HIGH PERFORMANCE COMPUTING IN THE OPEN DATA SCIENCE ERA

Stan Seibert, Continuum Analytics
Sergey Maidanov, Intel

Presenter Bio

Stan Seibert
High Performance Python
Team Lead
Continuum Analytics

Stan Seibert received his Ph.D. in Physics from the University of Texas at Austin.

Prior to joining Continuum, he worked at Los Alamos National Laboratory, the University of Pennsylvania, and Mobi.

Stan has 10+ years of experience in areas including scientific and parallel computing, system administration, open-source software development, and Monte Carlo simulation.

Presenter Bio

Sergey Maidanov
Software Eng. Manager
Intel

Sergey Maidanov has 15+ years of experience in numerical analysis with a range of contributions to Intel software products such as Intel® MKL, Intel® IPP, Intel compilers and others.

He currently leads a team of software engineers working on the optimized Intel® Distribution for Python. Among his recently completed projects was the Intel® Data Analytics Acceleration Library.

Sergey received a master's degree in Mathematics from the State University of Nizhyny Novgorod.

CONTINUUM ANALYTICS

Agenda

1. High Performance Python for Data Science
2. Finding Bottlenecks with Profiling
3. Intel Tools and Libraries for High Performance
4. Speeding up Python with Compilation
5. Conclusion

ANACONDA

CONTINUUM®
ANALYTICS

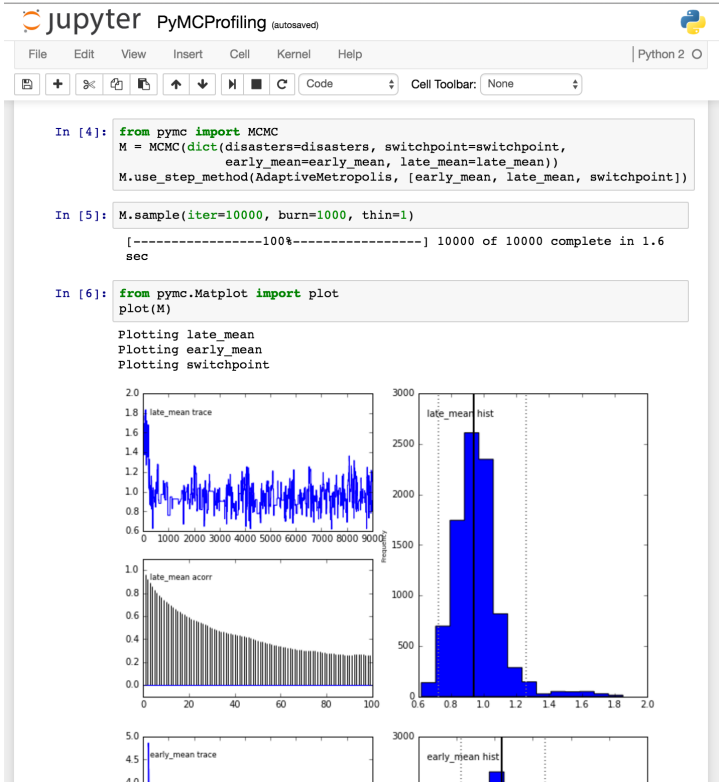
HIGH PERFORMANCE PYTHON FOR DATA SCIENCE

The Challenge of Data Science

- Data sizes constantly growing
- Need high performance to turn data into insight:
  - *Faster:* React more quickly to information
  - *Cheaper:* Spend less on computing
  - *Better:* Tackle previously impossible tasks

The Transformative Power of Performance

- Performance also enables interactivity
- Interactive analysis allows algorithms to extend your intuition!



ANACONDA®

Scale Up vs Scale Out

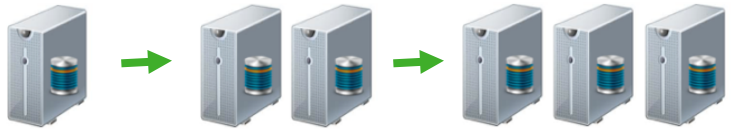
Scale Up
(Bigger Nodes)

Big Memory &
Many Cores
/ GPU Box

Best of Both
(e.g. GPU Cluster)

Many commodity
nodes in a cluster

Scale Out
(More Nodes)

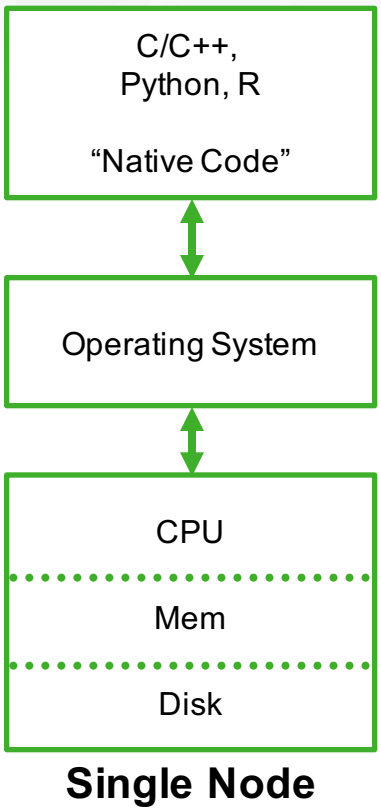


ANACONDA

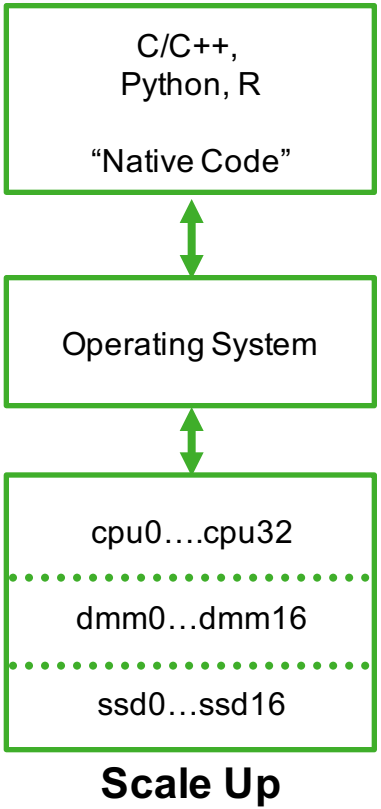
What to scale?

- Roughly speaking, program execution time is taken up by:
  1. I/O time: Time spent reading or writing data (files on disk, databases, cloud storage, etc.)
  2. Compute time: Time spent doing calculations on data already in memory
- Everyone focuses on #2, but often #1 is *equally important and frequently overlooked*

Scaling Up



Single Node



Scale Up

Same code, better performance

More compute

More memory

More disk

Computers come in many sizes

4 CPU cores, mobile GPU

8 CPU cores, midrange GPU

16 CPU cores per node, high end GPUs

32 cores, No GPU

Same code, different devices, maximum performance

ANACONDA®

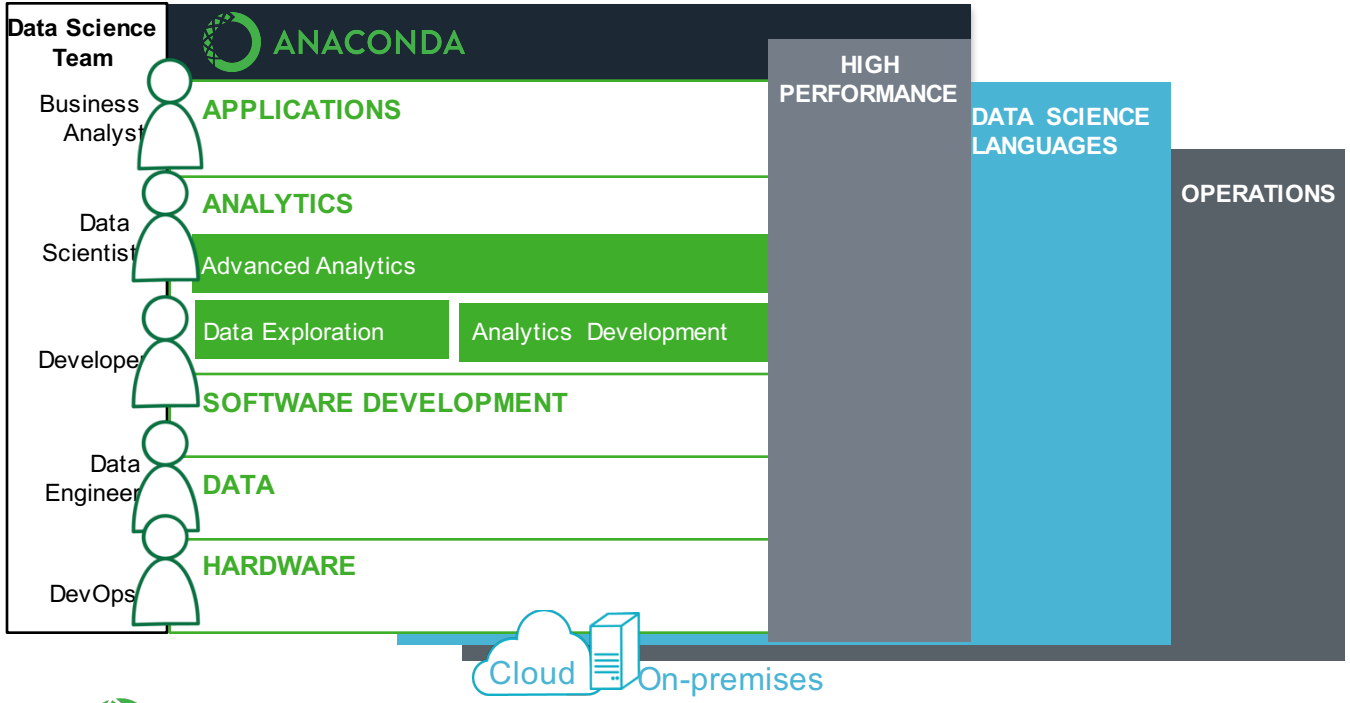
Why is Python Great for Data Science?

- Easy to use
  - Simple easy to read & write syntax
- "Batteries included"
  - Ships with lots of basic functions
- Innovation from Open Source
  - Open access to a huge variety of existing libraries and algorithms
- Can get high performance when you need it…

… with Anaconda

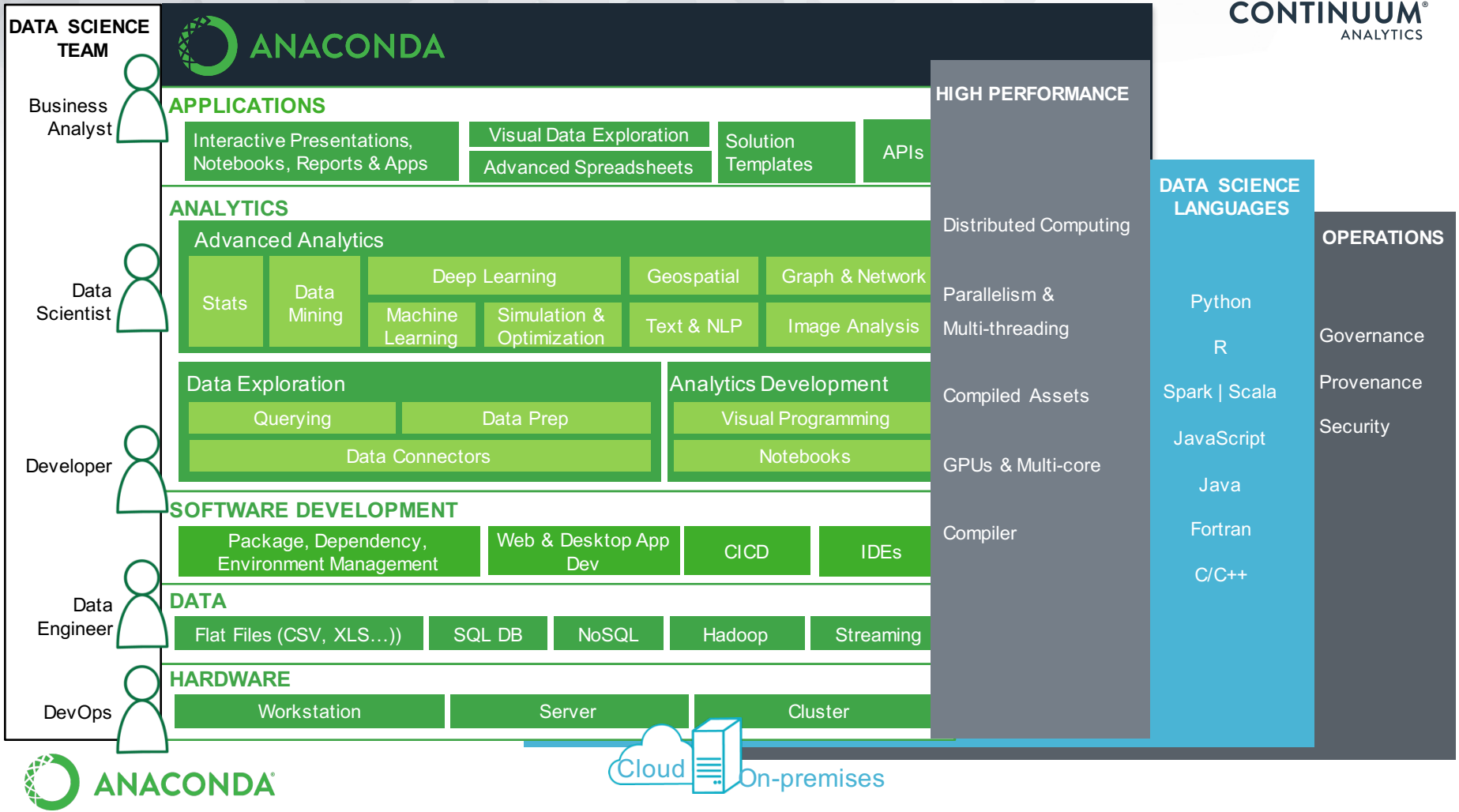
Introducing Anaconda

The Open Data Science Platform Powered by Python



- Enterprise Ready Platform
  - Simplify administration
  - Use modern data science
  - Collaborate with the entire team
  - Leverage modern architectures
  - Integrate data sources
  - Accelerate performance

ANACONDA®


DATA SCIENCE TEAM
Business Analyst
Data Scientist
Developer
Data Engineer
DevOps
ANACONDA
APPLICATIONS
Interactive Presentations, Notebooks, Reports & Apps
Visual Data Exploration
Advanced Spreadsheets
Solution Templates
APIs
ANALYTICS
Advanced Analytics
Stats
Data Mining
Deep Learning
Machine Learning
Simulation & Optimization
Geospatial
Graph & Network
Text & NLP
Image Analysis
Data Exploration
Querying
Data Prep
Data Connectors
Analytics Development
Visual Programming
Notebooks
SOFTWARE DEVELOPMENT
Package, Dependency, Environment Management
Web & Desktop App Dev
CICD
IDEs
DATA
Flat Files (CSV, XLS…))
SQL DB
NoSQL
Hadoop
Streaming
HARDWARE
Workstation
Server
Cluster
HIGH PERFORMANCE
Distributed Computing
Parallelism & Multi-threading
Compiled Assets
GPUs & Multi-core
Compiler
DATA SCIENCE LANGUAGES
Python
R
Spark | Scala
JavaScript
Java
Fortran
C/C++
OPERATIONS
Governance
Provenance
Security
Cloud
On-premises
ANACONDA
CONTINUUM ANALYTICS

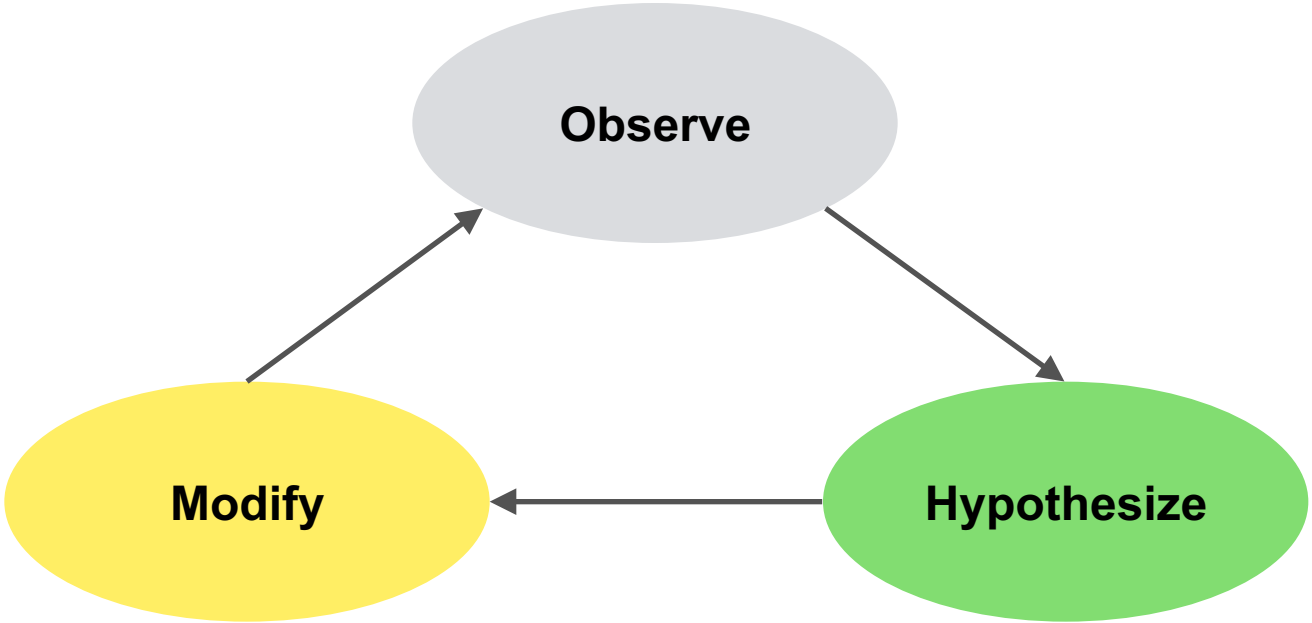
Tools for Performance

- Profiling: Find the bottlenecks
- Optimized Libraries: Solve the common problems
- Compilation: Accelerate your custom algorithms

CONTINUUM®
ANALYTICS

FINDING BOTTLENECKS WITH PROFILING

Optimization is the Scientific Process



Profiling: Observe your Algorithm

- A profiler records the amount of time each part of your program takes to execute.
- Usually record data at the function level, but some go down to the line level.
- We'll talk about both high and low level profilers.

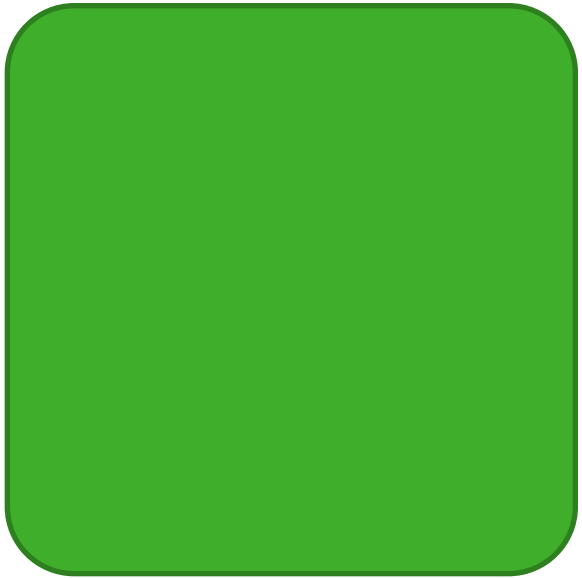
The Anaconda Profiler

CONTINUUM ANALYTICS

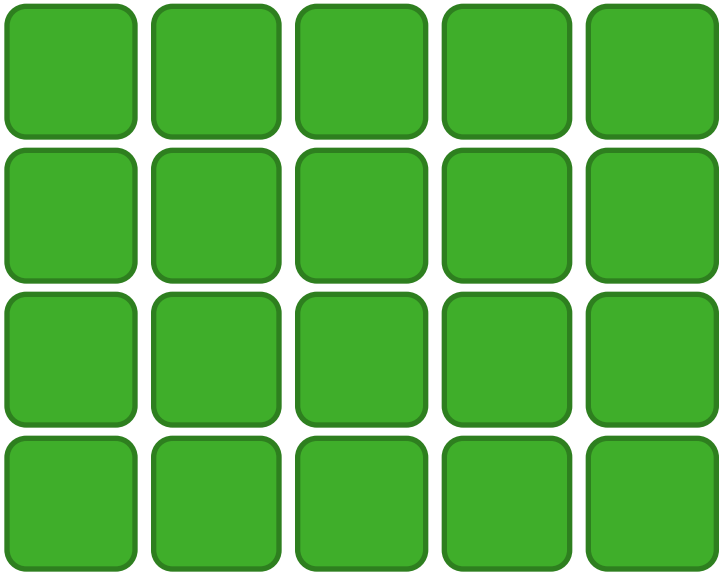
- Included with Anaconda Workgroup and Enterprise
- Extends Python profiler with “data profiling”
  - Record the data types and array sizes
  - Data types and sizes are critical for deciding how to optimize a function

ANACONDA

Data-Driven Optimization Strategies



Large Data: Look to more threads, SIMD, memory access patterns



Small Data: Look to batch calls, group data into larger blocks, caching

Visualization with the Anaconda Profiler

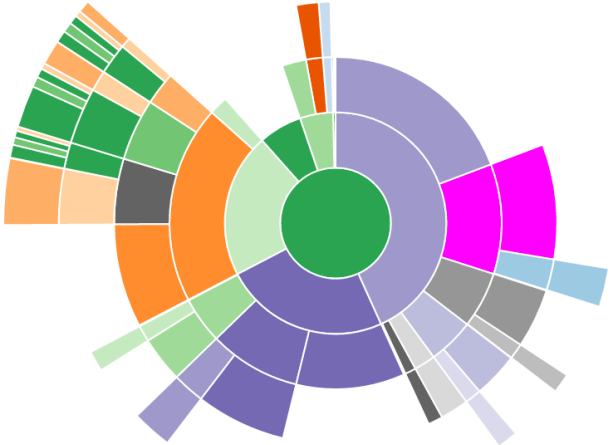
```
In [9]: profiler.plot(p)
```

Out[9]:

Reset

Call Stack

Style: Sunburst
Depth: 5
Cutoff: 1/1000



Name:
`Exponential.set_value(value:float64, force:bool)`
Cumulative Time:
`1.65 s (7.00 %)`
File:
`PyMCObjects.py`
Line:
`837`
Directory:
`/Users/stan/anaconda/envs/pymc/lib/python2.7/site-packages/pymc/`

CONTINUUM ANALYTICS

Visualization with the Anaconda Profiler

Search: linalg

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
150/50	0.006636	0.0001327	0.01559	0.0003117	linalg.py:532(cholesky(a:ndarray(dtype=float64, shape=(3, 3))))
100/50	0.00222	4.44e-05	0.00376	7.52e-05	linalg.py:139(_commonType())
100/50	0.001763	3.526e-05	0.002728	5.456e-05	linalg.py:106(_makearray(a:ndarray(dtype=float64, shape=(3, 3))))
200/100	0.001399	1.399e-05	0.001399	1.399e-05	linalg.py:111(isComplexType(t:type))
150/50	0.001087	2.174e-05	0.001087	2.174e-05	linalg.py:209(_assertNdSquareness())
100/50	0.000834	1.668e-05	0.000834	1.668e-05	linalg.py:124(_realType(t:type, default:NoneType))
100/50	0.000588	1.176e-05	0.000588	1.176e-05	linalg.py:198(_assertRankAtLeast2())
50	9.5e-05	1.9e-06	9.5e-05	1.9e-06	linalg.py:101(get_linalg_error_extobj(callback:function))

Showing 1 to 8 of 8 entries (filtered from 161 total entries)

ANACONDA

CONTINUUM®
ANALYTICS

INTEL TOOLS AND LIBRARIES FOR HIGH PERFORMANCE

Boost Python* Performance with Intel® Tools

Sergey Maidanov

Software Engineering Manager for Intel® Distribution for Python*

Performance Profiling with Intel® Vtune™ Amplifier

What makes Intel® VTune™ Amplifier special

Right tool for high performance application profiling at all levels

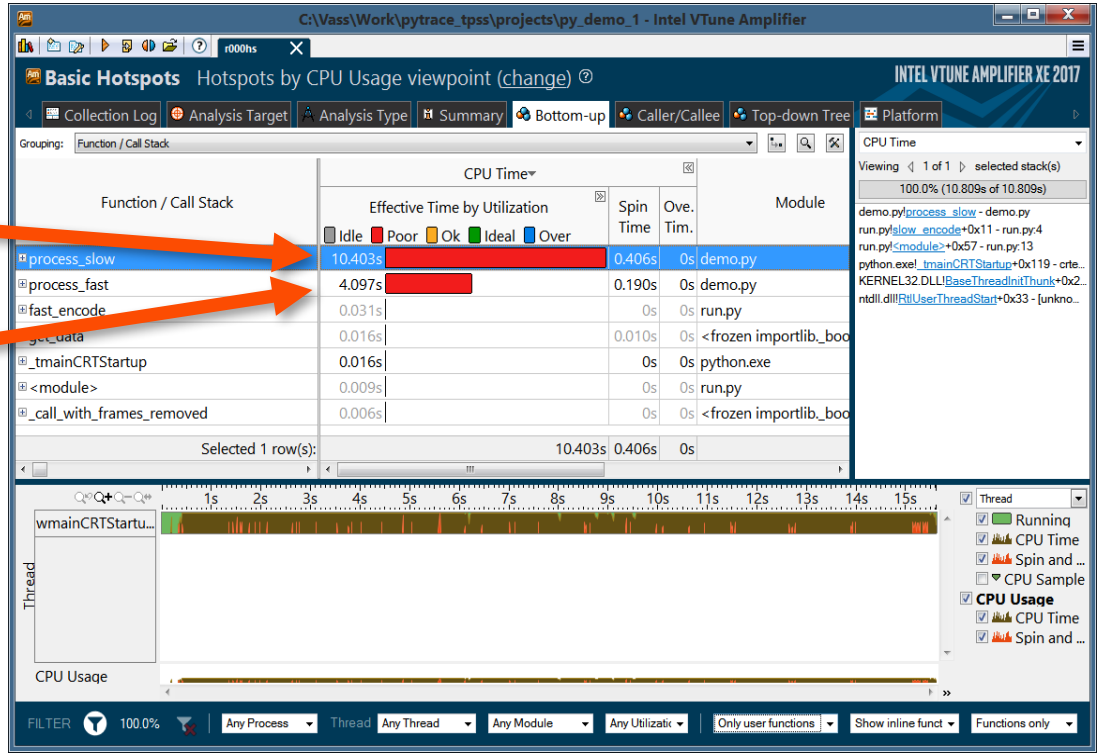
- Function-level and line-level hotspot analysis, down to disassembly
- Call stack analysis
- Low overhead
- Mixed-language, multi-threaded application analysis
- Advanced hardware event analysis for native codes (Cython, C++, Fortran) for cache misses, branch misprediction, etc.

Feature	cProfile	Line_profiler	Intel® VTune™ Amplifier
Profiling technology	Event	Instrumentation	Sampling, hardware events
Analysis granularity	Function-level	Line-level	Line-level, call stack, time windows, hardware events
Intrusiveness	Medium (1.3-5x)	High (4-10x)	Low (1.05-1.3x)
Mixed language programs	Python	Python	Python, Cython, C++, Fortran

Download Intel® VTune™ Amplifier 2017 Beta for evaluation

Encoder demo: Strings vs. Lists

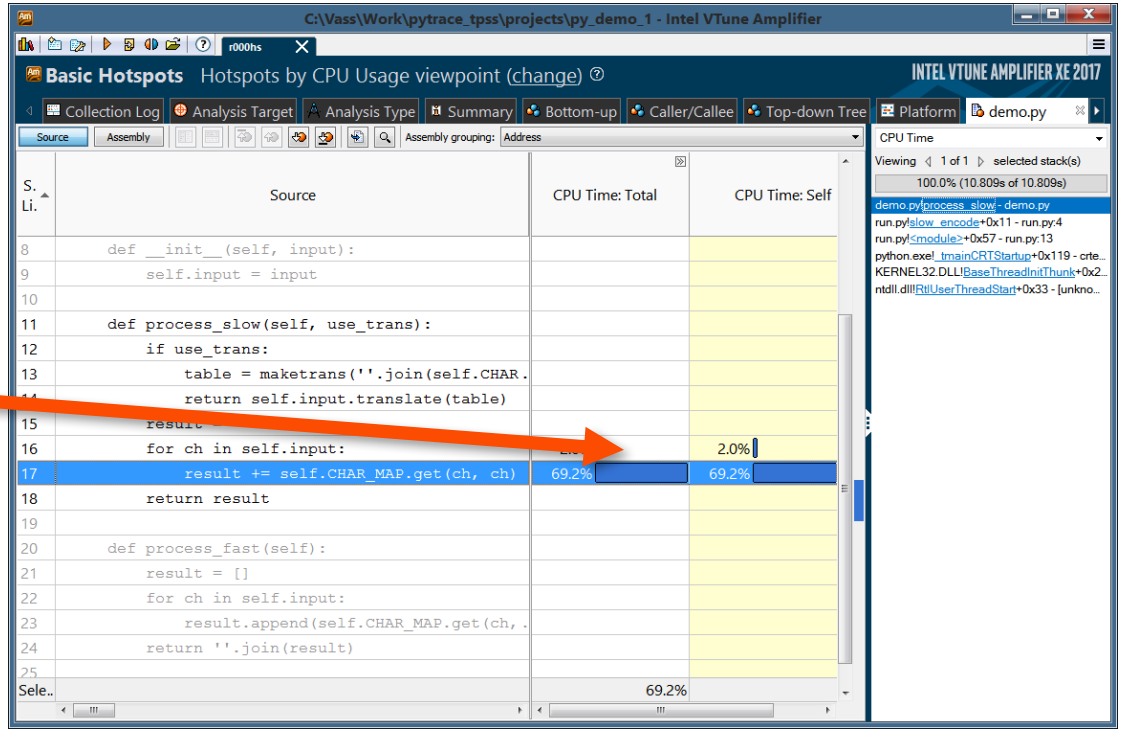
```
class Encoder:
    CHAR_MAP = {'a': 'b', 'b': 'c'}
    def __init__(self, input):
        self.input = input

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

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



Strings concatenation is >2x worse than appending to a list

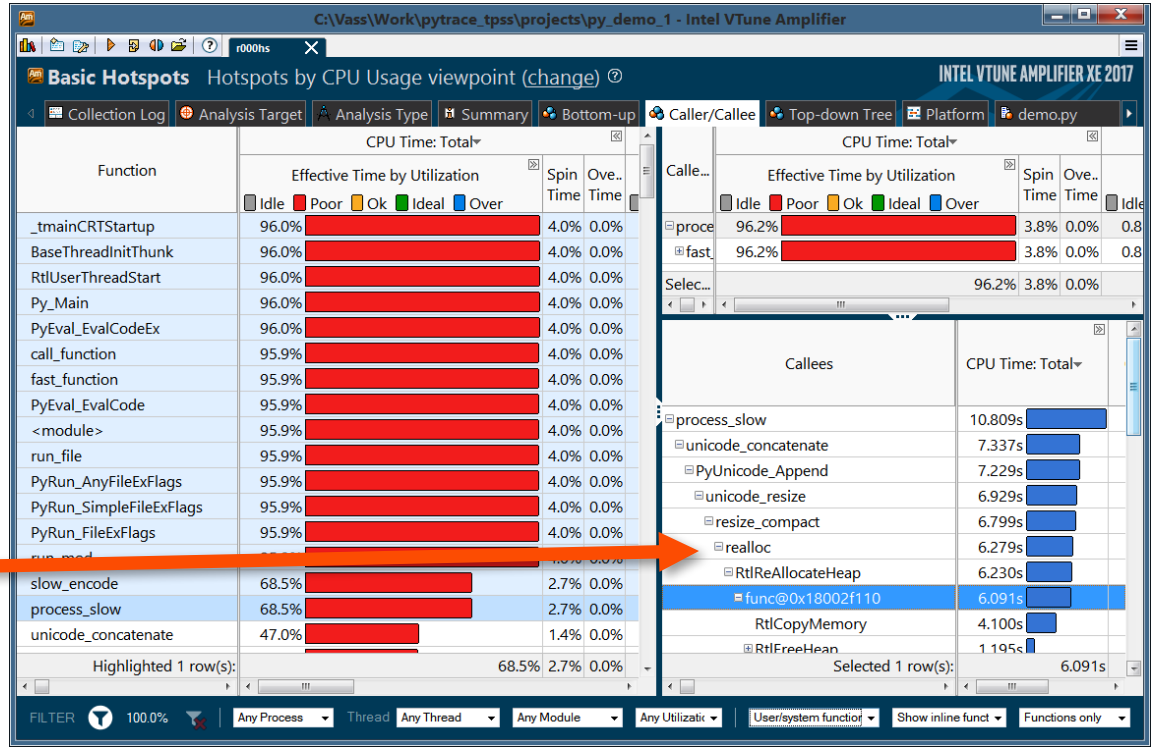
Encoder demo: Line-level statistics

70% time spent to retrieve a value from the dictionary and to concatenate it to the string



Encoder demo: What's in native code (call stack)

Each concatenation results in expensive memory reallocation



How Intel Enables Python Performance Through Libraries

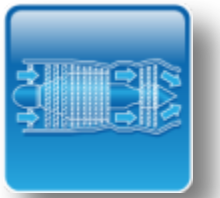
Family of Intel libraries optimized for IA

- Numerical computing, data analytics and data processing
  - Intel® Math Kernel Library
  - Intel® Data Analytics Acceleration Library
  - Intel® Integrated Performance Primitives
- Multi-threading
  - Intel® Treading Building Blocks
- Multi-node parallelism
  - Intel® MPI


Energy


Signal Processing

Financial Analytics


Engineering Design

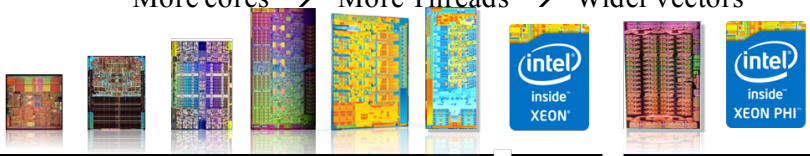

Digital Content Creation


Science & Research

Our approach

1. Enable hooks to optimized Intel libraries in the most popular numerical/data processing packages
   - NumPy, SciPy, Scikit-Learn, PyTables, Scikit-Image, MPI4Py, …
2. Available through Intel® Distribution for Python* and as Conda packages that can be installed in your Anaconda* distribution
   - Most optimizations eventually upstreamed to home open source projects and available through other Python distributions

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	~	Future Intel® Xeon® Processor[1]	Intel® Xeon Phi™ x100 Coprocessor	Intel® Xeon Phi™ x200 Processor & Coprocessor
Up to Core(s)	1	2	4	6	12	18	~	*Tbd*	57-61	TBD
Up to Threads	2	2	8	12	24	36		*tbd*	228-244	TBD
SIMD Width	128	128	128	128	256	256	~	512	512	512
Vector ISA	Intel® SSE3	Intel® SSE3	Intel® SSE4.2	Intel® AVX	Intel® AVX	Intel® AVX2		Intel® AVX-512	IMCI 512	Intel® AVX-512

How Intel Enables Python Performance Through Intel® MKL

Optimized mathematical building blocks

Intel® Math Kernel Library (Intel MKL)

Linear Algebra

- BLAS
- LAPACK
- ScaLAPACK
- Sparse BLAS
- Sparse Solvers
  - Iterative
  - PARDISO* SMP & Cluster

Up to 100x faster

Fast Fourier Transforms

- Multidimensional
- FFTW interfaces
- Cluster FFT

Up to 10x faster!

Vector Math

- Trigonometric
- Hyperbolic
- Exponential
- Log
- Power
- Root

Up to 10x faster!

Vector RNGs

- Multiple BRNG
- Support methods for independent streams creation
- Support all key probability distributions

Up to 60x faster!

Summary Statistics

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

And More

- Splines
- Interpolation
- Trust Region
- Fast Poisson Solver

Optimization Notice

Functional domain in this color accelerate respective NumPy, SciPy, etc. domain

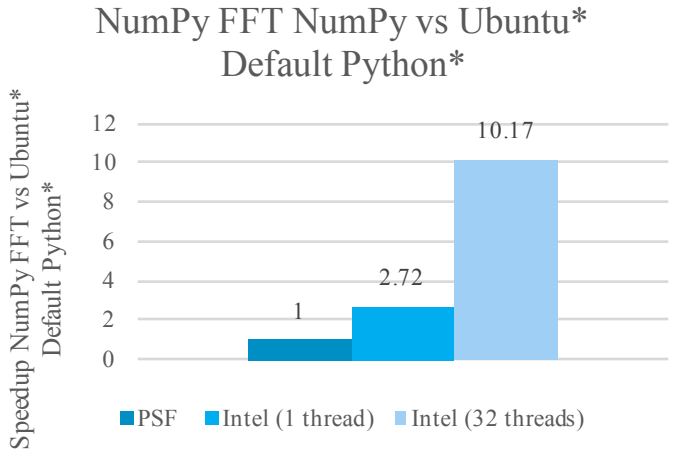
Configuration info: - Versions: Intel® Distribution for Python 2017 Beta, icc 15.0; Hardware: Intel® Xeon® CPU E5-2698 v3 @ 2.30GHz (2 sockets, 16 cores each, HT=OFF), 64 GB of RAM, 8 DIMMS of 8GB@2133MHz; Operating System: Ubuntu 14.04 LTS.

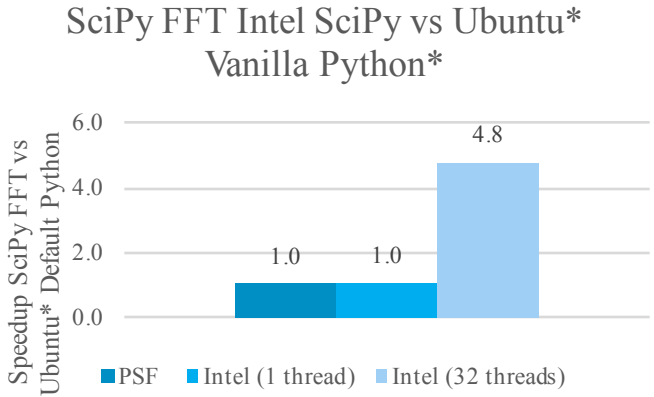
Optimized FFT show case

Intel® Math Kernel Library (Intel MKL)

- Original SciPy FFT implementation is about 2x faster than original NumPy FFT
- Intel engineers bridged NumPy and SciPy implementations via common layer and embedded MKL FFT calls, what measurably accelerates both NumPy and SciPy
  - NumPy and SciPy are computationally compatible
  - FFT descriptors caching applied for enhanced performance in repetitive and multidimensional FFT calculations

Available starting Intel® Distribution for Python* 2017 Beta



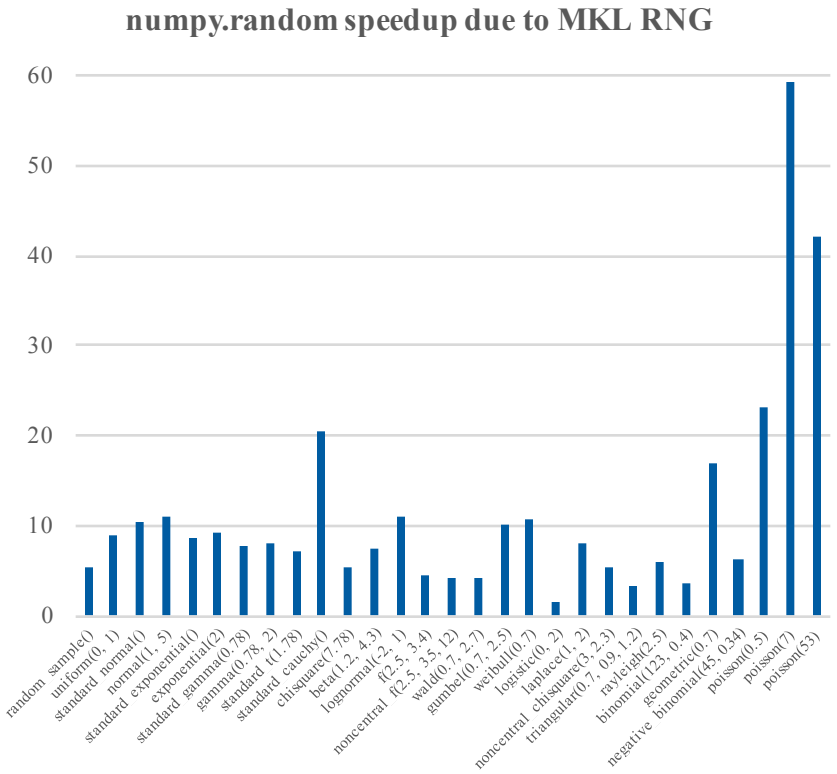


Configuration info: - Versions: Intel® Distribution for Python 2017 Beta, icc 15.0; Hardware: Intel® Xeon® CPU E5-2698 v3 @ 2.30GHz (2 sockets, 16 cores each, HT=OFF), 64 GB of RAM, 8 DIMMS of 8GB@2133MHz; Operating System: Ubuntu 14.04 LTS.

Optimized RNG show case

Intel® Math Kernel Library (Intel MKL)

- Implemented numpy.random in vector fashion to enable vector MKL RNG and VML calls
- Enabled multiple BRNG
- Enabled multiple distribution transformation methods



Initial data. Final data to be available in the update for Intel® Distribution for Python* 2017 Beta

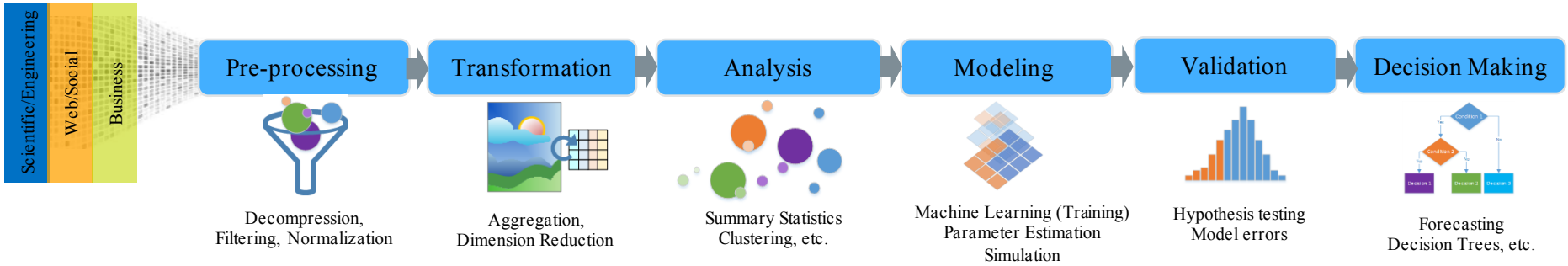
Configuration info: - Versions: Intel® Distribution for Python (internal build 28.4.2016), icc 15.0; Hardware: Intel® Xeon® CPU E5-2630 v3 @ 2.40GHz (16 cores), 32 GB; Operating System: Ubuntu 14.04 LTS.

Intel® Data analytics acceleration library

Optimized blocks for data analytics pipelines

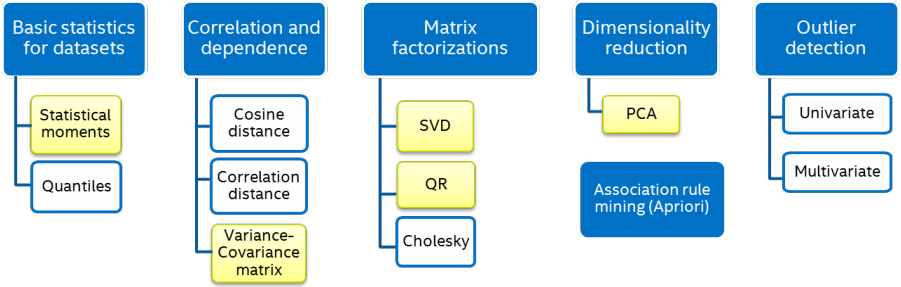
Intel® Data Analytics Acceleration Library (Intel DAAL)

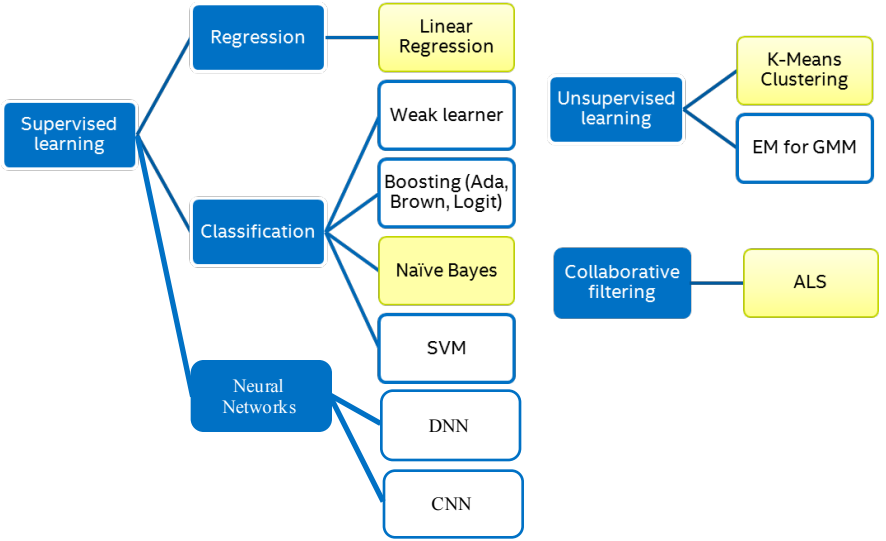
- Optimizes entire analytics pipeline, from data acquisition to training and scoring
  - Allows different stages to be executed on different devices, e.g. server and edge
- Bridges prototyping and production analytics
  - Multiple language interfaces C++, Java, Python
  - Runtime CPU detection, runs best from Intel® Atom to Intel® Xeon and Intel® Xeon Phi processor families
  - Code samples for MPI, Hadoop, Spark



Intel DAAL algorithms at a glance

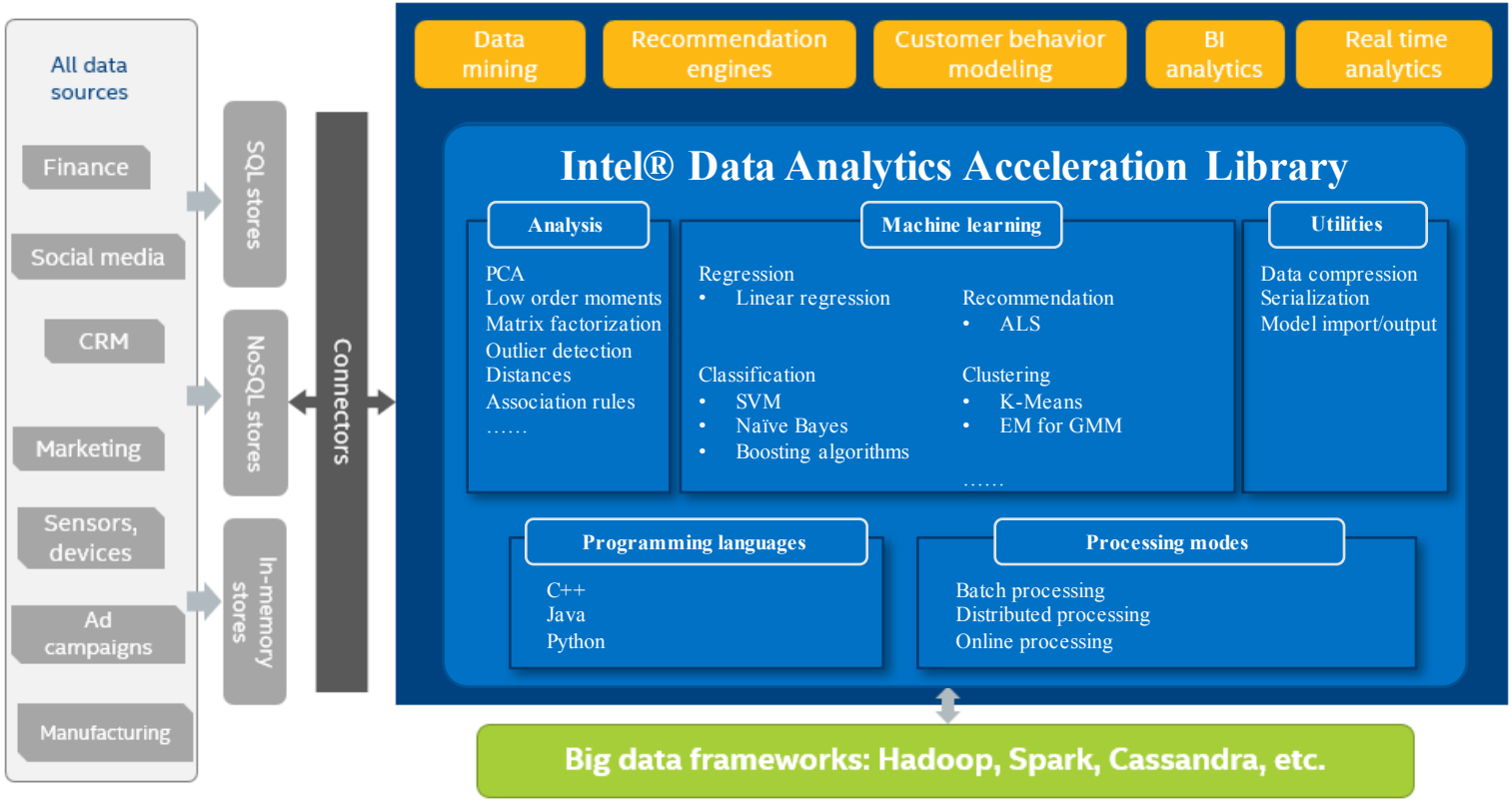
- Classic and modern data mining and machine learning algorithms
  - Supervised and unsupervised learning
  - Neural networks (DNN, CNN)
  - Batch, online, and distributed





Algorithms support streaming and distributed processing in the current release.

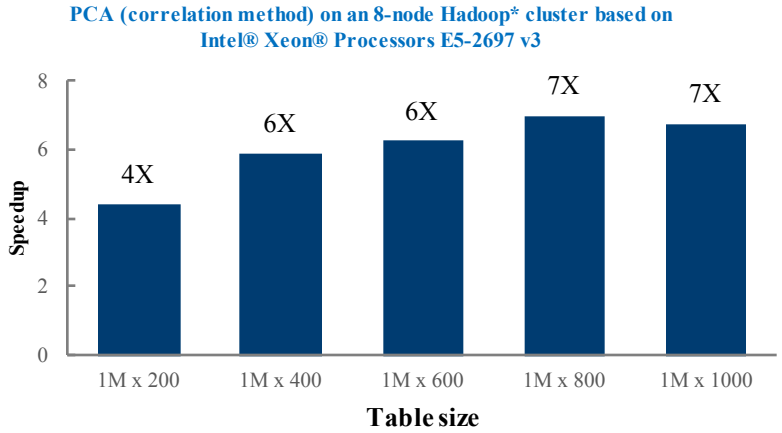
Where DAAL fits



How DAAL accelerates Python

1. Available as PyDAAL package
   - Pre-installed with Intel® Distribution for Python*
   - Through anaconda.org as Conda* package
   - Part of Intel® DAAL and Intel® Parallel Studio XE products
2. Accelerates Scikit-learn
   - Work in progress, more news soon

PCA Performance Boosts Using Intel® DAAL vs. Spark* MLLib



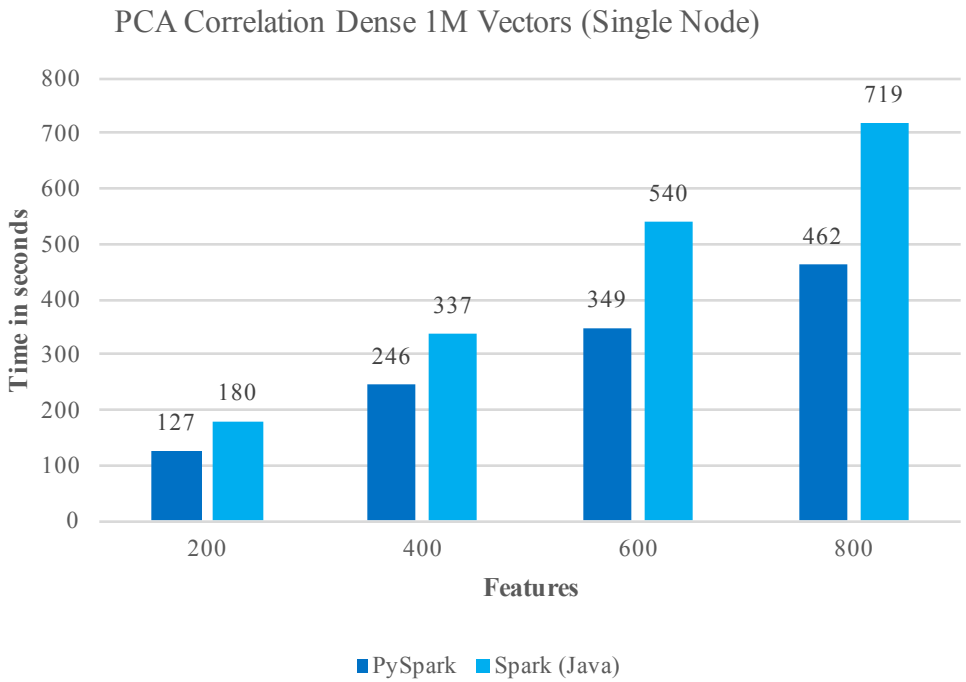
Configuration Info - Versions: Intel® Data Analytics Acceleration Library 2016, CDH v5.3.1, Apache Spark* v1.2.0; Hardware: Intel® Xeon® Processor E5-2699 v3, 2 Eighteen-core CPUs (45MB LLC, 2.3GHz), 128GB of RAM per node; Operating System: CentOS 6.6 x86_64.

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. * Other brands and names are the property of their respective owners. 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 .

PyDAAL and PySpark*

Work in progress to optimize PyDAAL distributed algorithms for PySpark*



Configuration Info - Versions: Intel® Data Analytics Acceleration Library 2016, CDH v5.3.1, Apache Spark* v1.2.0; Hardware: Intel® Xeon® Processor E5-2699 v3, 2 Eighteen-core CPUs (45MB LLC, 2.3GHz), 128GB of RAM per node; Operating System: CentOS 6.6 x86_64.

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. * Other brands and names are the property of their respective owners. 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 .

CONTINUUM®
ANALYTICS

SPEEDING UP PYTHON WITH COMPILATION

Interpreters vs. Compilers

Compiled Language	Interpreted Language
Must be translated from source code to machine code before execution	Can be executed line-by-line immediately and interactively
Application can be distributed standalone, without the compiler.	Application must be distributed with the interpreter.
Executes much faster	Executes more slowly
Examples: C, C++, FORTRAN	Examples: Python

Just-In-Time Compilation

- JIT compilers combine benefits of interpreted and compiled languages:
  - Allow interactive development
  - Translate algorithms during execution to machine code for speed
  - Can be applied to interpreted languages, like Python and Javascript

Numba: A Python JIT Compiler

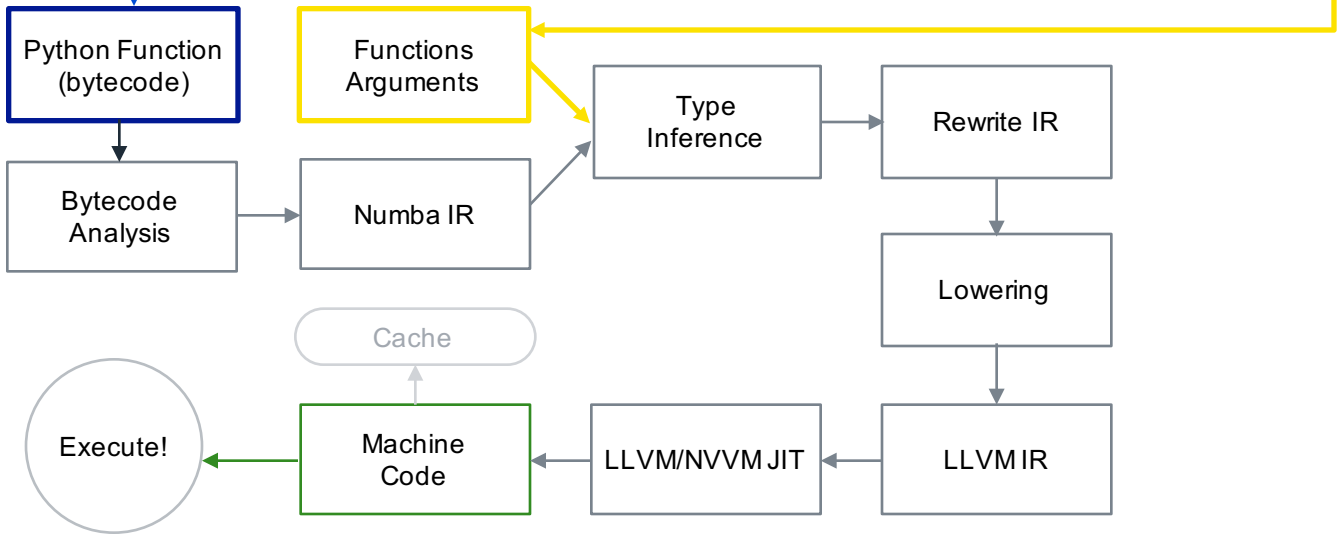
- Compile specific functions, leave the rest of your application unchanged.
- Designed for numerical workloads.
- Does not replace the standard interpreter: Keep using your favorite Python packages!
- Built-in support for NumPy arrays.

Supported Platforms

OS	HW	SW
Windows (7 and later)	32 and 64-bit x86 CPUs	Python 2 and 3
OS X (10.9 and later)	CUDA-capable NVIDIA GPUs	NumPy 1.7 through 1.11
Linux (~RHEL 5 and later)	HSA-capable AMD GPUs	

How Does Numba Work?

```
@jit
def do_math(a, b):
    …
>>> do_math(x, y)
```



Example: Filter an array

In [87]:
```
@jit(nopython=True)
def nan_compact(x):
    out = np.empty_like(x)
    out_index = 0
    for element in x:
        if not np.isnan(element):
            out[out_index] = element
            out_index += 1
    return out[:out_index]
```

In [88]:
```
a = np.random.uniform(size=10000)
a[a < 0.2] = np.nan
np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])
```

In [89]:
```
%timeit a[~np.isnan(a)]
%timeit nan_compact(a)
```

```
10000 loops, best of 3: 52 µs per loop
100000 loops, best of 3: 19.6 µs per loop
```

Example: Filter an array

```
In [87]: @jit(nopython=True)
         def nan_compact(x):
             out = np.empty_like(x)
             out_index = 0
             for element in x:
                 if not np.isnan(element):
                     out[out_index] = element
                     out_index += 1
             return out[:out_index]
```

Numba decorator (nopython=True not required)

Array Allocation

Looping over ndarray x as an iterator

Using numpy math functions

Returning a slice of the array

```
In [88]: a = np.random.uniform(size=10000)
         a[a < 0.2] = np.nan
         np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])
```

```
In [89]: %timeit a[~np.isnan(a)]
         %timeit nan_compact(a)

10000 loops, best of 3: 52 µs per loop
100000 loops, best of 3: 19.6 µs per loop
```

2.7x Speedup

CONTINUUM®
ANALYTICS

CONCLUSION

Anaconda

Accelerating Adoption of Python for Enterprise

ANACONDA®

ENTERPRISE DATA INTEGRATION With optimized connectors & out-of-core processing	NumPy & Pandas	Numba	**PERFORMANCE** With compiled Python for lightning fast execution
COLLABORATIVE NOTEBOOKS With publication, authentication, & search	Jupyter/ IPython	Bokeh	**VISUAL APPS** for interactivity and streaming data
Parallel Computing Scaling up Python analytics on your cluster	Dask	Conda	**SECURE & ROBUST REPOSITORY** for interactivity and streaming data

CONTINUUM® ANALYTICS

Anaconda Subscriptions

ANACONDA	ANACONDA PRO	ANACONDA WORKGROUP	ANACONDA ENTERPRISE
Open Source Modern Analytics Platform Powered by Python	Anaconda with Support & Indemnification	Anaconda with High Performance and Team Collaboration	Anaconda with Scalable High Performance and Team Collaboration
FREE FOREVER	Starting at **$10,000** per year up to 10 users + $1,000 per year for additional user	Starting at **$30,000** per year up to 10 users More than 10 users? Contact us for a quote.	Starting at **$60,000** per year up to 10 users More than 10 users? Contact us for a quote.
Community Support	Enterprise Support	Enterprise Support	Enterprise Support with dedicated Customer Support Rep.
DOWNLOAD	**LEARN MORE**	**CONTACT US**	**CONTACT US**

Intel Optimization Tools and Libraries

- Intel® Distribution for Python Beta:
  - https://software.intel.com/en-us/python-distribution
- Intel® VTune™ Amplifier 2017 – Mixed mode profiling Beta:
  - https://software.intel.com/en-us/python-profiling
- Intel Software Development tools and libraries:
  - https://software.intel.com/en-us/tools-by-segment/technical-enterprise
- Community licensing (free) Libraries:
  - https://software.intel.com/sites/campaigns/nest/

CONTINUUM ANALYTICS

Solutions For Python Performance Optimization

- **Download Anaconda** for free at Continuum.io/downloads
- **Engage with us** to help you speed up and optimize your Python at sales@Continuum.io
- **Take Performance Optimization Deep Dive** course at Continuum.io/training/
- **See Anaconda in action** at Continuum.io/webinars
- **Read** our whitepapers at Continuum.io/whitepapers

ANACONDA

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