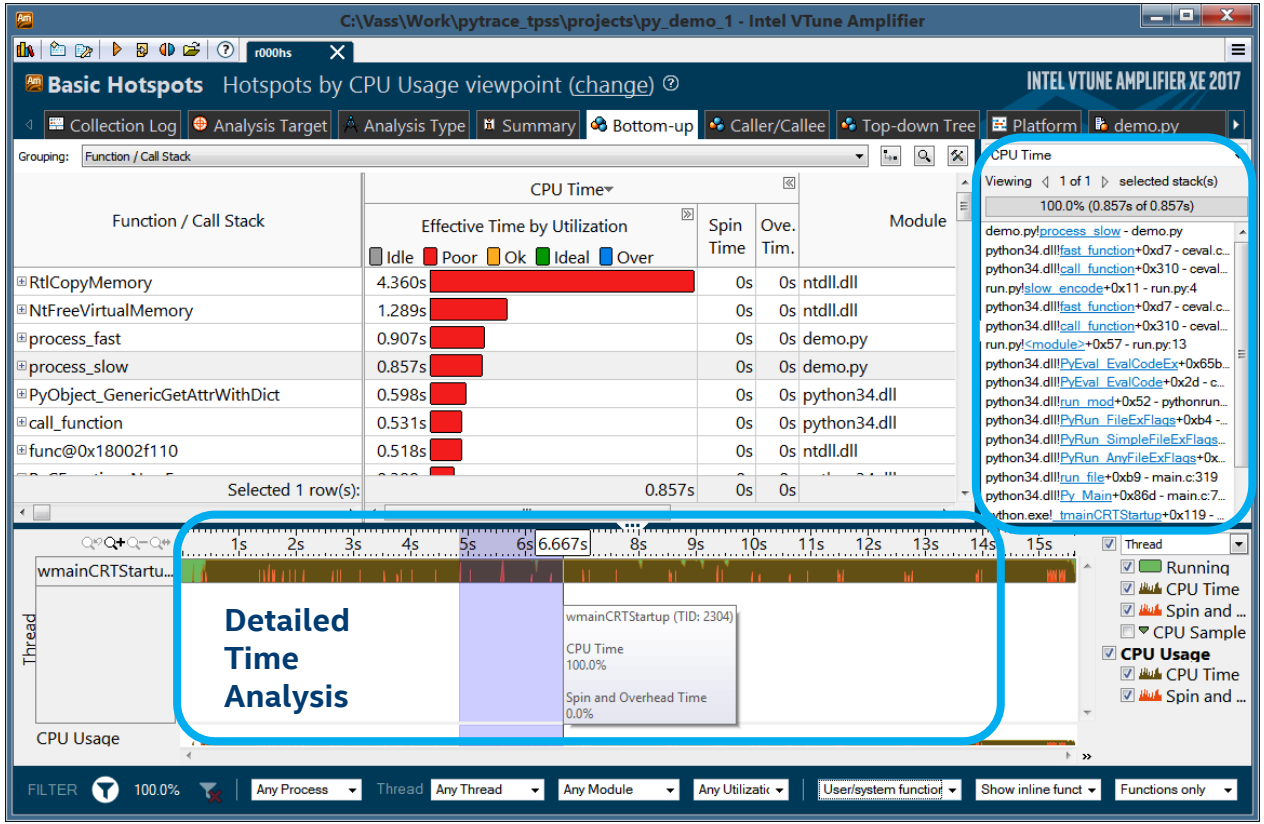


Identifies exact line of code that is a bottleneck

Details Python* calling into native functions



Deeper Analysis for Better Insight



Call Stack Listing for Python* & Native Code

A 2-prong approach for Faster Python* Performance

High Performance Python Distribution + Performance Profiling

Step 1: Use Intel® Distribution for Python

- Leverage optimized native libraries for performance
- Drop-in replacement for your current Python - no code changes required
- Optimized for multi-core and latest Intel processors

Step 2: Use **Intel® VTune™ Amplifier** for profiling

- Get detailed summary of entire application execution profile
- Auto-detects & profiles Python/C/C++ mixed code & extensions with low overhead
- Accurately detect hotspots - line level analysis helps you make smart optimization decisions fast!
- Available in Intel® Parallel Studio XE Professional & Cluster Edition

VTUNE

VTune

Demo

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

intel Software

HANDS-ON PYTHON PERFORMANCE TOOLS

Preparing hands-on sessions

```
# source activate handson

python -V

python -c "import sklearn, numba, daal, numexpr, mpi4py"

which amplxe-cl && which amplxe-gui && which icc && which
mpirun

cd <material>

jupyter notebook
```

Notebooks

interactive

python, markup, and shell (and other)

python kernel running

cells get executed individually

Using parallelism under the hood

Similar syntax
- Vector based (no loop)
- List -> np.array
- Import from numpy

Numpy

Mathematical building blocks

Universal functions (ufuncs) operate on vectors

Under the hood accelerates computation (e.g. with Intel® MKL)

NumExpr

Hands-On Numpy

Black Scholes option pricing: 01_numpy_blackscholes.ipynb

Numpy

Naïve '[]' are lists, not arrays

Numpy provides fast array implementation

- contiguous memory
- written in C

Numpy also provides optimized operations on arrays

- vector/array/scalar operations allows calling vectorized code
- Umath dispatches to right vector/array/scalar implementation

- Intel-optimized Numpy is accelerated with Intel® Math Kernel Library
- Also accelerates FFT, RNG, Umath, …

Using parallelism under the hood

Numpy

NumExpr

- Evaluates strings of numeric expressions
- Internally translates to numpy ufuncs

Hands-On numexpr

Black Scholes option pricing: 02_numexpr_blackscholes.ipynb

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

Compiling Python

Numba

- Just-In-Time (LLVM-based, produces parallel code)
- Decorators annotate functions
- JIT API

Cython

Hands-On numexpr

Black Scholes option pricing: 03_numba_blackscholes.ipynb

Compiling Python - Numba

Decorators

Defining instantiations

GIL, nopython

API and target

Compiling Python - Cython

Numba

Cython

Ahead-Of-Time

- produces C code which can be vectorized and parallel

Hybrid language Python and C

For creating python package to be loaded

Hands-On Cython

Black Scholes option pricing: 04_cython_blackscholes.ipynb

Compiling Python - Cython

Import C functions

Annotations

- optimization decorators (gil, checks etc)
- types

target, nogil

AGENDA

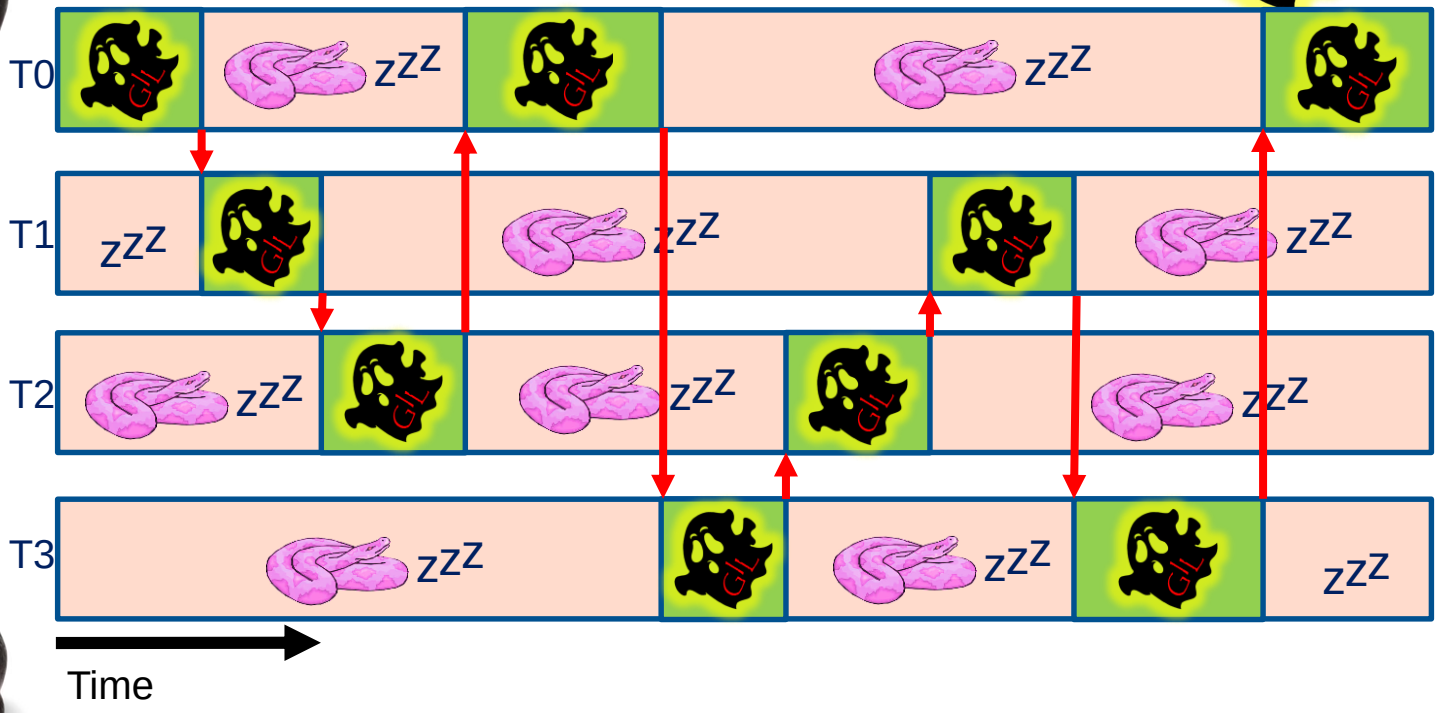
10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

intel Software

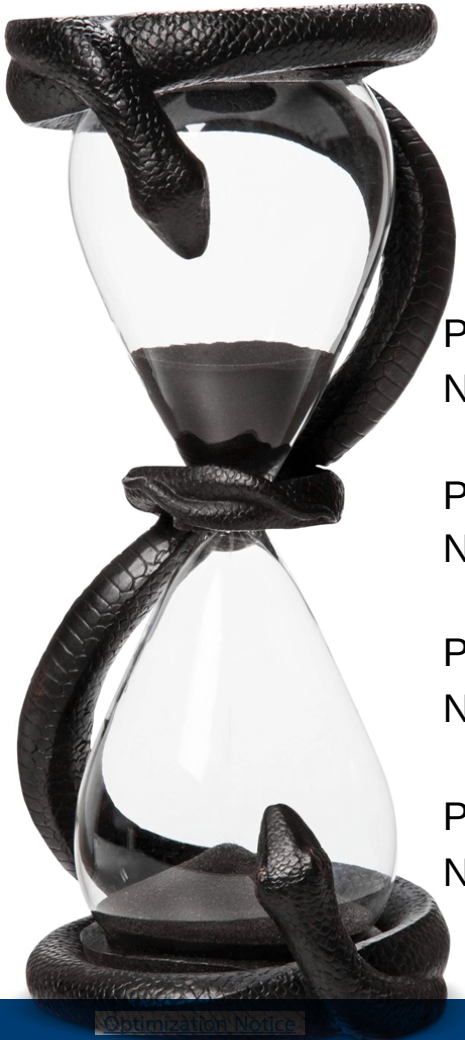
HANDS-ON PYHTON EXPLICIT PARALLELISM

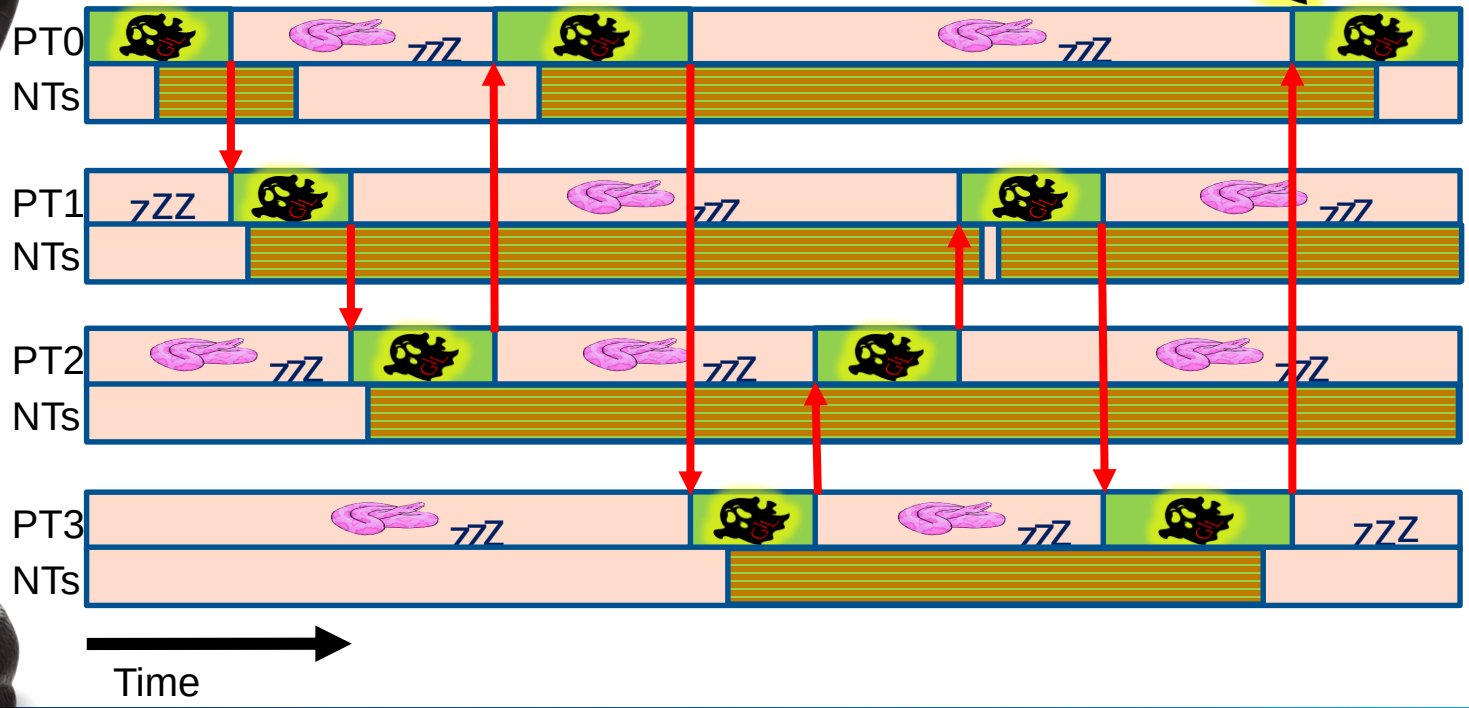
Global Interpreter Lock

The GIL



Releasing the GIL





How to tame the GIL?

Call out to native extensions (releasing the GIL)

- Limited efficiency by Amdahl's law
  - Think of it as if all python code is executed on a single thread/core
- 3rd party, home-brewed, …

Use multi-processing when possible

- Data/computation ratio
- Memory footprint

Use compilers

Use PyPy, Jython, IronPython, …

- Parallelism not necessarily their primary concern, though

Parallelism in Python - MPI

MPI

- Wrappers around (Intel®) MPI C library
- Simpler API than MPI standard
- Uses pickle for marshalling

Python's multiprocessing

Dask

Others

Hands-On mpi4py

Black Scholes option pricing: 05_mpi4py_blackscholes.ipynb

Parallelism in Python - MPI

Kernel: fan-out & compute & fan-in

Opportunity for proper SPMD

Hands-On dask

Black Scholes option pricing: 06_dask_blackscholes.ipynb

Hands-On Threading

Black Scholes option pricing: 07_apply_blackscholes.ipynb

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

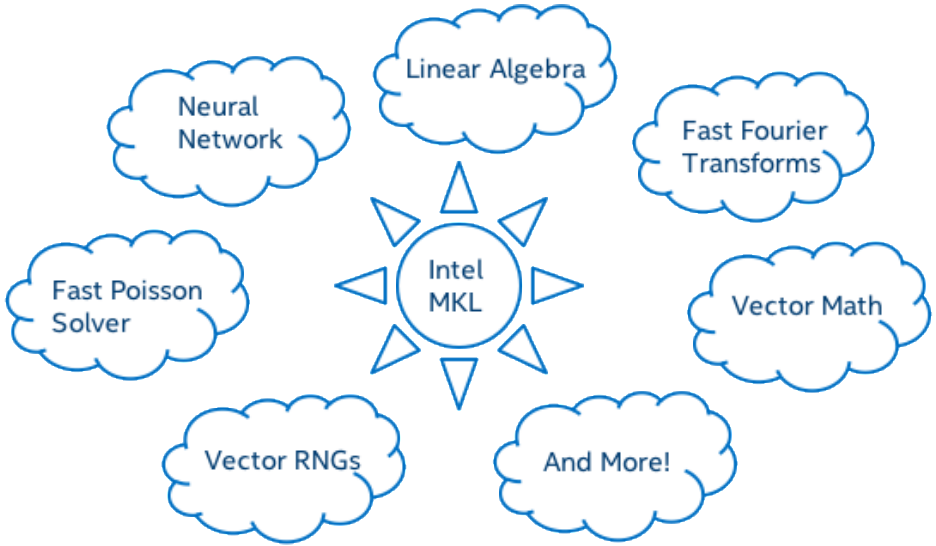
AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

intel Software

INTEL® MATH KERNEL LIBRARY
MKL

Faster, Scalable Code with Intel® Math Kernel Library



- 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

- What's New in Intel® MKL 2018
- Improved small matrix multiplication performance in GEMM and LAPACK
- Improved ScaLAPACK performance for distributed computation
- 24 new vector math functions
- Simplified license for easier adoption and redistribution
- Additional distributions via YUM, APT-GET, and Conda repositories

Learn More: software.intel.com/mkl

What's Inside Intel® MKL

Accelerate HPC, Enterprise, IoT & Cloud Applications

Linear Algebra
- BLAS
- LAPACK
- ScaLAPACK
- Sparse BLAS
- Iterative sparse solvers
- PARDISO*
- Cluster Sparse Solver

FFTs
- Multidimensional
- FFTW interfaces
- Cluster FFT

Neural Networks
- Convolution
- Pooling
- Normalization
- ReLU
- Inner Product

Vector RNGs
- Congruential
- Wichmann-Hill
- Mersenne Twister
- Sobol
- Neiderreiter
- Non-deterministic

Summary Statistics
- Kurtosis
- Variation coefficient
- Order statistics
- Min/max
- Variance-covariance

Vector Math
- Trigonometric
- Hyperbolic
- Exponential
- Log
- Power
- Root

And More
- Splines
- Interpolation
- Trust Region
- Fast Poisson Solver

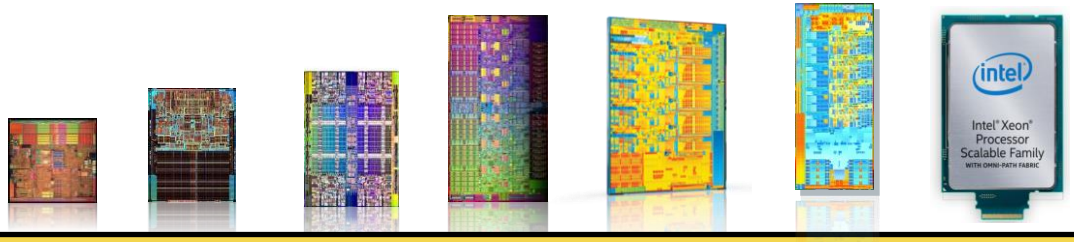
Intel® Architecture Platforms

Operating System: Windows*, Linux*, MacOS[1]*

Available only in Intel® Parallel Studio Composer Edition.

Automatic Dispatching to Tuned ISA-specific Code Paths

More cores → More Threads → Wider vectors



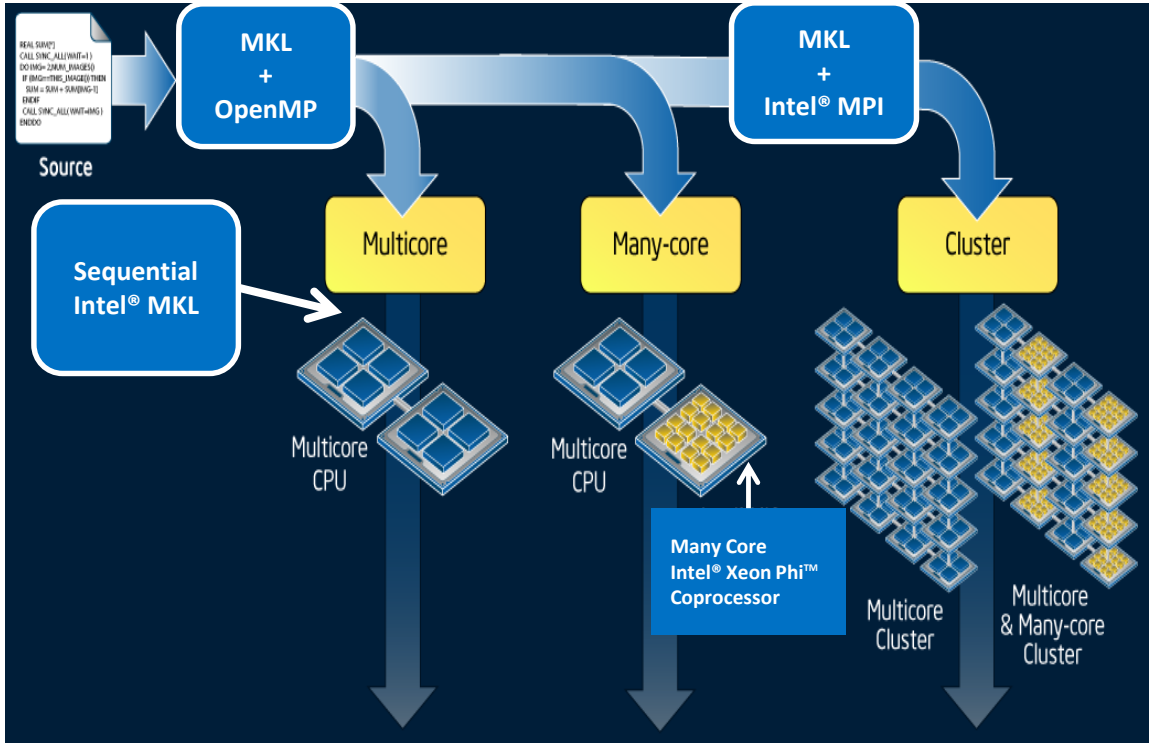
	Intel® Xeon® Processor 64-bit	Intel® Xeon® Processor 5100 series	Intel® Xeon® Processor 5500 series	Intel® Xeon® Processor 5600 series	Intel® Xeon® Processor E5-2600 v2 series	Intel® Xeon® Processor E5-2600 v3 series v4 series	Intel® Xeon® Scalable Processor[1]	Intel® Xeon Phi™ x200 Processor (KNL)
Up to Core(s)	1	2	4	6	12	18-22	28	72
Up to Threads	2	2	8	12	24	36-44	56	288
SIMD Width	128	128	128	128	256	256	512	512
Vector ISA	Intel® SSE3	Intel® SSE3	Intel® SSE4-4.1	Intel® SSE 4.2	Intel® AVX	Intel® AVX2	Intel® AVX-512	Intel® AVX-512

1. Product specification for launched and shipped products available on ark.intel.com.

AUTOMATIC PERFORMANCE SCALING FROM THE CORE, TO MULTICORE, TO MANY CORE AND BEYOND
INTEL® MKL

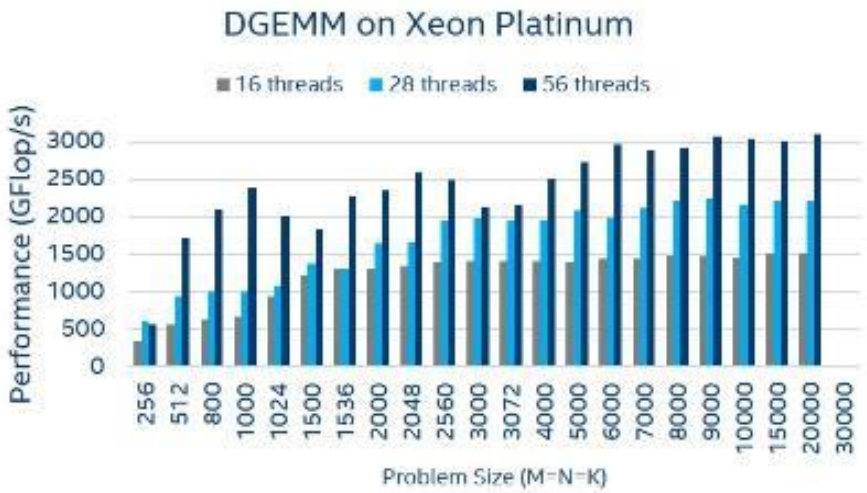
Extracting performance from the computing resources

- Core: **vectorization**, prefetching, cache utilization
- Multi-Many core (processor/socket) level **parallelization**
- Multi-socket (node) level **parallelization**
- Clusters **scaling**



Performance Benefits for the latest Intel Architectures

DGEMM, SGEMM Optimized by Intel® Math Kernel Library for Intel® Xeon® Platinum Processor
(formerly codenamed Skylake Server)



Configuration: Intel® Xeon® Platinum 8180, 2x28 cores, 2.5GHz, 38.5MB L3 cache, 376GB RAM, OS Ubuntu 16.04 LTS; Intel® Math Kernel Library (Intel® MKL) 2018. 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 visit www.intel.com/benchmarks. Benchmark Source: Intel Corporation. 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.

INTEL® MKL 2018 NEW FEATURES AND OPTIMIZATIONS

Intel® Xeon Phi™ Knights Mill Optimizations	DNN Convolution and Inner Product functions Single Precision Level 3 BLAS, including SGEMM New INT8 and INT16 GEMM functions
BLAS and LAPACK	Compact BLAS and LAPACK functions Direct Call LAPACK Cholesky and QR factorizations LU factorization and Inverse without pivoting Aasen-based factorization and solve functions Bounded Bunch-Kaufman (Rook) pivoting factorizations
FFTs	Verbose Mode Support
Vector Math - 24 New Functions	v?Fmod, v?Remainder v?Powr, v?Exp2, v?Exp10 v?Cospi, v?Sinpi, v?Tanpi, and more

INTEL® MKL 11.0 – 2017 NOTEWORTHY ENHANCEMENTS

- Conditional Numerical Reproducibility (CNR)
- Intel Threading Building Blocks (TBB) Composability
- Intel® Optimized High Performance Conjugate Gradient (HPCG) Benchmark
- Small GEMM Enhancements (Direct Call) and Batch
- Sparse BLAS Inspector-Executor API
- Extended Cluster Support (MPI wrappers and macOS)
- Parallel Direct Sparse Solver for Clusters
- Extended Eigensolvers
- Deep Neural Networks Convolution, Normalization, Activation, and Pooling Primitives

INTEL® MKL SUMMARY

Boosts application performance with minimal effort	feature set is robust and growing provides scaling from the core, to multicore, to manycore, and to clusters automatic dispatching matches the executed code to the underlying processor future processor optimizations included well before processors ship
Showcases the world's fastest supercomputers[1]	Intel® Distribution for LINPACK* Benchmark Intel® Optimized High Performance Conjugate Gradient Benchmark

[1]http://www.top500.org

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

intel Software

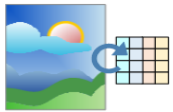
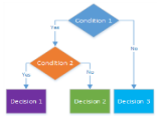
INTEL® DATA ANALYTICS AND ACCELERATION LIBRARY DAAL

Faster Machine Learning & Analytics with Intel® DAAL

- Features highly tuned functions for classical machine learning and analytics performance across spectrum of Intel® architecture devices
- Optimizes data ingestion together with algorithmic computation for highest analytics throughput
- Includes Python*, C++, and Java* APIs and connectors to popular data sources including Spark* and Hadoop*
- Free and open source community-supported versions are available, as well as paid versions that include premium support.

What's New in 2018 version

- New Algorithms:
  - Classification & Regression Decision Tree
  - Classification & Regression Decision Forest
  - k-NN
  - Ridge Regression
- Spark* MLlib-compatible API wrappers for easy substitution of faster Intel DAAL functions
- Improved APIs for ease of use
- Repository distribution via YUM, APT-GET, and Conda

Pre-processing	Transformation	Analysis	Modeling	Validation	Decision Making
					
Decompression, Filtering, Normalization	Aggregation, Dimension Reduction	Summary Statistics Clustering, etc.	Machine Learning (Training) Parameter Estimation Simulation	Hypothesis testing Model errors	Forecasting Decision Trees, etc.

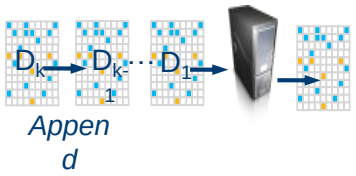
Learn More: software.intel.com/daal

INTEL DAAL PERFORMANCE SCALING

- Within a CPU core:
  - SIMD vectorization: optimized for the latest instruction sets, AVX2, AVX512…
  - Internally relies on sequential MKL
- Scale to multi cores or many cores:
  - Intel TBB threading
- Scale to cluster:
  - Distributed processing done by user application (MPI, MapReduce, etc.)
  - DAAL provides
    - Data structures for partial and intermediate results
    - Functions to combine partial or intermediate results into global result

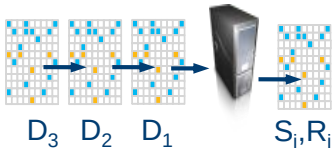
Processing modes

Batch Processing



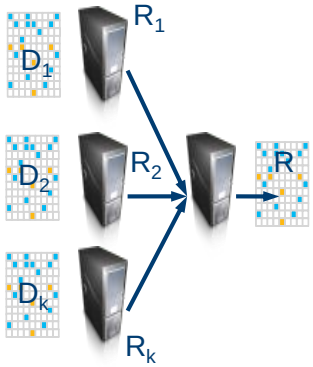
$R = F(D_1,\ldots,D_k)$

Online Processing



$S_{i+1} = T(S_i,D_i)$
$R_{i+1} = F(S_{i+1})$

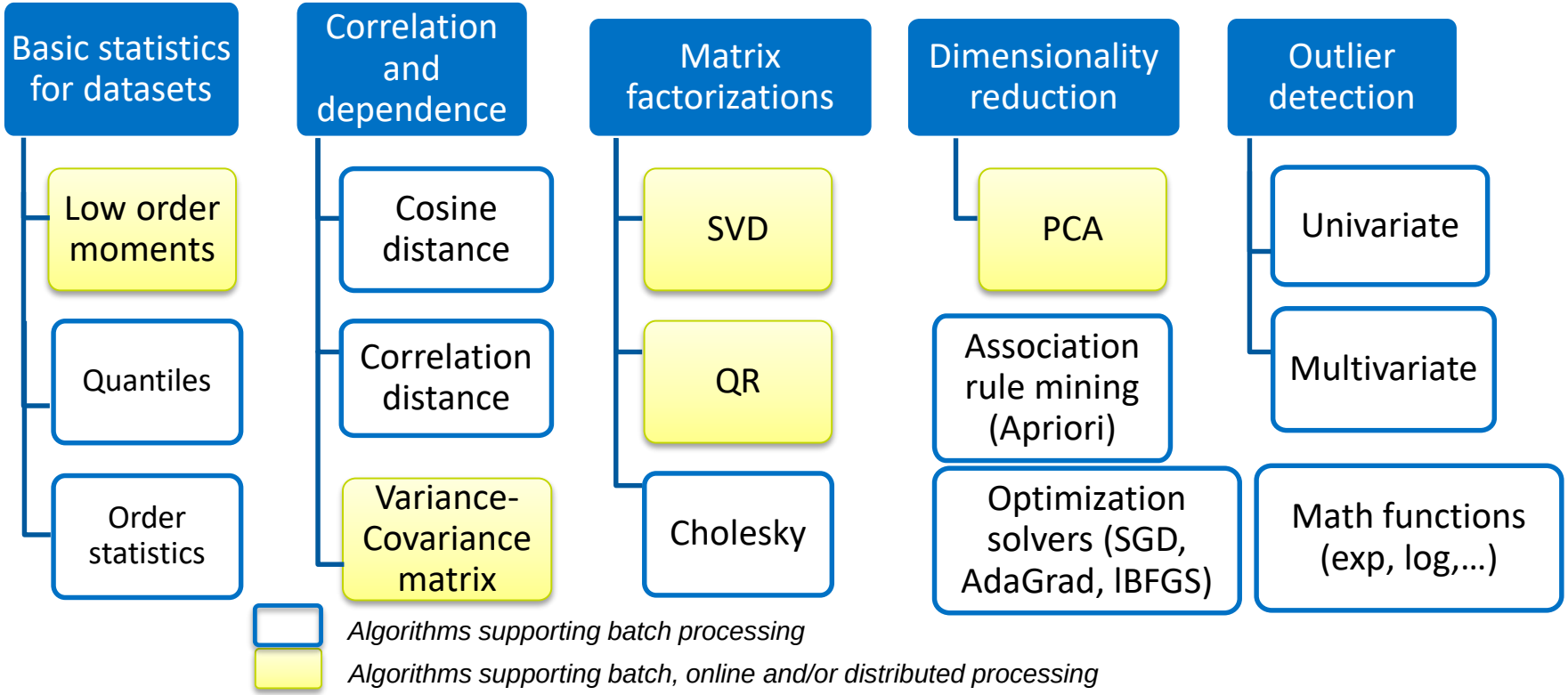
Distributed Processing



$R = F(R_1,\ldots,R_k)$

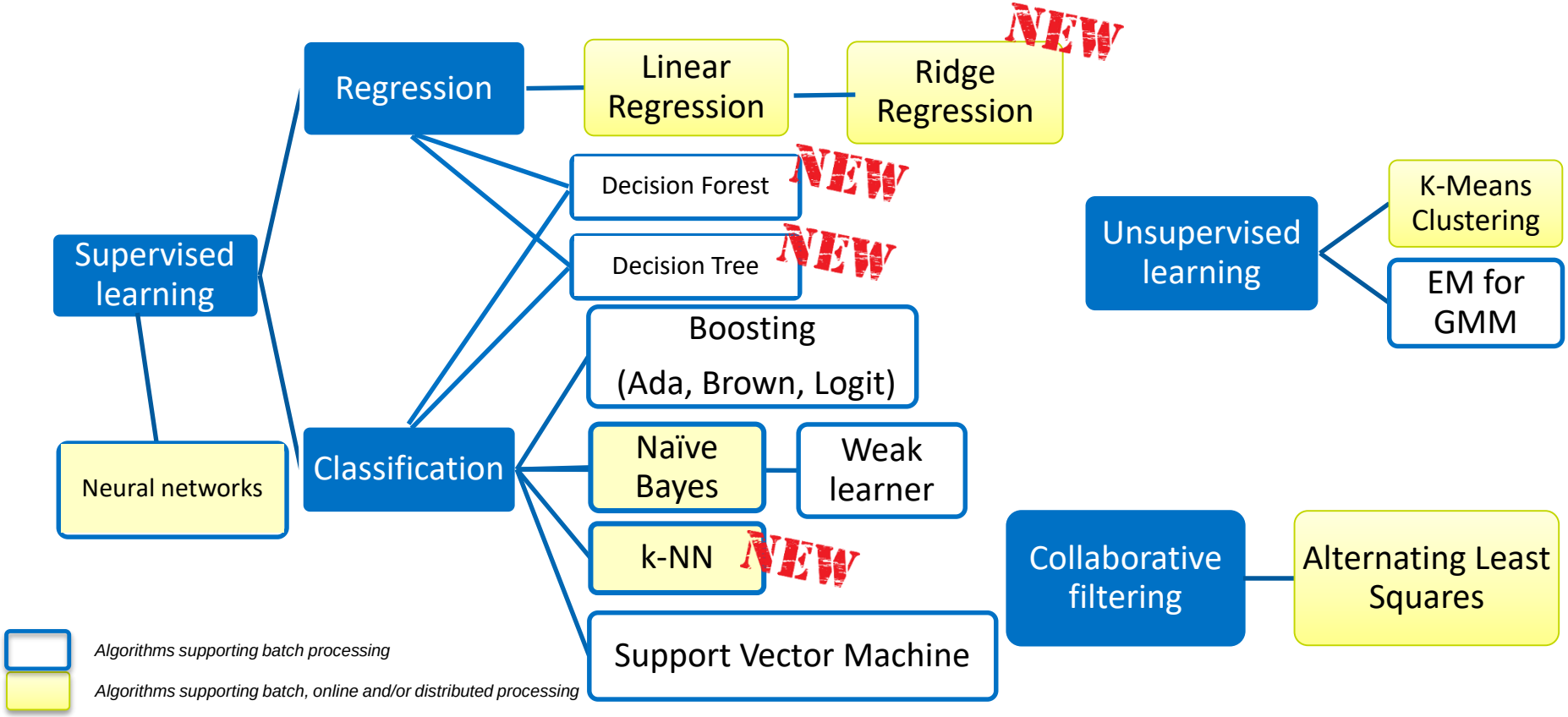
INTEL® DAAL ALGORITHMS

DATA TRANSFORMATION AND ANALYSIS IN INTEL® DAAL



INTEL® DAAL ALGORITHMS

MACHINE LEARNING IN INTEL® DAAL



AGENDA

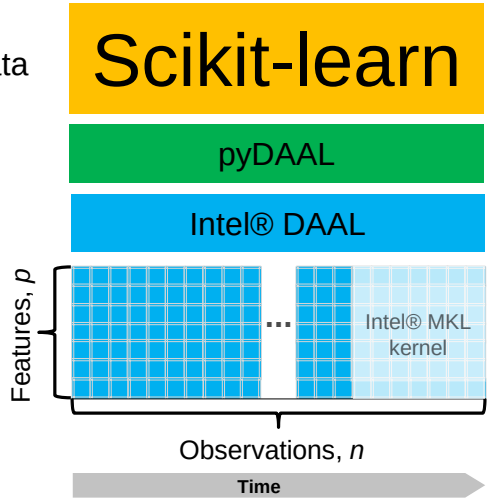
10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

intel Software

HANDS-ON INTEL-OPTIMIZED SCIKIT-LEARN

INTEL-OPTIMIZED SCIKIT-LEARN

Machine learning for in-memory homogeneous data



Powerful low-level API for machine learning with non-homogeneous, streaming & distributed data

Built upon highly optimized Intel® MKL kernels

HANDS-ON SCIKIT-LEARN

- Color Quantization using K-Means: 11_kmeans_sklearn.ipynb

HANDS-ON SCIKIT-LEARN

- Recognizing hand-written digits using SVM: 11_svm_sklearn.ipynb

AGENDA

10:00	11:30	Intel® Distribution for Python
		Introduction and Overview
		Docker images
		Package management with conda
11:30	11:45	Coffee Break
11:45	13:00	Introduction to Intel® Vtune™ for Python
		Python Performance Techniques - Part 1
		Numpy, numexpr
		Profiling
13:00	14:00	Lunch
14:00	15:45	Python Performance Techniques - Part 1
		Numba, cython
		MPI
		Parallelism (dask and others)
15:45	16:00	Coffee Break
16:00	17:45	Classical Machine learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)
		Overview Intel® Math Kernel Library
		Overview Intel® DAAL
		Hands-on Kmeans / SVM / Others
		Tech Preview: daal4py/HPAT
17:45	18:00	Wrap-up

intel Software

PATHFINDING/TECH PREVIEW
DAAL4PY/HPAT

PREPARE HANDS-ON HPAT

```
conda create -n HPAT -c ehsantn -c anaconda -c conda-
forge hpat boost=1.66
```

TWO INGREDIENTS TO GET CLOSE-TO-NATIVE PERFORMANCE IN PYTHON

Pure Python

Serial
Interpreted

Python + Libraries

Partially Ninja-level
Partially Interpreted

Libraries + JIT

Largely Ninja-level
100% native

C++

100% Ninja-level
100% native

CLOSE-TO-NATIVE PERFORMANCE ON A CLUSTER

- Extend the same two ingredients to cover distributed computation
  - Distributed performance library: daal4py
  - JIT compiler to partition and distribute data/operations: HPAT

CHALLENGES

- Ease of use (also for existing code)
- Efficiency
- Eliminate Python overhead
- Efficient communication
- Parallelism, data-partitioning and -distribution
- Out-of-the-box "standard" algorithms
- Custom code
- Experimenting with custom algorithms
- Data manipulation (pandas)

SCALABLE ALGORITHMS

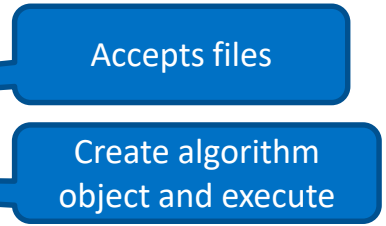
- Simple (scikit-learn-like) Python API that can scale out to a cluster
  - 1-line per classical algorithm
  - Use DAAL as the backend
  - Apply HPC technologies (MPI)
  - Work on numpy arrays and pandas data-frames
  - Hide compute mode (single-node, cluster, streaming) details from user

Simple example: PCA

Simple Syntax

```
import daal4py as d4p
file = "pca.csv"
result = d4p.pca().compute(file)
```

```
python pca.py
```



SIMPLE EXAMPLE: PCA

Simple Syntax

```
import daal4py as d4p
file = "pca.csv"
result = d4p.pca().compute(file)
```

```
python pca.py
```

Easy distribution: Single Process View

```
import daal4py as d4p
d4p.daalinit()
files = ["pca1.csv", "pca2.csv", ...]
result = d4p.pca(distributed=True).compute(files)
d4p.daalfini()
```

```
mpirun -n 4 -genv DIST_CNC=MPI python ./pca.py
```

Initialize

Multiple input arrays/files

Request distributed execution

Easy distribution: SPMD View

```
import daal4py as d4p
d4p.daalinit(spmd=True)
file = "pca_me.csv"
result = d4p.pca(distributed=True).compute(file)
d4p.daalfini()
```

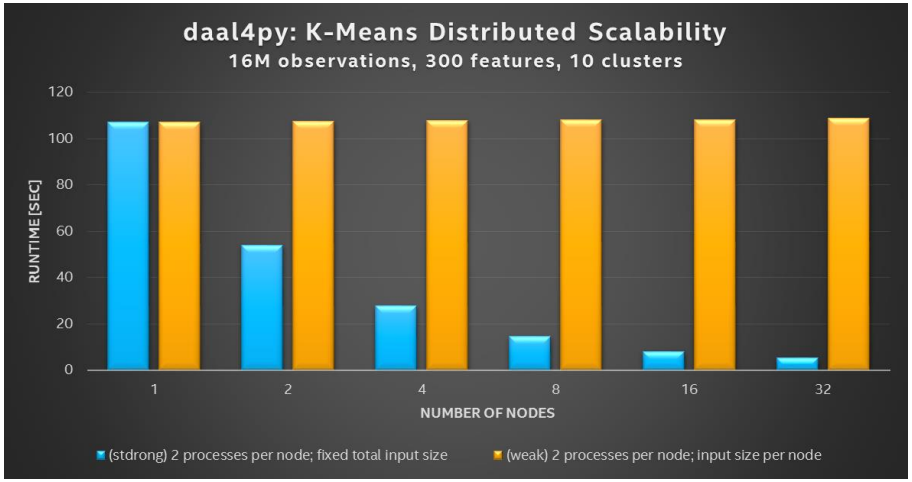
```
mpirun -n 4 -genv DIST_CNC=MPI python ./pca.py
```

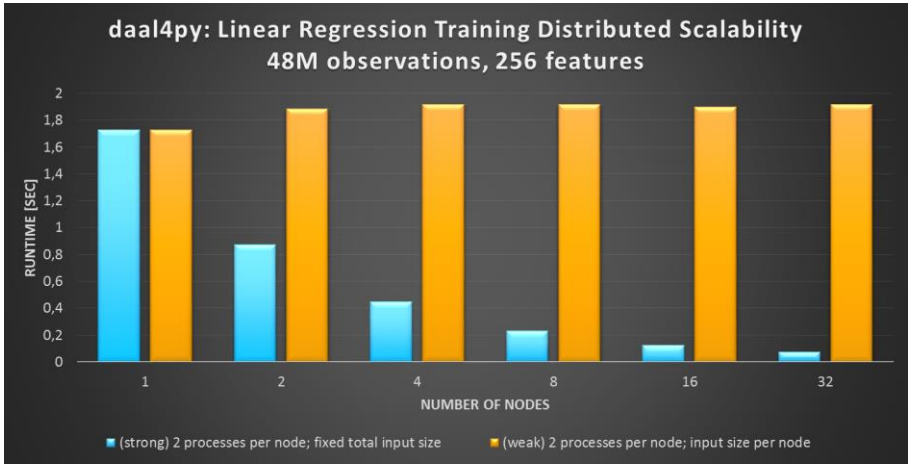
Initialize for SPMD

One file per process

SCALABILITY

- No difference in API overhead compared to official Python interface to DAAL
- Good strong and weak scaling
- Excellent tradeoff (simplicity vs scalability)





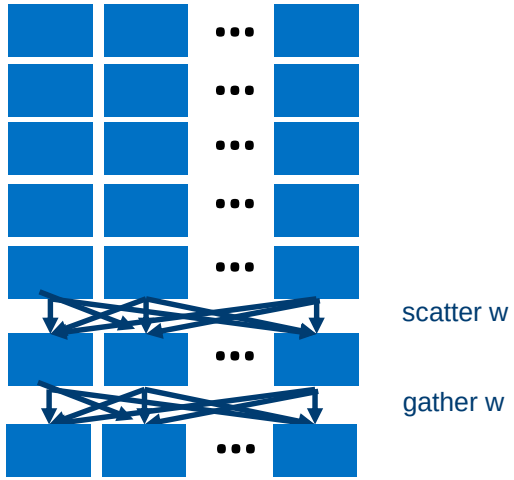
Hardware	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz, EIST/Turbo on
	2 sockets, 20 Cores per socket
	192 GB RAM
	16 nodes connected with Infiniband
Operating System	Oracle Linux Server release 7.4
Data Type	double

PYTHON COMPILER FOR AUTO-DISTRIBUTION: HPAT

- A JIT framework which automatically distributes **data** and **operations** (and **computation-kernels)** for data analytics codes
- High performance/scalability for analytics/ML/AI with little effort
  - Minimal changes to scripting source code
- Compiler optimization and parallelization (based on numba)
  - Scripting program → efficient parallel **MPI binary**
- *High Performance Analytics Toolkit (HPAT)*
- Aims to support popular data/file formats like pandas, numpy, hdf5, parquet,…

HPAT HDF5 Example

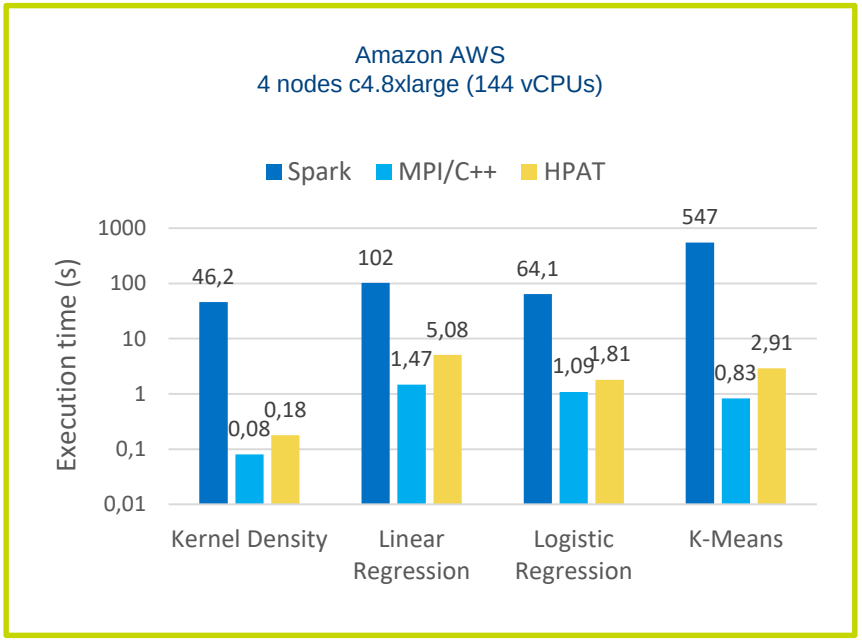
```
@hpat.jit
def logistic_regression(iterations):
    f = h5py.File("lr.hdf5", "r")
    X = f['points'][:]
    Y = f['responses'][:]
    D = X.shape[1]
    w = np.ones(D) - 0.5
    for i in range(iterations):
        w -= np.dot(((1.0 / (1.0 + np.exp(-Y * np.dot(X, w))) - 1.0) * Y), X)
return w
```

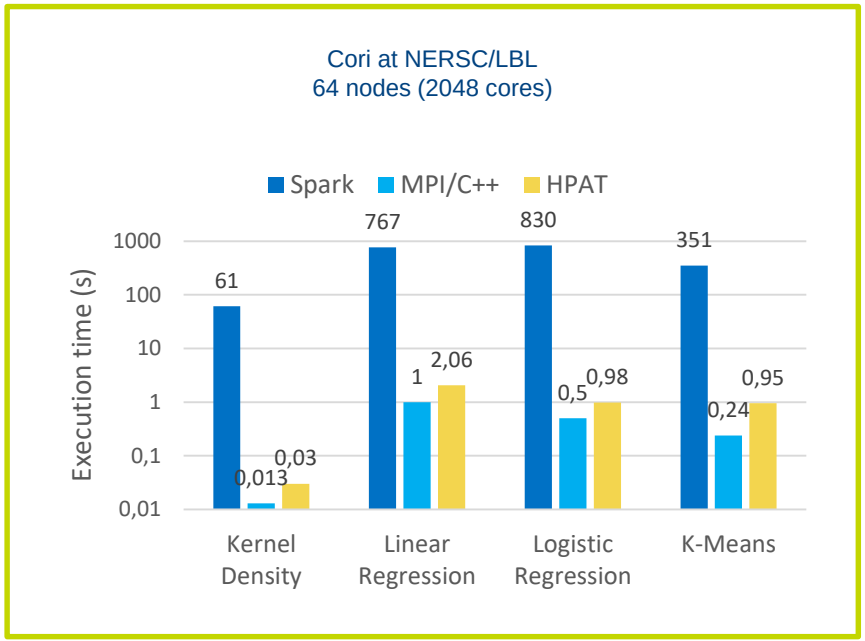


Example launch command:

```
mpirun -n 144 python logistic_regression.py
```

Performance Evaluation





HPAT is within 2-4x MPI/C++

HPAT Julia used, Python will be similar

Totoni et al. “HPAT: High Performance Analytics with Scripting Ease-of-Use”, *ICS’17*

Summary

- Compiler approach superior to library approach for analytics
- HPAT bridges productivity-performance gap
  - Compiles Python programs to efficient parallel binaries
  - Available on GitHub: https://github.com/IntelLabs/hpat

Technical Preview (daal4py, HPAT)

https://software.intel.com/en-us/articles/daal4py-overview-a-high-level-python-api-to-the-intel-data-analytics-acceleration-library
https://github.com/IntelLabs/hpat

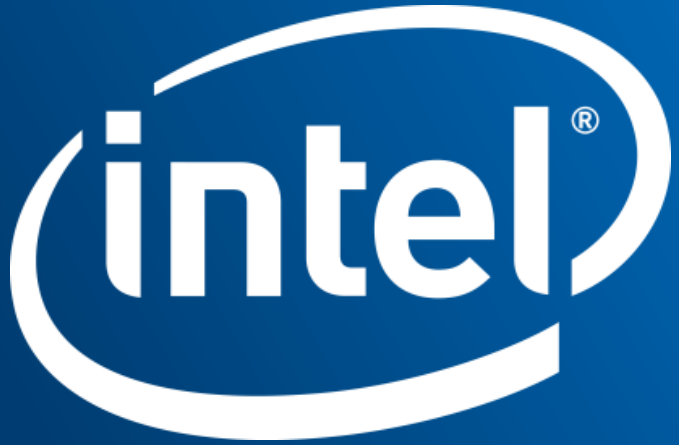
- No official product
- Open Source (Apache 2.0)
- Binary packages available and recommended
- Looking for feedback
  - Easy to use?
  - Right level of abstraction?
  - Better performance than existing solution(s)?
  - Suggestions for improvements, whish lists…

HANDS-ON HPAT

```
conda create -n HPAT -c ehsantn -c anaconda -c conda-forge hpat boost=1.66
```

- Linear Regression with HPAT: 21_linreg_hpat.ipynb

THE END


intel