MORE PERFORMANCE USING INTEL® DISTRIBUTION FOR PYTHON

Frank Schlimbach
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

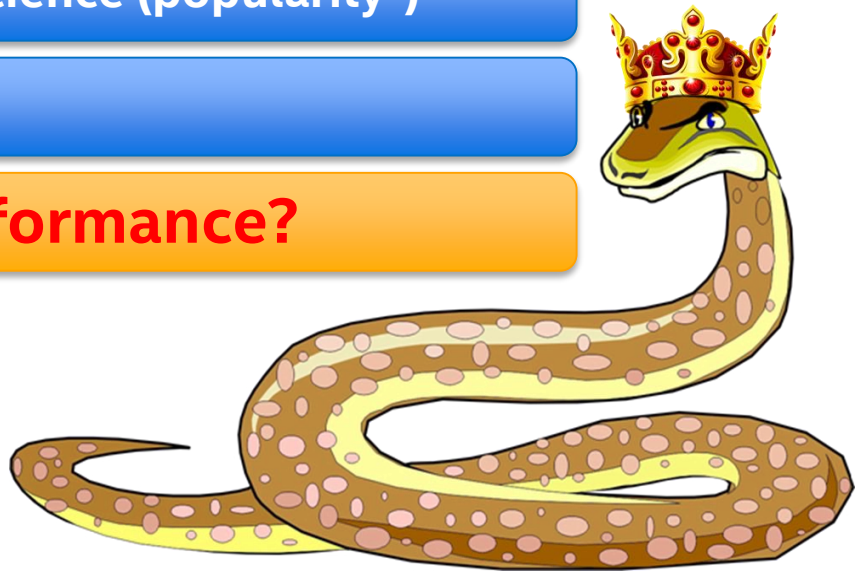
Most Wanted: Python

~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?**



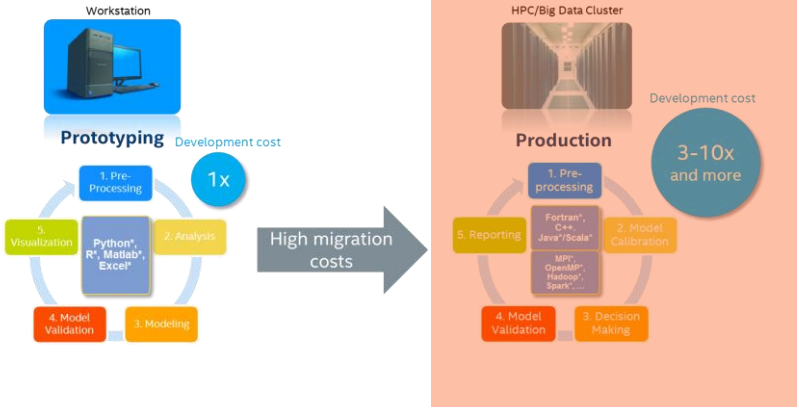
[1] https://www.kdnuggets.com/2017/01/most-popular-language-machine-learning-data-science.html

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]*

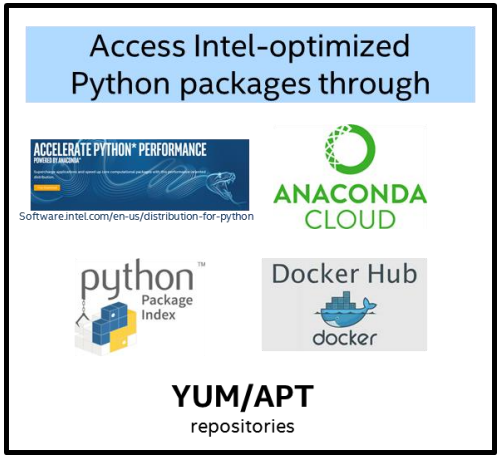
[1]Intel® Math Kernel Library
[2]Intel® Data Analytics Acceleration Library

[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.

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

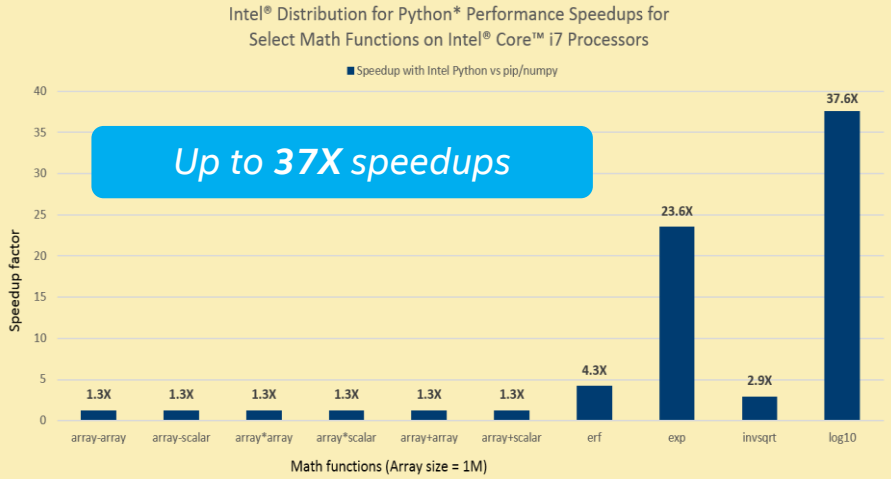
Close to Native Performance

Micro-benchmarks form Numpy/Scipy show performance similar to C (Intel® Math Kernel Library [MKL])

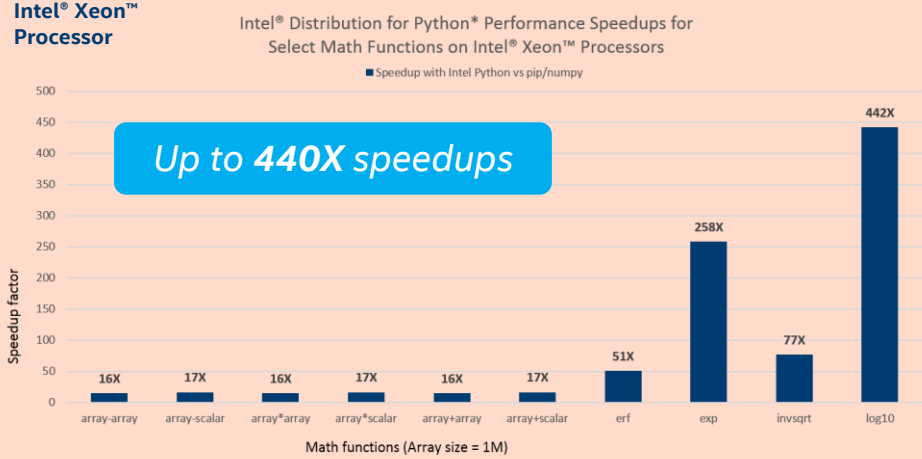
- In many cases > 90% of native performance
  - Fast Fourier Transformation
  - Linear Algebra
  - Random Number Generation
  - ML algorithms
- See backup for details

UMath Optimizations & Vectorization to Utilize Multiple Cores, Memory Management

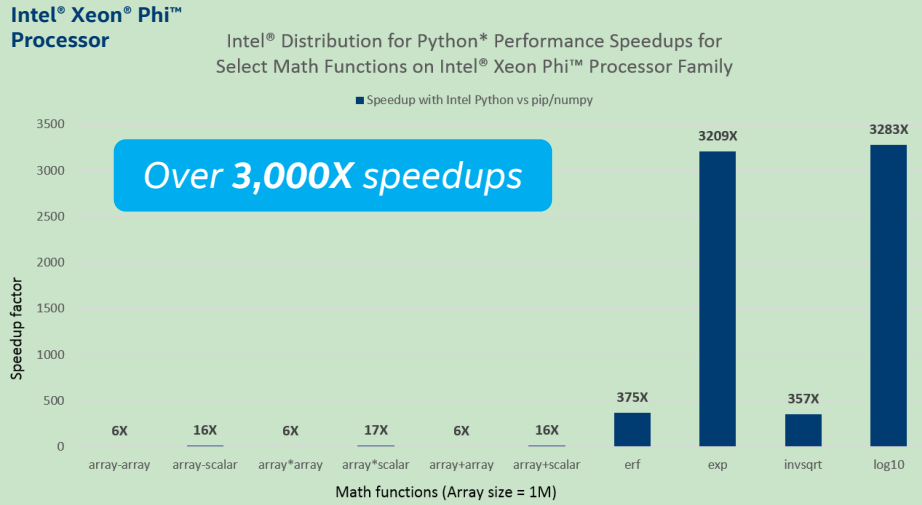
Intel® Core™ i7 Processor



Intel® Xeon™ Processor



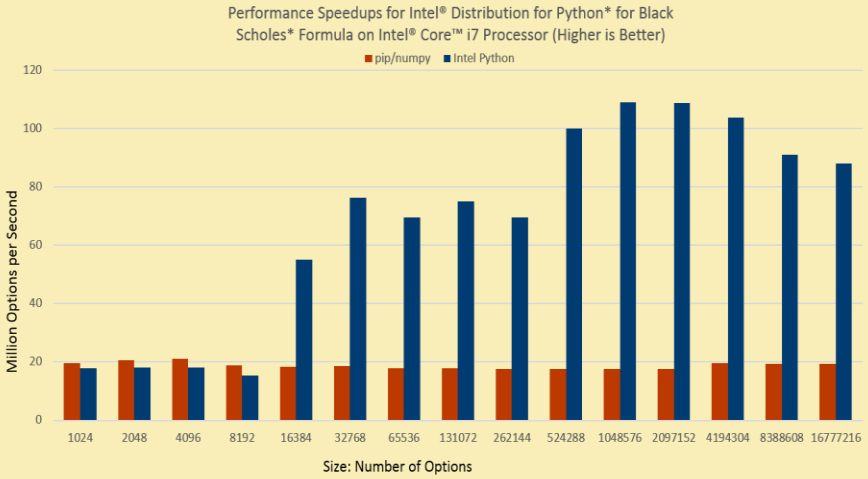
Intel® Xeon® Phi™ Processor



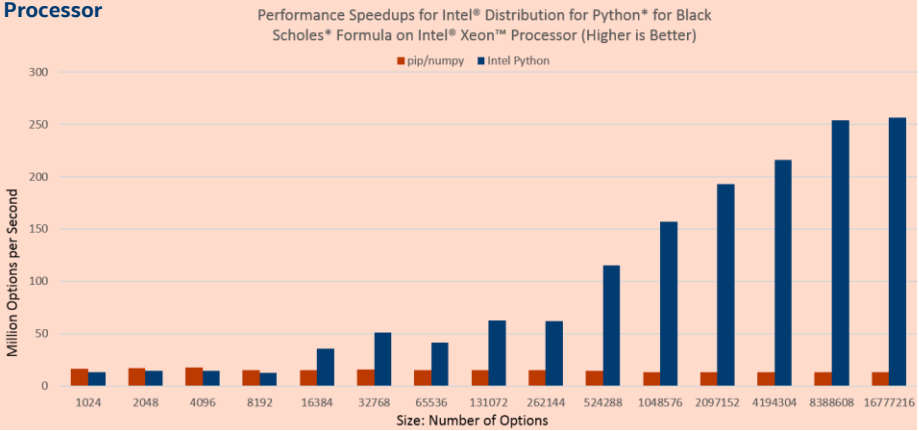
Hardware: Intel® Core™ i7-7567U CPU@3.50GHz (1 socket, 2 cores per socket, 2 threads per core), 32GB DDR4 @ 2133MHz. Intel® Xeon® CPU E5-2699 v4@2.20GHz (2 sockets, 22 cores per socket, 1 thread per core-HT is off), 256GB DDR4@2400MHz. Intel® Xeon Phi™ CPU 7250@1.40GHz (1 socket, 68 cores per socket, 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

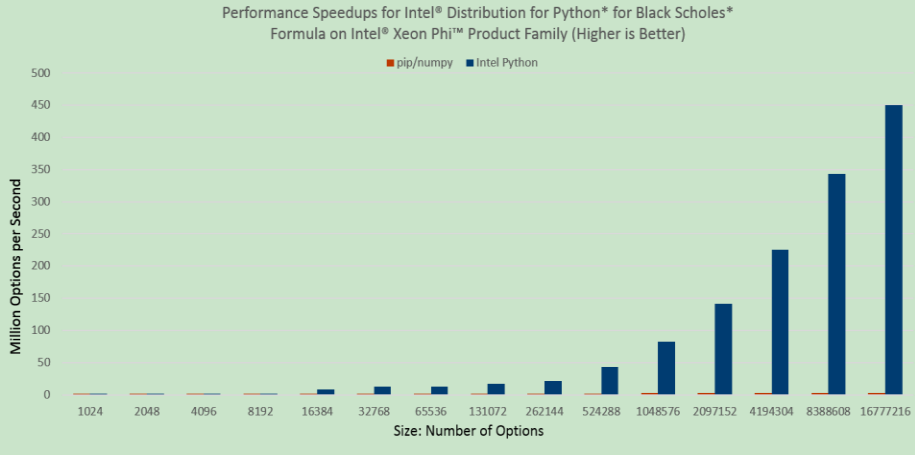
Intel® Core™ i7 Processor



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.

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

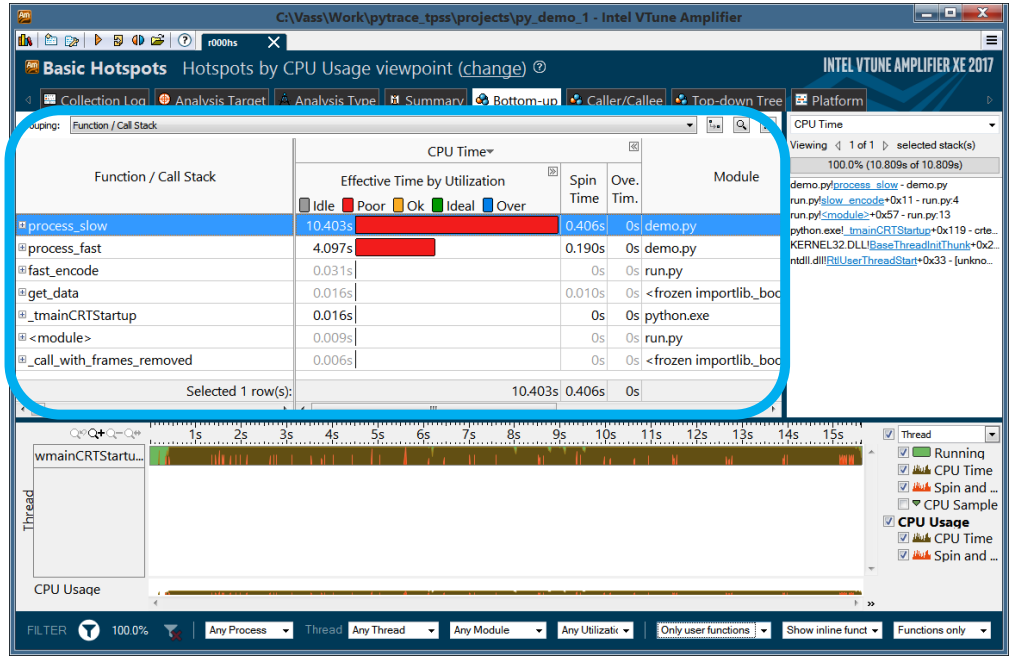
Analyze Performance with Intel® VTune™ Amplifier (available in Intel® Parallel Studio XE)

Challenge

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

Solution

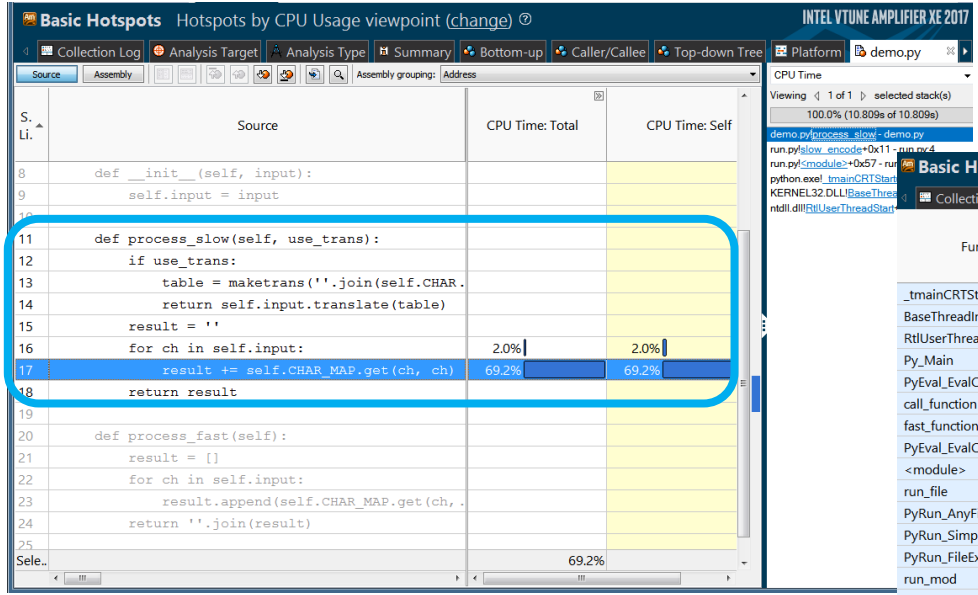
- Auto-detect mixed Python/C/C++ code & extensions
- Accurately identify performance hotspots at line-level
- Low overhead, attach/detach to running application
- Focus your tuning efforts for most impact on performance



Auto detection & performance analysis of Python & native functions

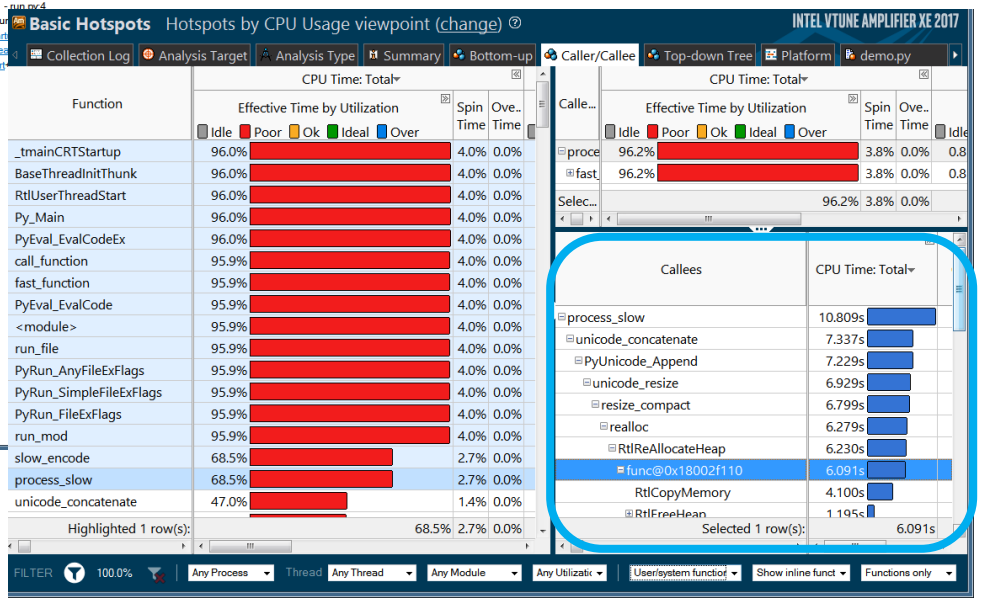
Available in Intel® VTune™ Amplifier & Intel® Parallel Studio XE

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

USING CONDA

Conda

Package manager

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

Environments

- 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:

- `conda deactivate`
- `conda create -n handson -c intel python=3.6 notebook numpy mpi4py scipy scikit-learn matplotlib pillow`
- `conda activate handson`

Windows:

- `deactivate`
- `conda create -n handson -c intel python=3.6 notebook numpy mpi4py scipy scikit-learn matplotlib`
- `activate handson`
- `pip install pillow`

intel Software

HANDS-ON NUMPY

Preparing hands-on sessions

```
conda deactivate

conda create -n handson -c intel python=3.6 notebook numpy mpi4py scipy
scikit-learn matplotlib pillow

conda activate handson


which python

source /opt/intel/parallel_studio_xe_2018/bin/psxevars.sh intel64

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

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, …

Hands-On Numpy

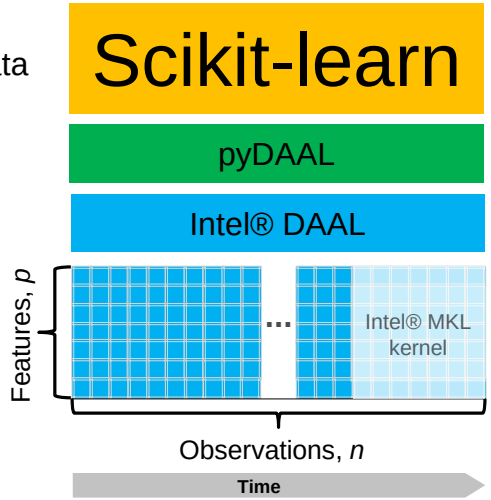
Black Scholes option pricing: 01_numpy_blackscholes.ipynb

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

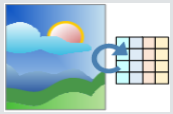
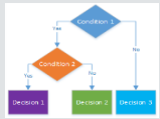
Speedup Analytics & Machine Learning with Intel® Data Analytics Acceleration Library (Intel® DAAL)

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

Learn More: software.intel.com/daal

What's New in the 2018 Edition

- New Algorithms
  - Classification & Regression Decision Tree and 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.

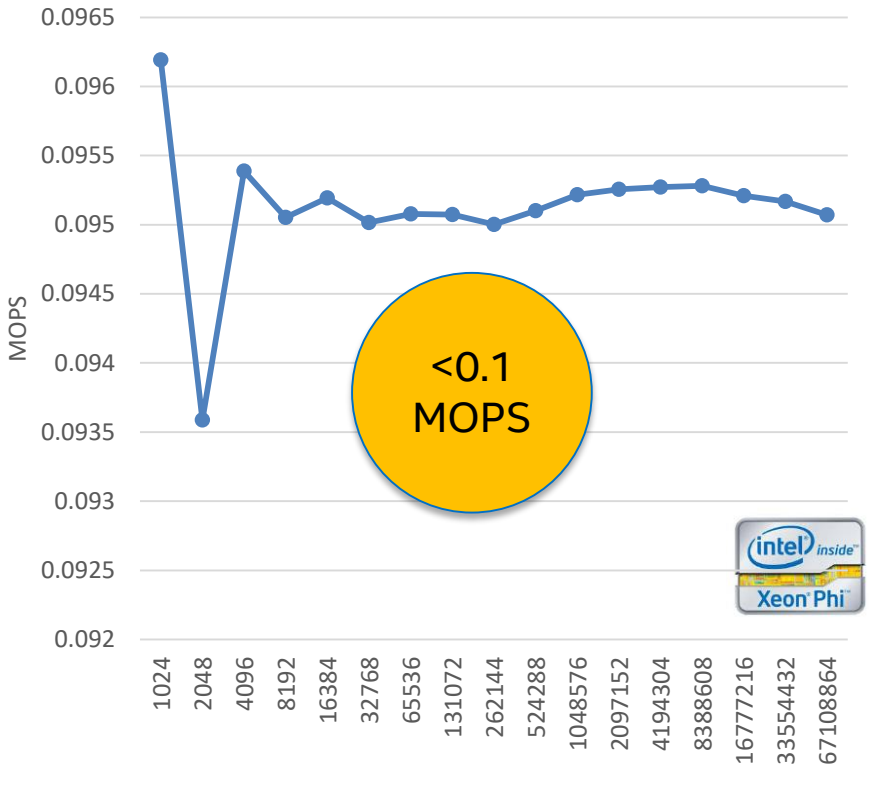
Hands-on Scikit-learn

Color Quantization using K-Means: 02_kmeans.ipynb

intel Software

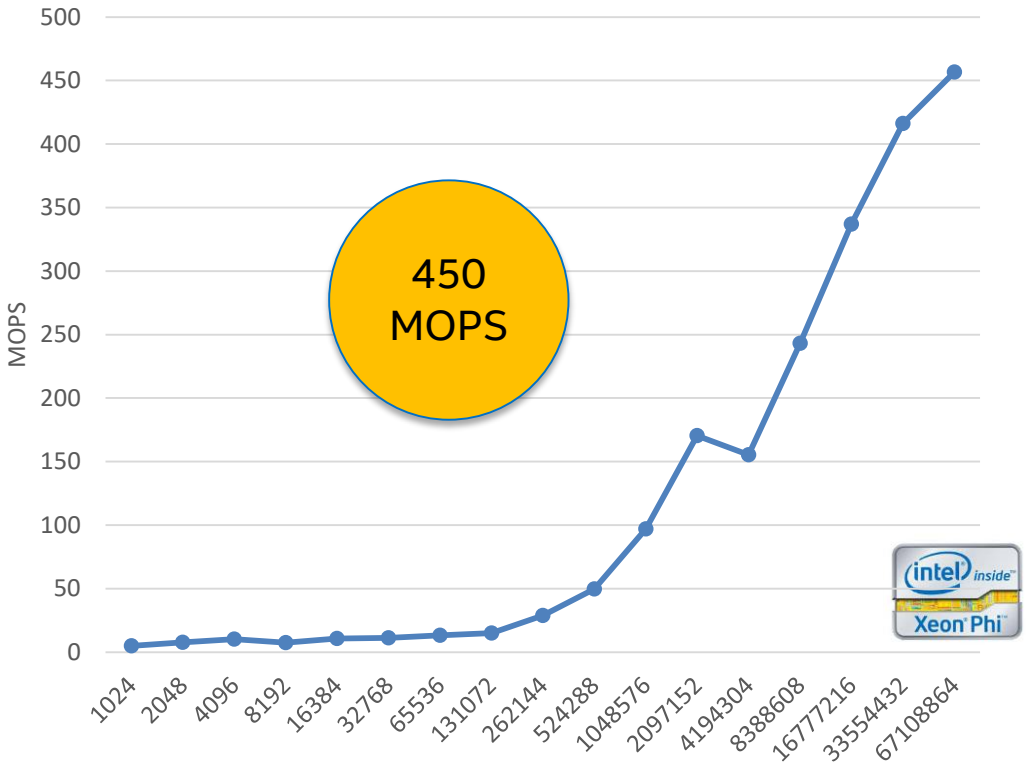
USE CASE
PERFORMANCE TOOLS APPLIED TO BLACK SCHOLES

Variant 1: Plain Python



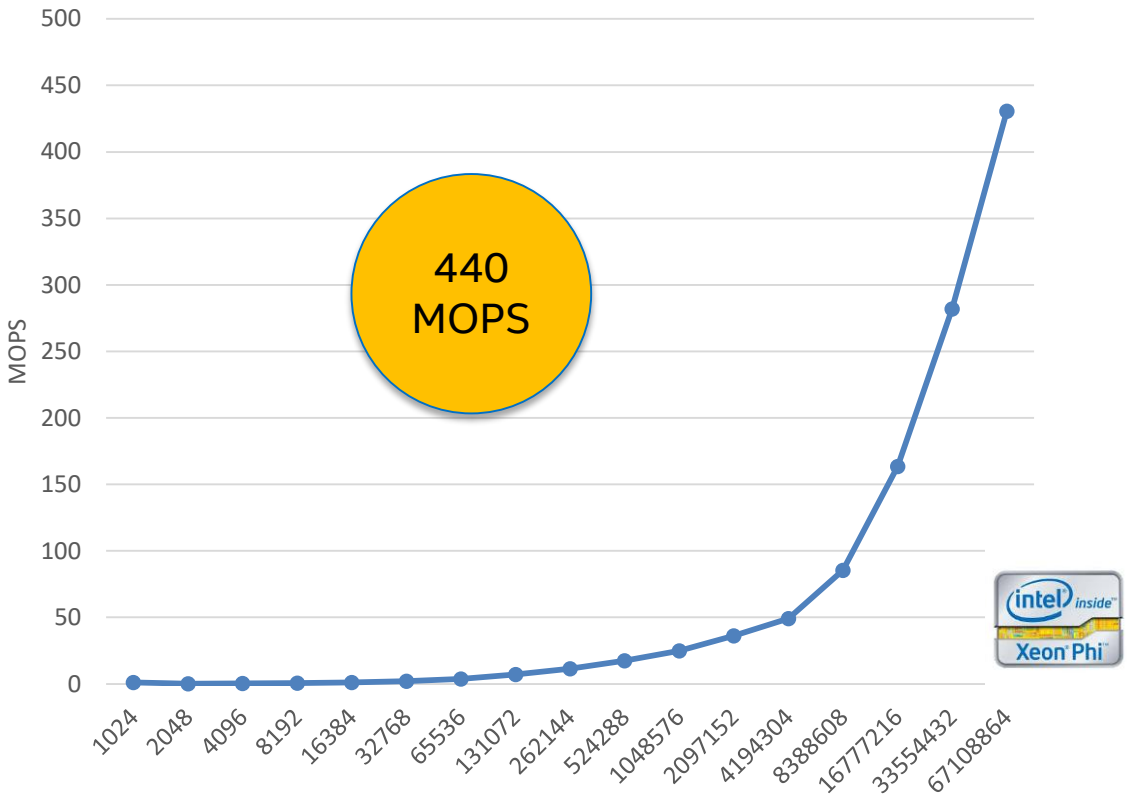

```python
def black_scholes ( nopt, price, strike, t, rate, vol, call, put ):
    mr = -rate
    sig_sig_two = vol * vol * 2

    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



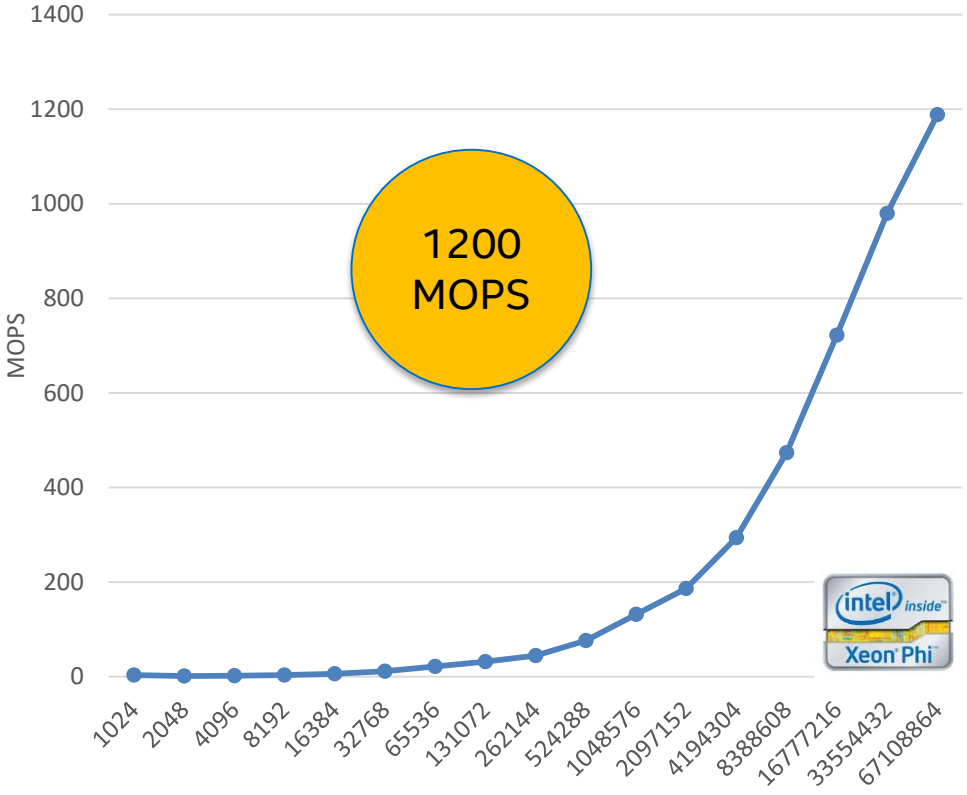

```
def black_scholes ( nopt, price, strike, t, rate, vol ):
    mr = -rate
    sig_sig_two = vol * vol * 2

    P = price
    S = strike
    T = t

    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 = P * d1 - Se * d2
    put = call - P + Se

    return call, put
```

Variant 3: NumExpr* (proxy for Umath implementation)



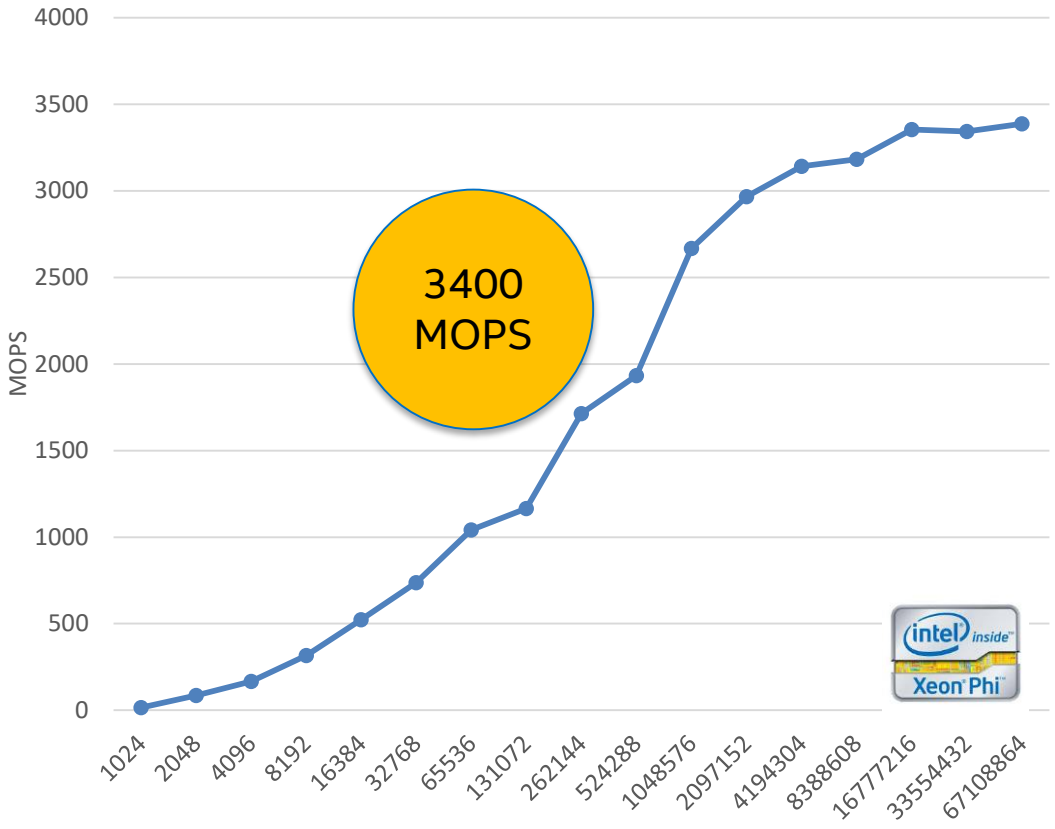

```python
import numexpr as ne

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)



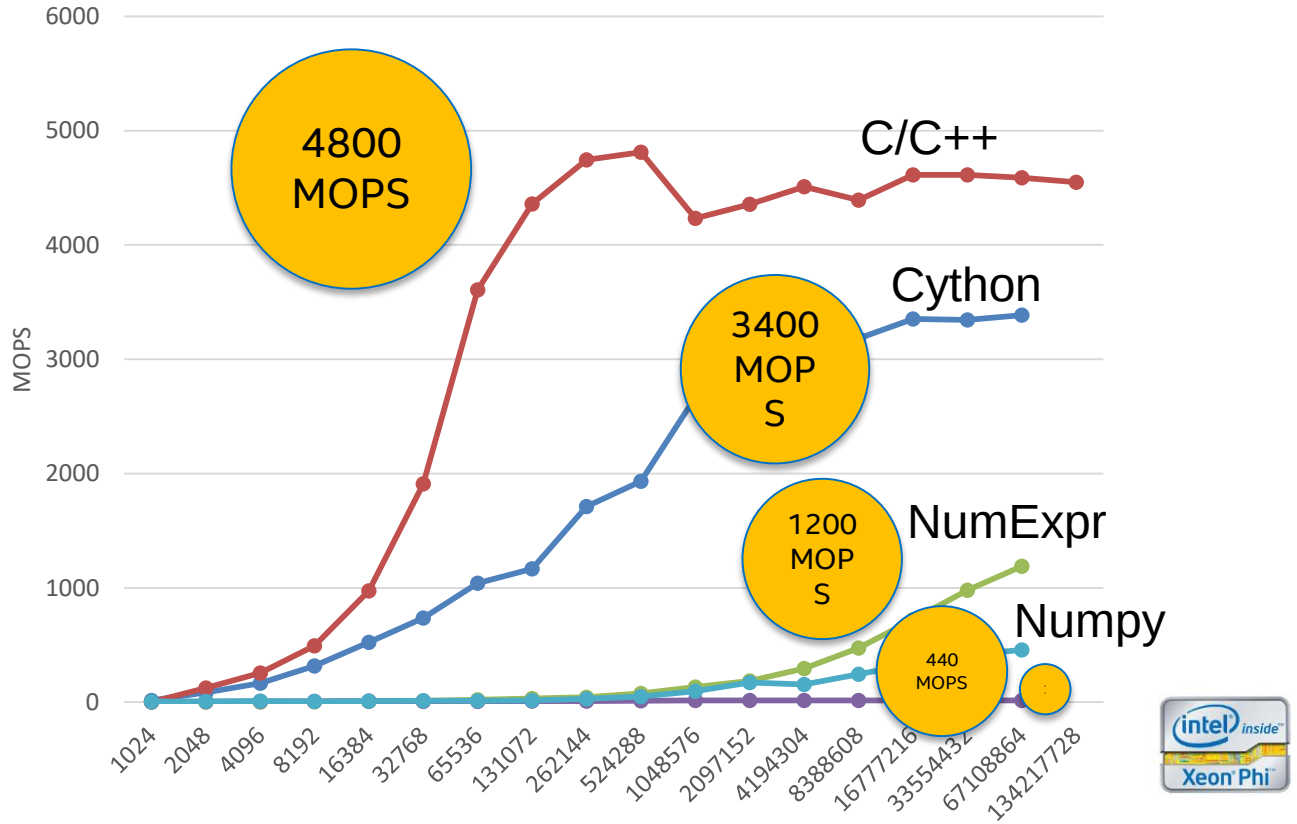

```python
import base_bs_erf
import numexpr as ne

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



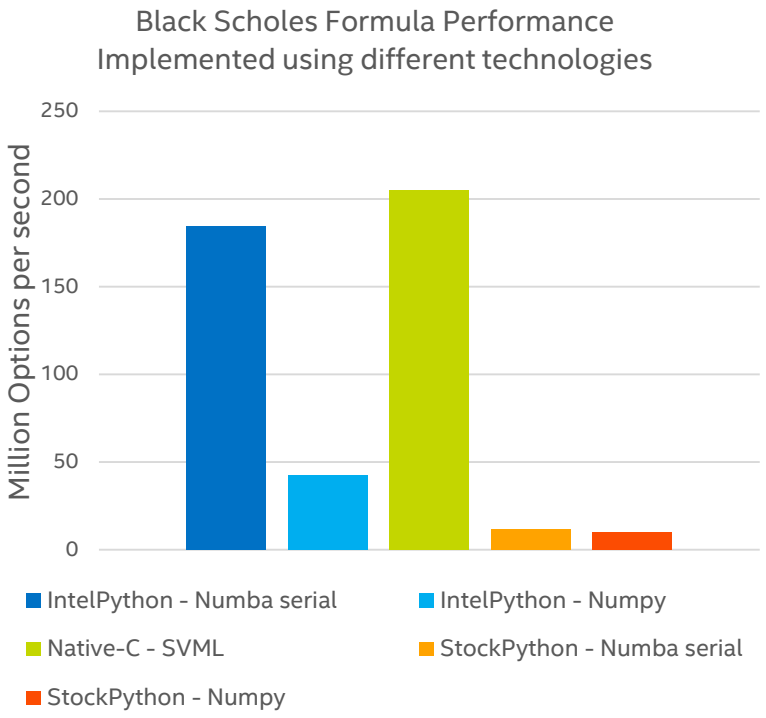
Optimizing NumPy, SciPy, NumExpr to scale

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



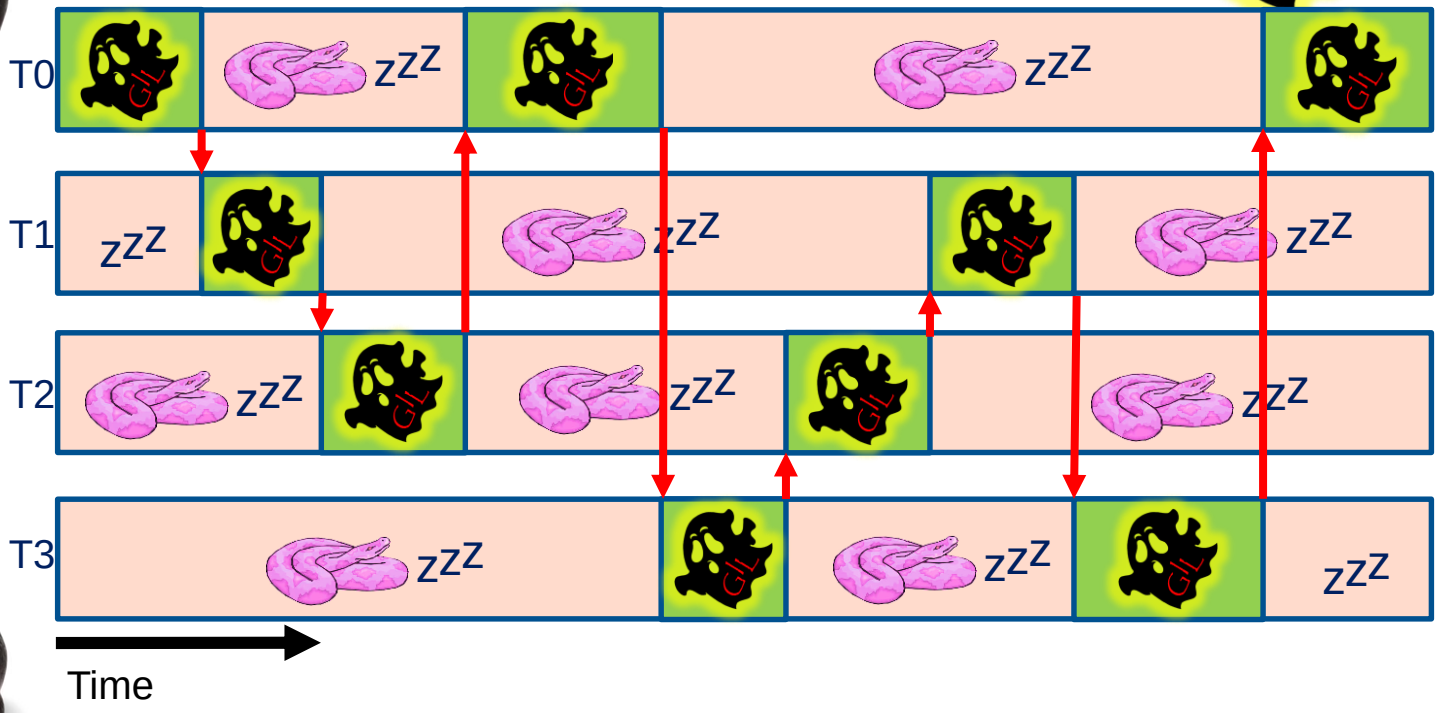
Configuration Info: Intel® Xeon® Gold 6140 CPU @ 2.30GHz (2 sockets, 18 cores per socket, 1 thread per core - HT is off), 256GB DDR4. Stock Python: CentOS Linux release 7.3.1611 (Core), python 3.6.4, numpy 1.14.0, numba 0.36.2, llvmlite 0.21.0. Intel® Distribution for Python 2018 Update 2 packages: python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

intel Software

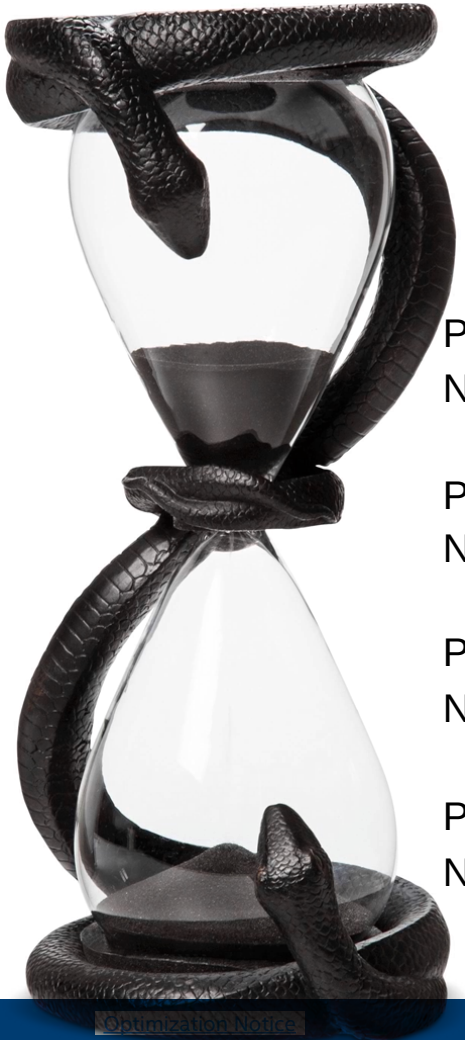
HANDS-ON PYHTON EXPLICIT PARALLELISM

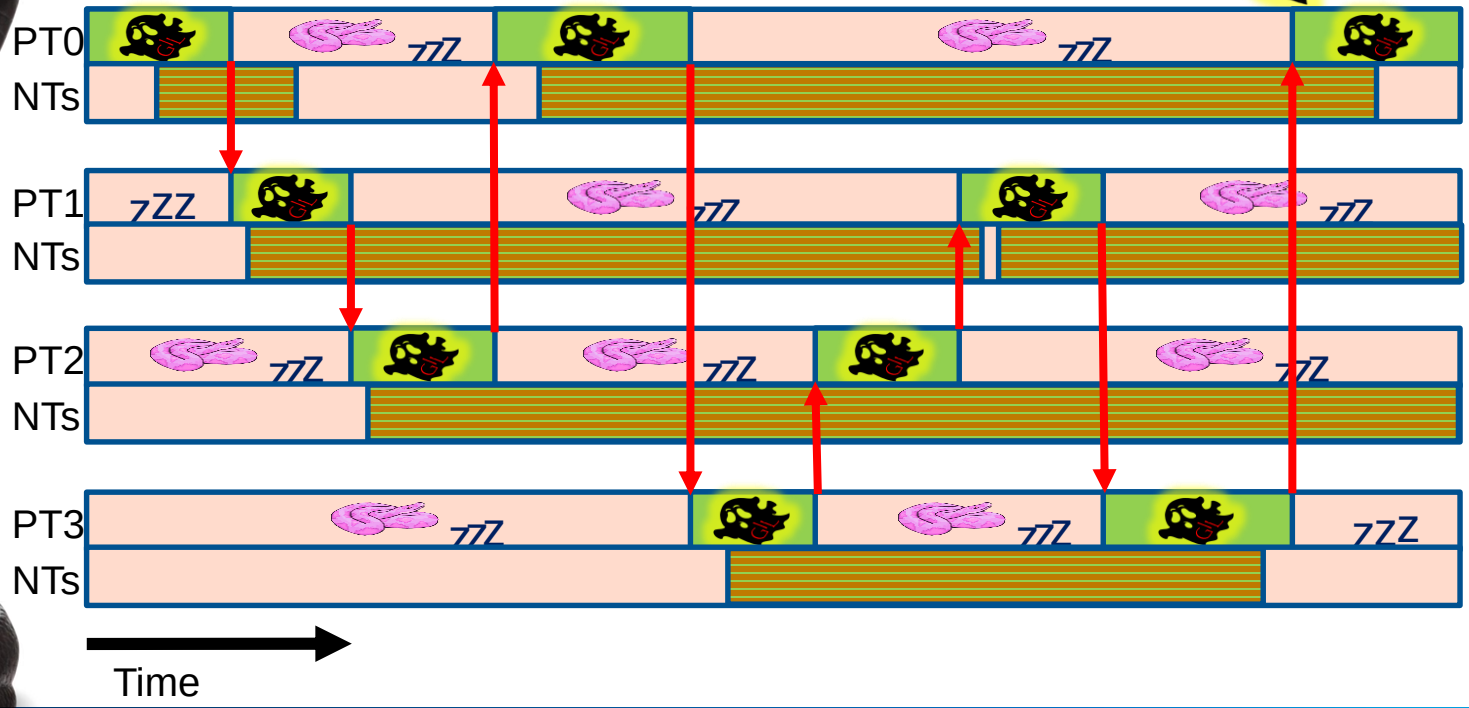
Global Interpreter Lock

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

Parallelism in Python - MPI

Kernel: fan-out & compute & fan-in

Opportunity for proper SPMD

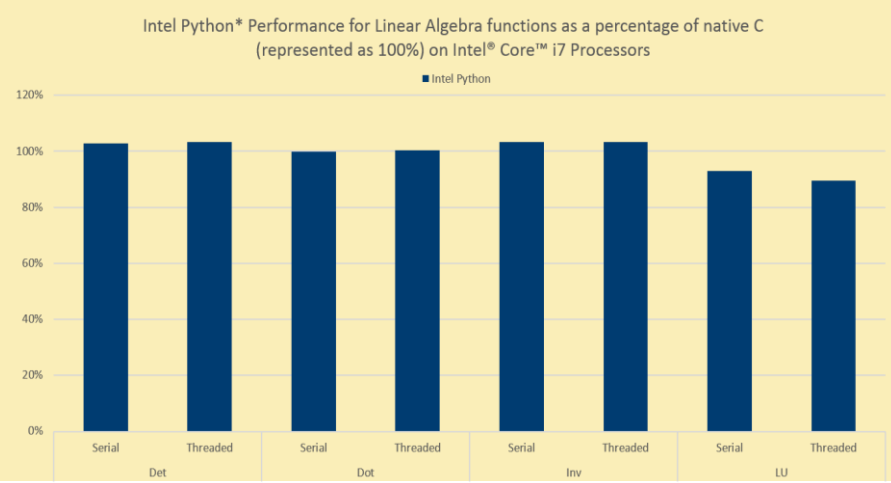
THE END

intel Software

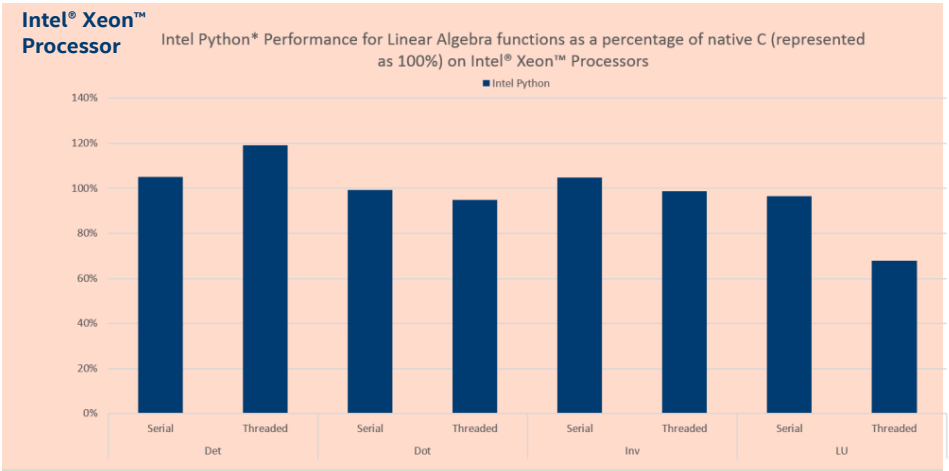
BACKUP

Native Code like Performance Speeds for Linear Algebra

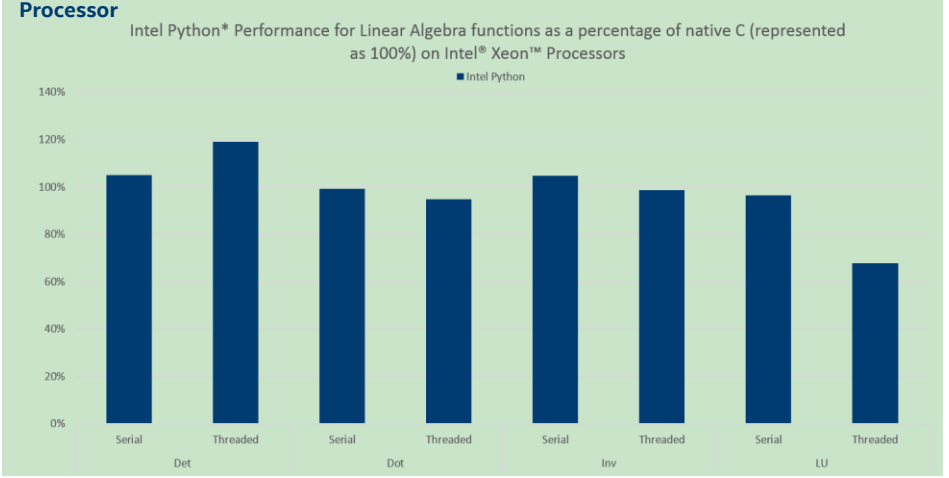
Intel® Core™ i7 Processor



Intel® Xeon™ Processor

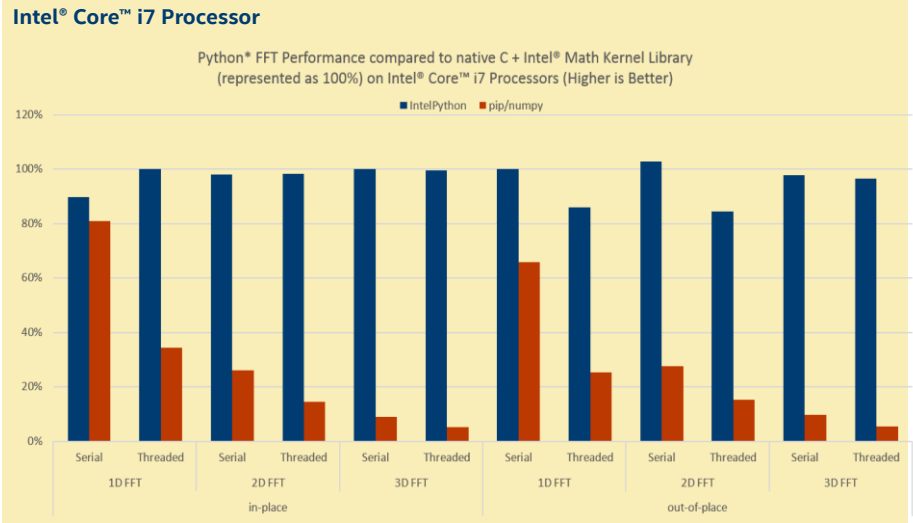


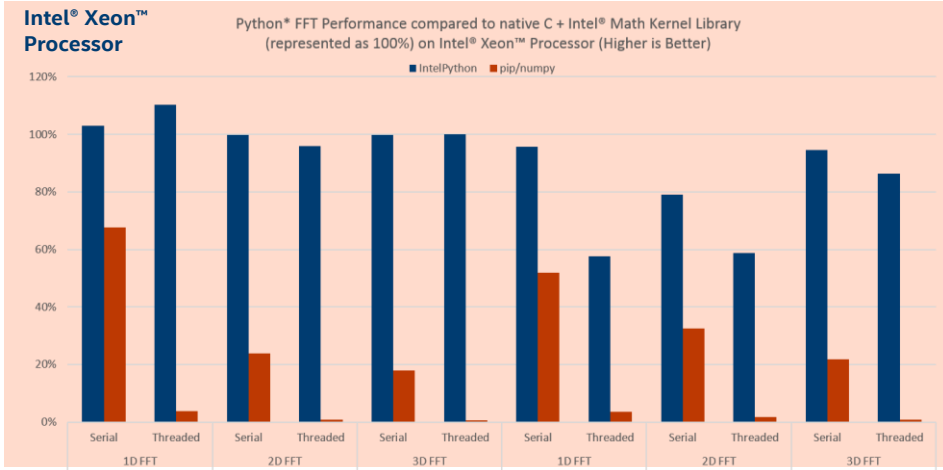
Intel® Xeon® Phi™ Processor

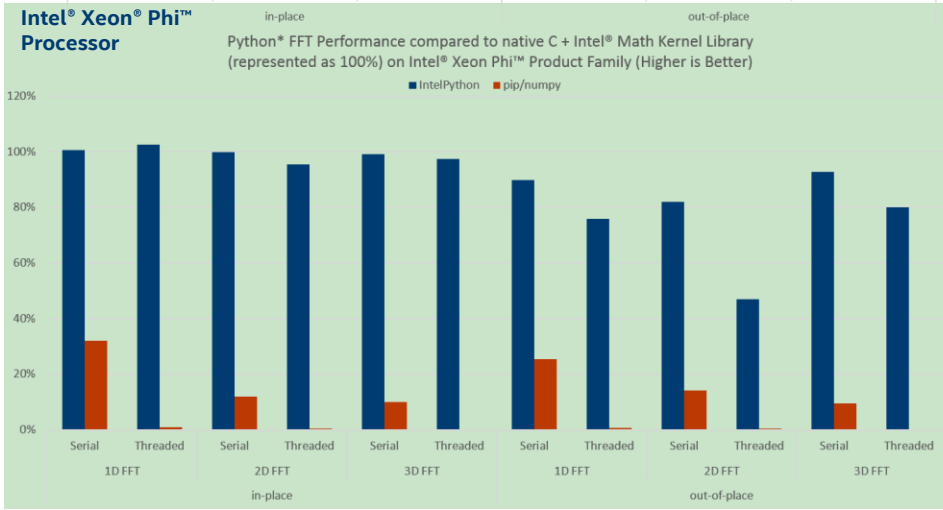


Hardware: Intel® Core™ i7-7567U CPU@3.50GHz (1 socket, 2 cores per socket, 2 threads per core), 32GB DDR4 @ 2133MHz. Intel® Xeon® CPU E5-2699 v4@2.20GHz (2 sockets, 22 cores per socket, 1 thread per core-HT is off), 256GB DDR4@2400MHz. Intel® Xeon Phi™ CPU 7250@1.40GHz (1 socket, 68 cores per socket, 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



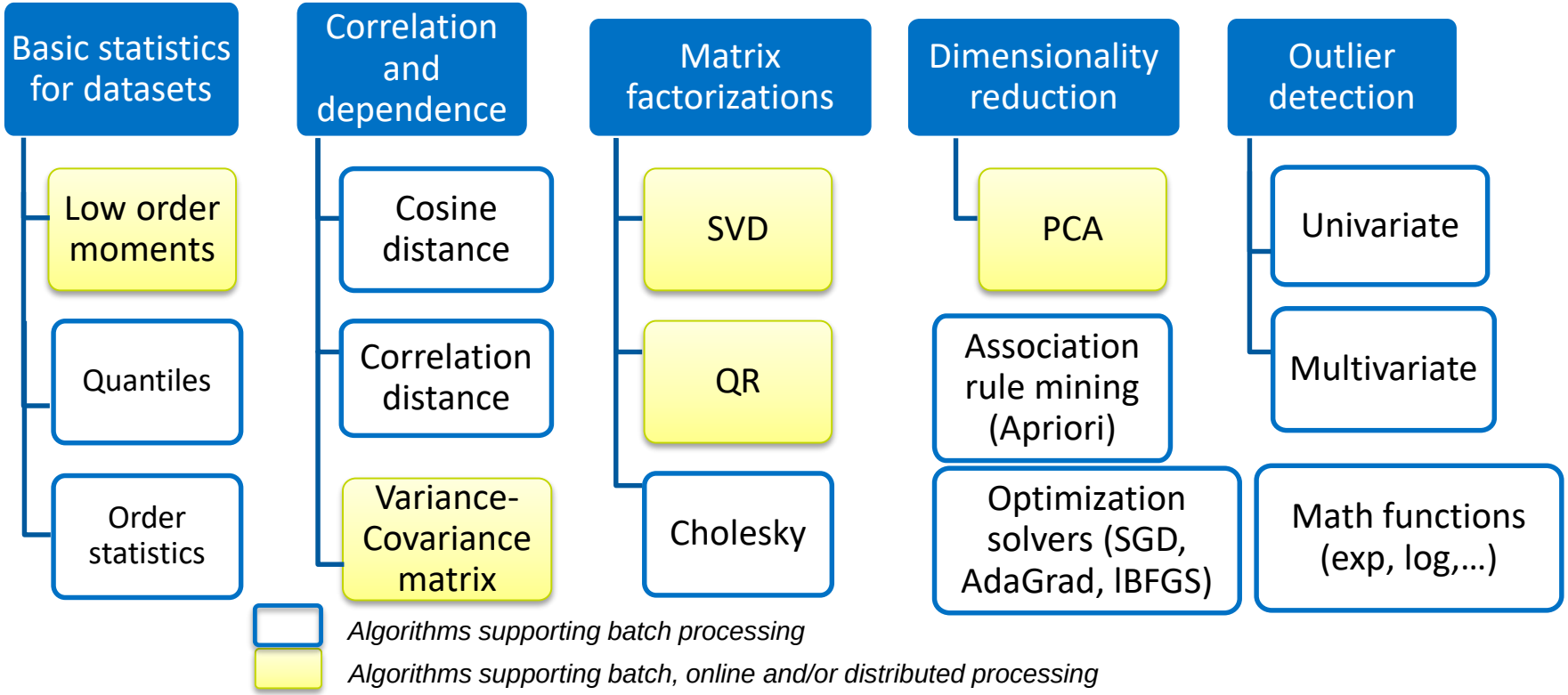




Hardware: Intel® Core™ i7-7567U CPU@3.50GHz (1 socket, 2 cores per socket, 2 threads per core), 32GB DDR4 @ 2133MHz. Intel® Xeon® CPU E5-2699 v4@2.20GHz (2 sockets, 22 cores per socket, 1 thread per core-HT is off), 256GB DDR4@2400MHz. Intel® Xeon Phi™ CPU 7250@1.40GHz (1 socket, 68 cores per socket, 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

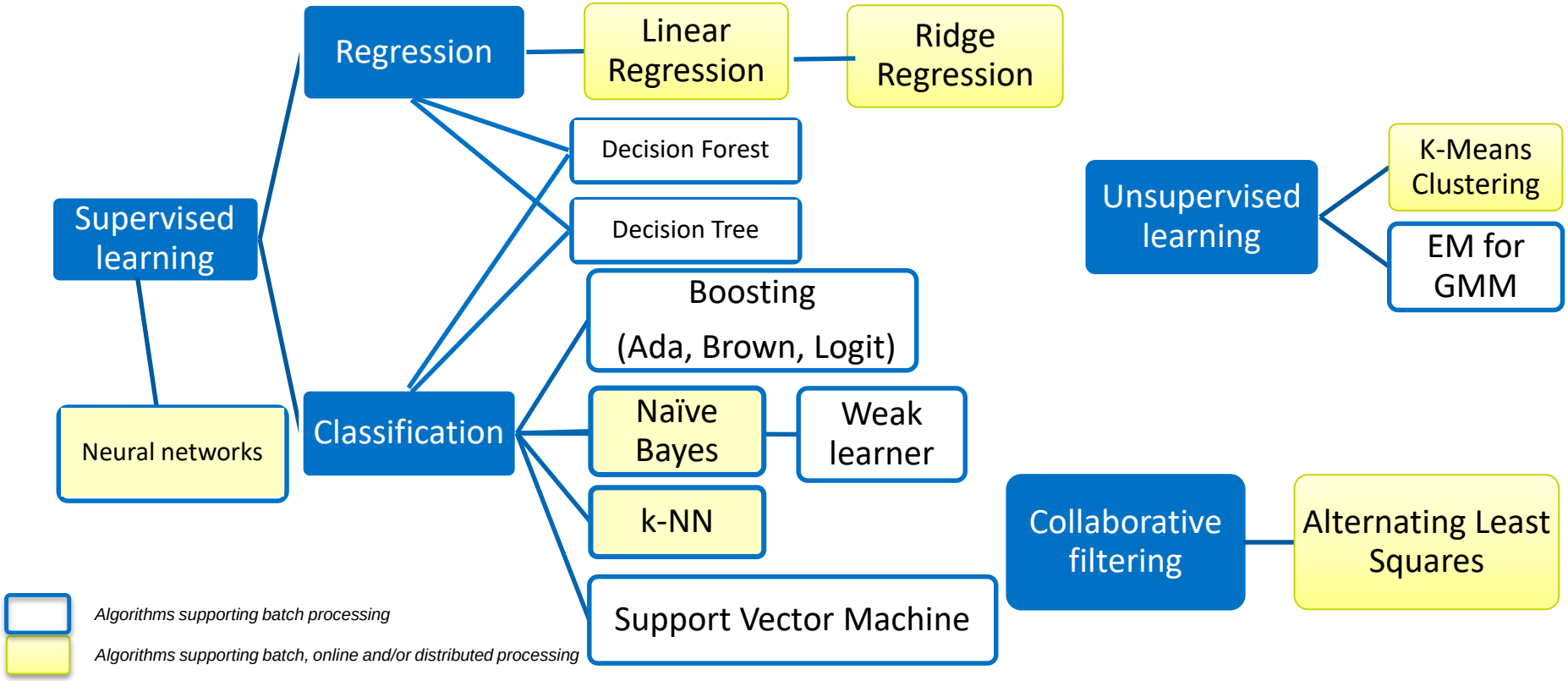
INTEL® DAAL ALGORITHMS

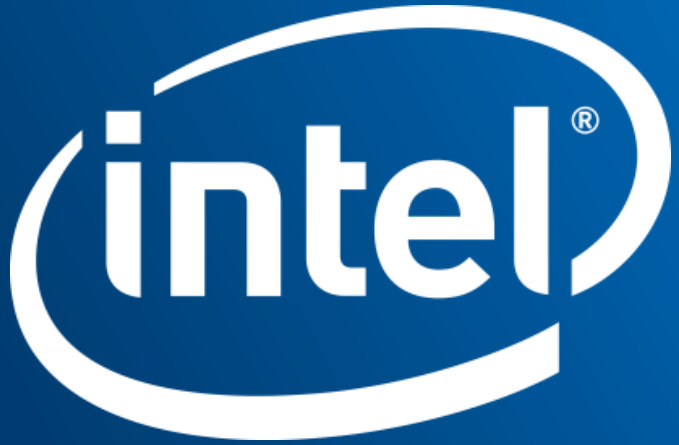
DATA TRANSFORMATION AND ANALYSIS IN INTEL® DAAL



INTEL® DAAL ALGORITHMS

MACHINE LEARNING IN INTEL® DAAL




intel