INTEL® DISTRIBUTION FOR PYTHON*

intel Software

Intel® Distribution for Python* 2017

Advancing Python performance closer to native speeds

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

- Prebuilt, optimized for numerical computing, data analytics, HPC
- Drop in replacement for your existing Python. No code changes required

High performance with multiple optimization techniques

- Accelerated NumPy*/SciPy*/Scikit-Learn* with Intel® MKL
- Data analytics with pyDAAL, enhanced thread scheduling with TBB, Jupyter* Notebook interface, Numba*, Cython*
- Scale easily with optimized MPI4Py and Jupyter notebooks

Faster access to latest optimizations for Intel architecture

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

INSTALLING INTEL® DISTRIBUTION FOR PYTHON* 2017

Stand-alone installer and anaconda.org/intel

Linux

Windows*

OS X*

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

OR

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

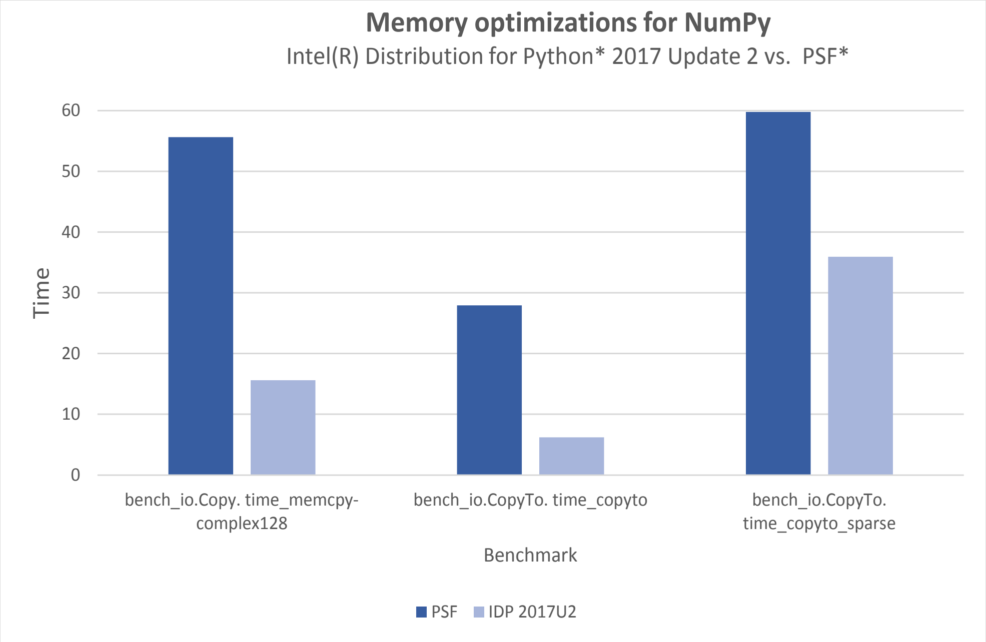
```
docker pull intelpython/intelpython3_full
```

2017 UPDATE 2 CHANGES

- Scikit-learn* accelerated with Intel® Data Analytics Acceleration Library (DAAL) for faster machine learning
- Increased optimizations on Fast Fourier Transforms (FFT) for NumPy* and SciPy* FFTs
- Changes in NumPy* to arithmetic and transcendental functions via umath optimizations and vectorization (AVX2, AVX-512 with MKL), can utilize multiple cores, memory management
- pyDAAL extensions for neural networks, advanced tensor inputs, distributed computing primitives

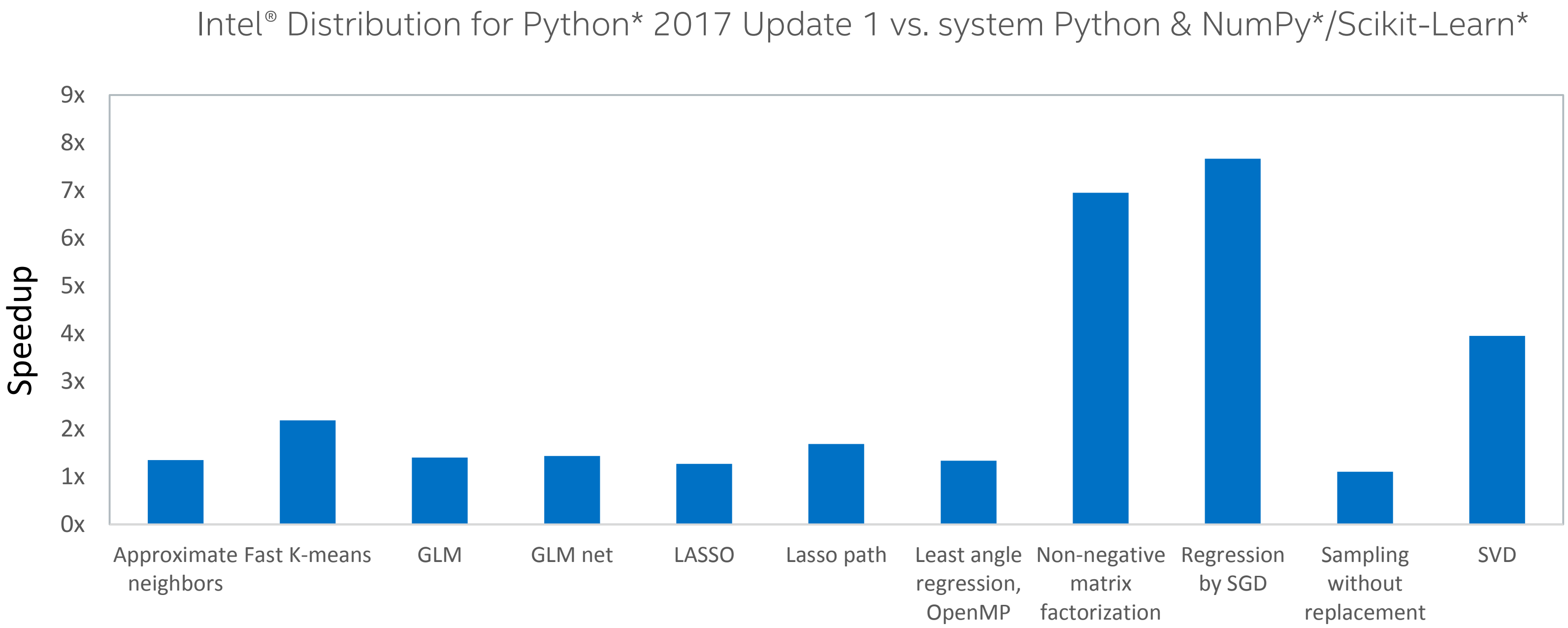
MEMORY OPTIMIZATIONS FOR NUMPY* ARRAYS

- Optimized array allocation/reallocation, copy/move
  - Memory alignment and data copy vectorization & threading



SCIKIT-LEARN* OPTIMIZATIONS WITH INTEL® MKL

Speedups of Scikit-Learn* Benchmarks (2017 Update 1)

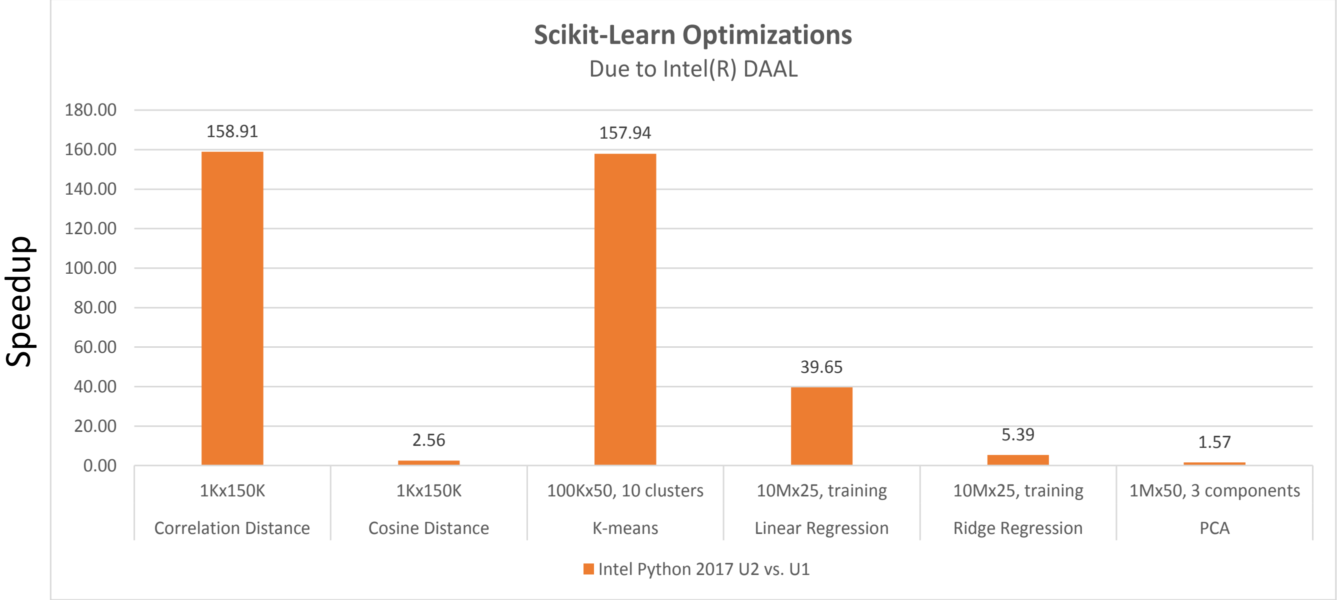


System info: 32x Intel® Xeon® CPU E5-2698 v3 @ 2.30GHz, disabled HT, 64GB RAM; Intel® Distribution for Python* 2017 Gold; Intel® MKL 2017.0.0; Ubuntu 14.04.4 LTS; Numpy 1.11.1; scikit-learn 0.17.1. See Optimization Notice.

MORE SCIKIT-LEARN* OPTIMIZATIONS WITH INTEL® DAAL

Speedups of Scikit-Learn* Benchmarks (2017 Update 2)

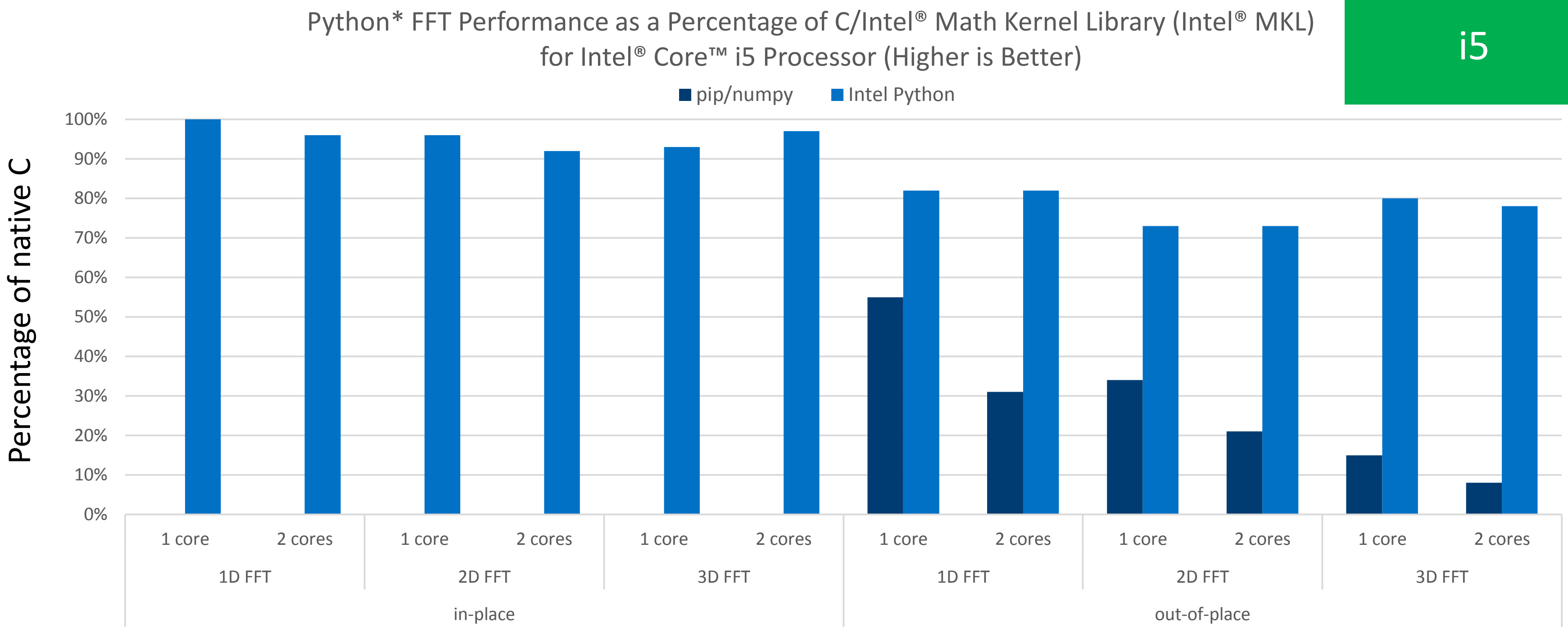
- Accelerated key Machine Learning algorithms with Intel® DAAL
  - Distances, K-means, Linear & Ridge Regression, PCA
  - Up to 160x speedup on top of MKL initial optimizations



FFT ACCELERATIONS WITH INTEL® DISTRIBUTION FOR PYTHON*

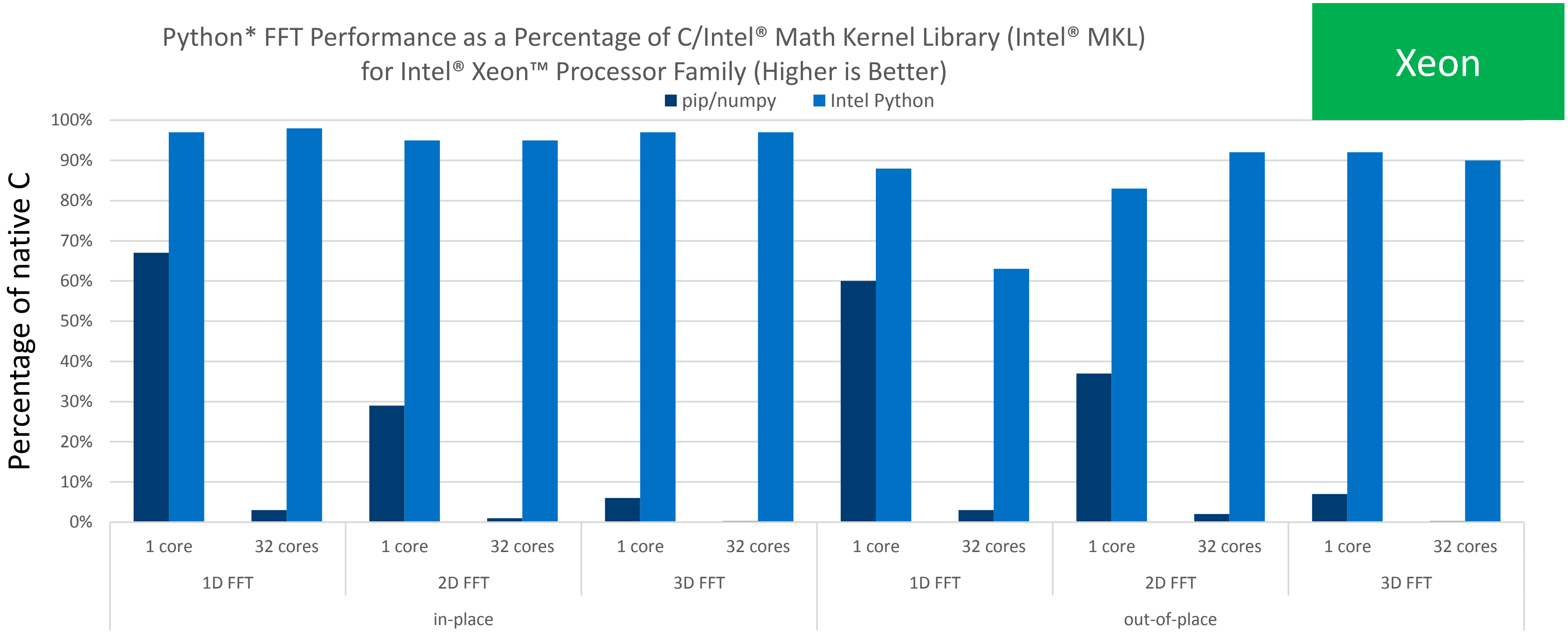
FFT Accelerations on i5 processors (2017 Update 2)

i5



FFT ACCELERATIONS WITH INTEL® DISTRIBUTION FOR PYTHON*

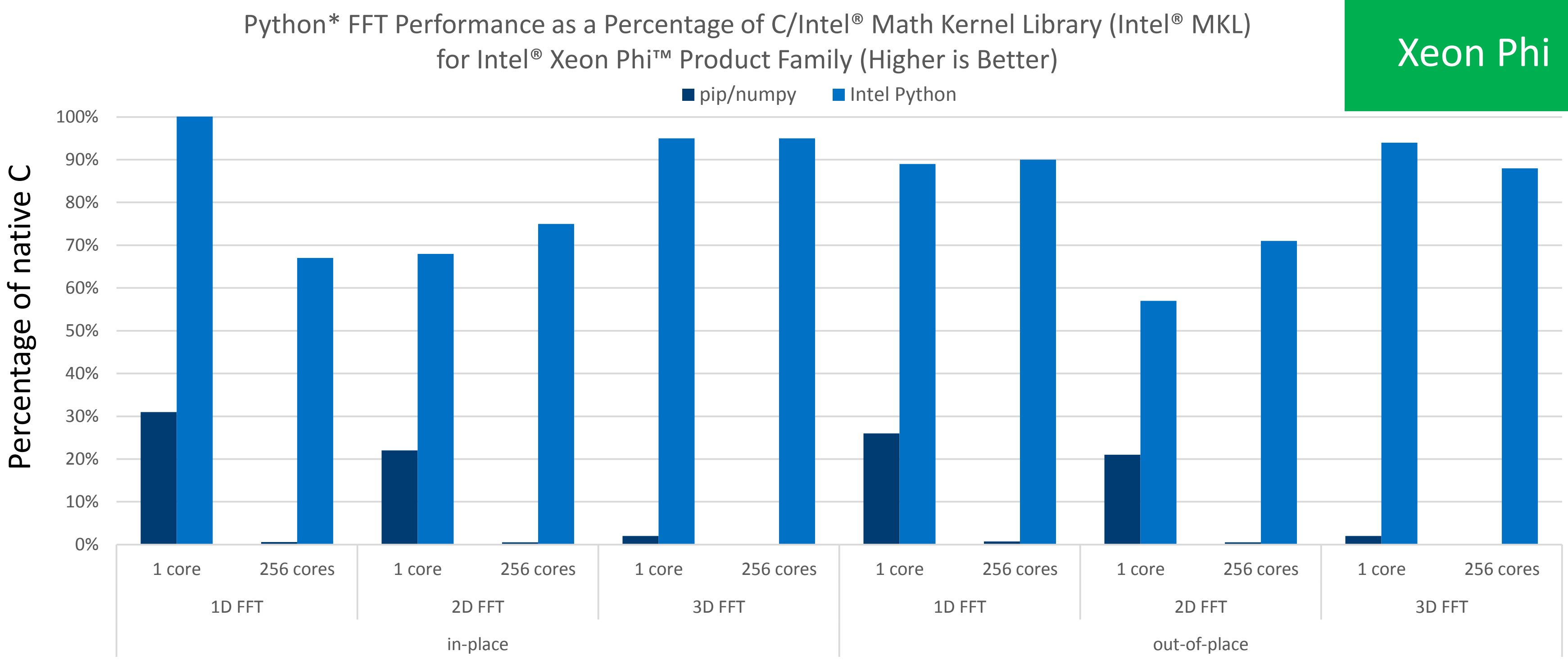
FFT Accelerations on Xeon processors (2017 Update 2)



FFT ACCELERATIONS WITH INTEL® DISTRIBUTION FOR PYTHON*

FFT Accelerations on Xeon Phi processors (2017 Update 2)

Xeon Phi

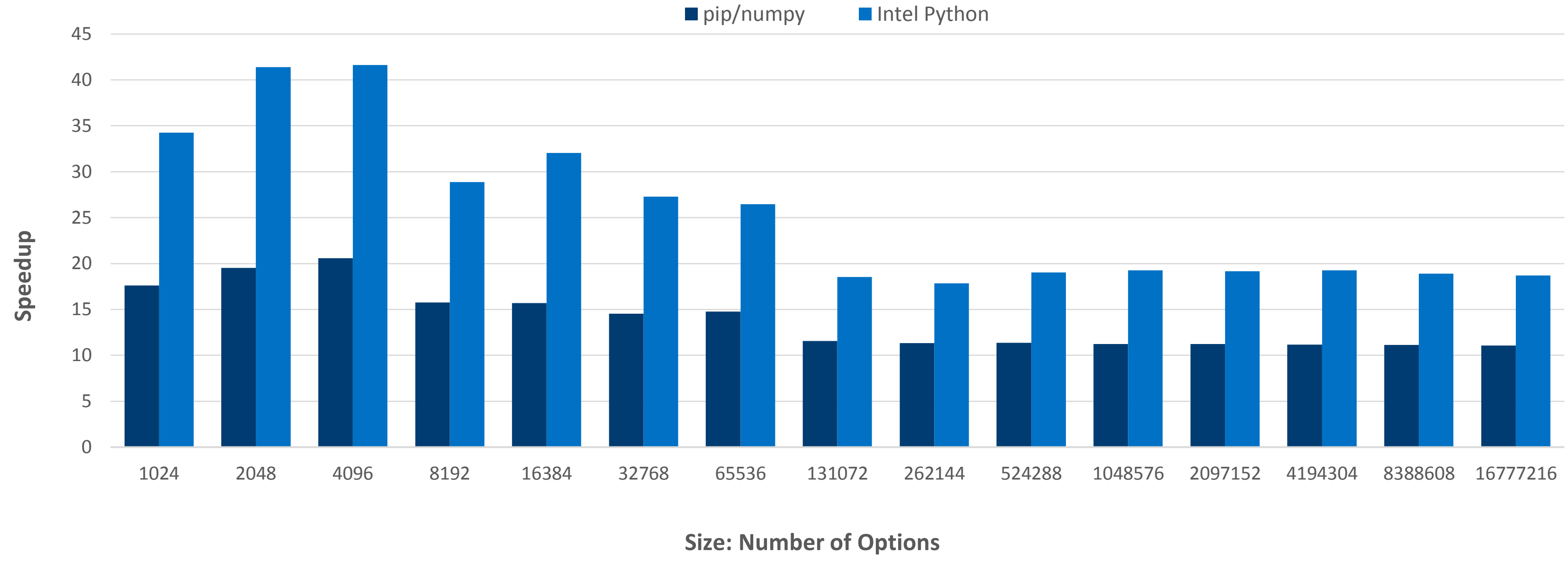


BLACK SCHOLES* BENCHMARKS

Black Scholes algorithm on i5 processors (2017 Update 2)

i5

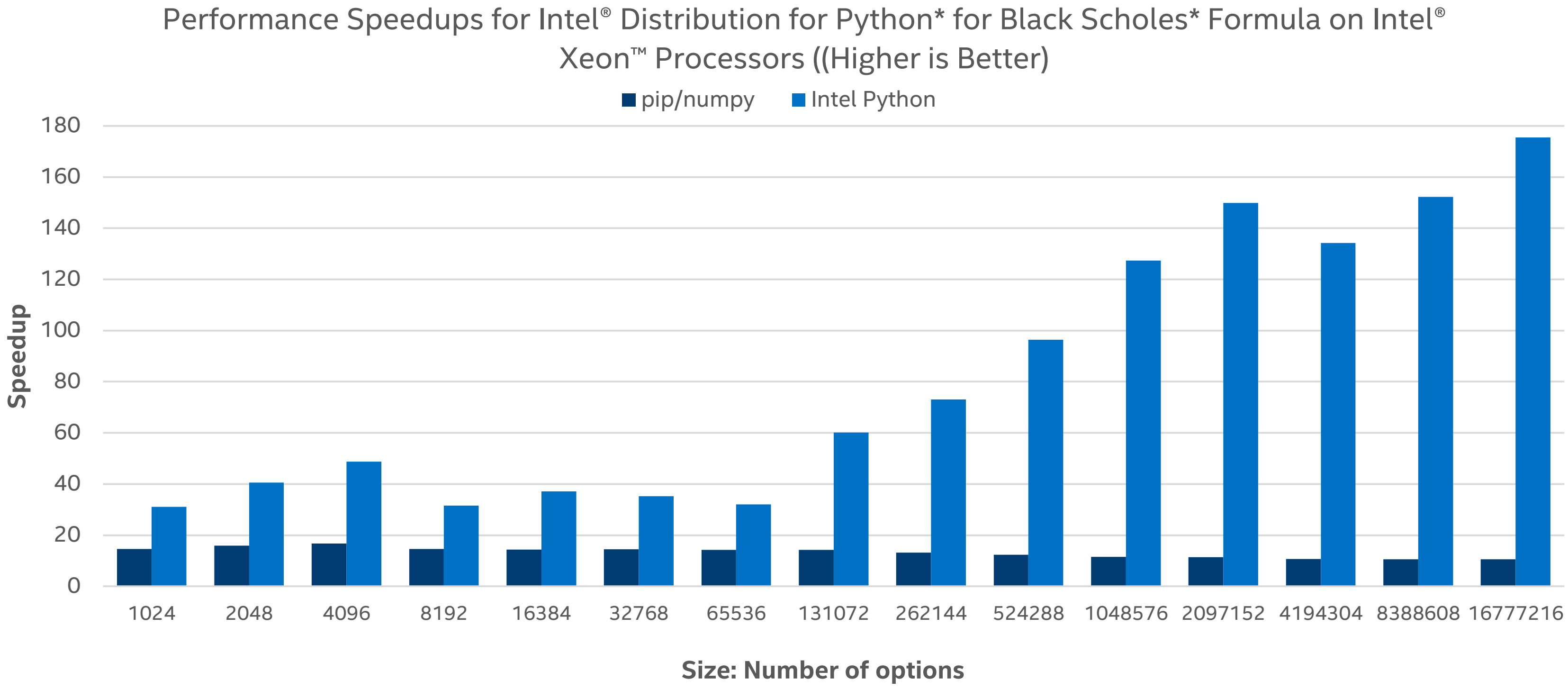
Performance Speedups for Intel® Distribution for Python* for Black Scholes* Formula on Intel® Core™ i5 Processor (Higher is Better)



BLACK SCHOLES* BENCHMARKS

Xeon

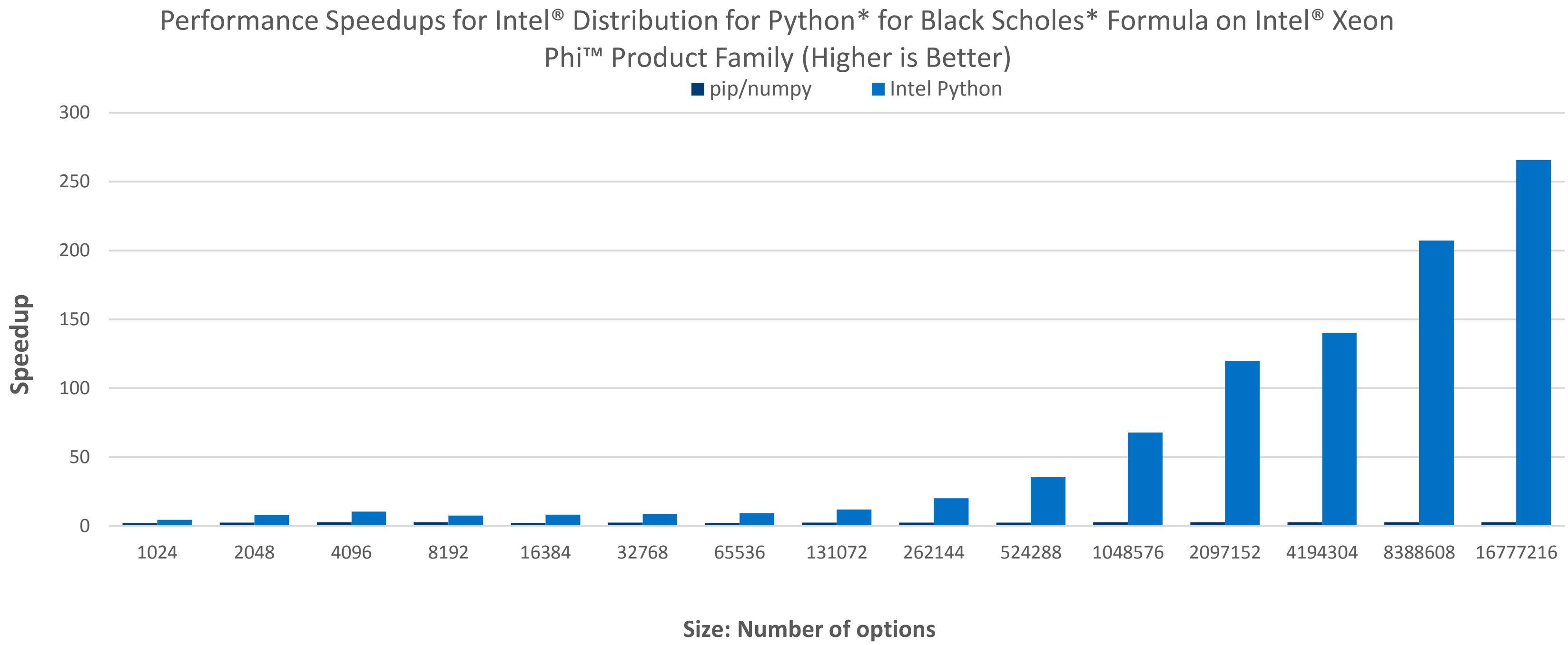
Black Scholes algorithm on Xeon processors (2017 Update 2)



BLACK SCHOLES* BENCHMARKS

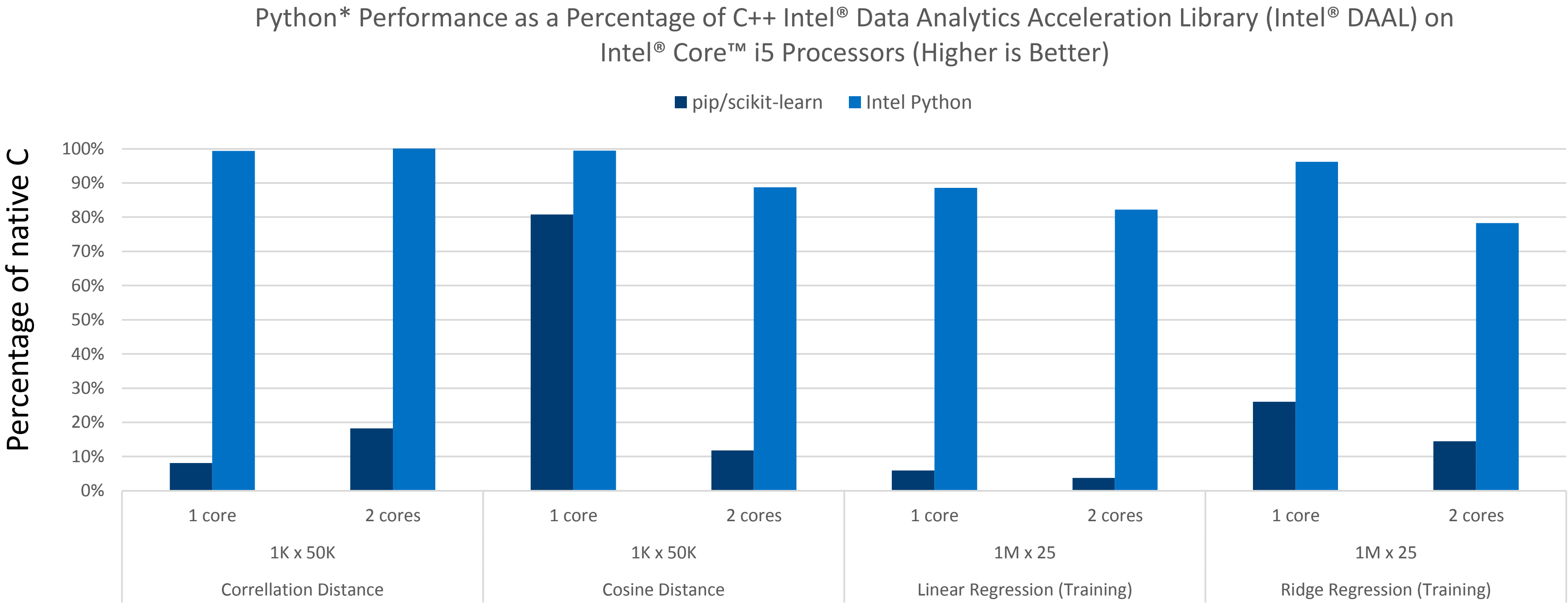
Black Scholes algorithm on i5 processors (2017 Update 2)

Xeon Phi



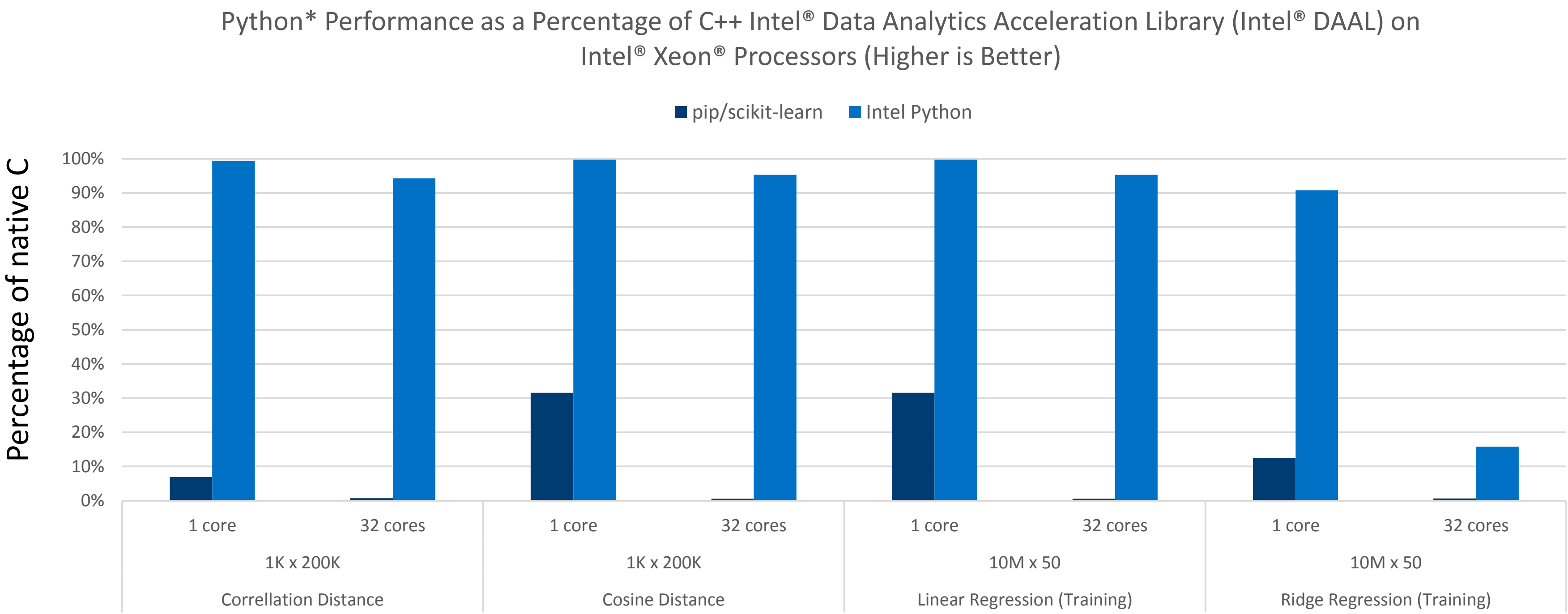
SCIKIT-LEARN* BENCHMARKS

i5



SCIKIT-LEARN* BENCHMARKS

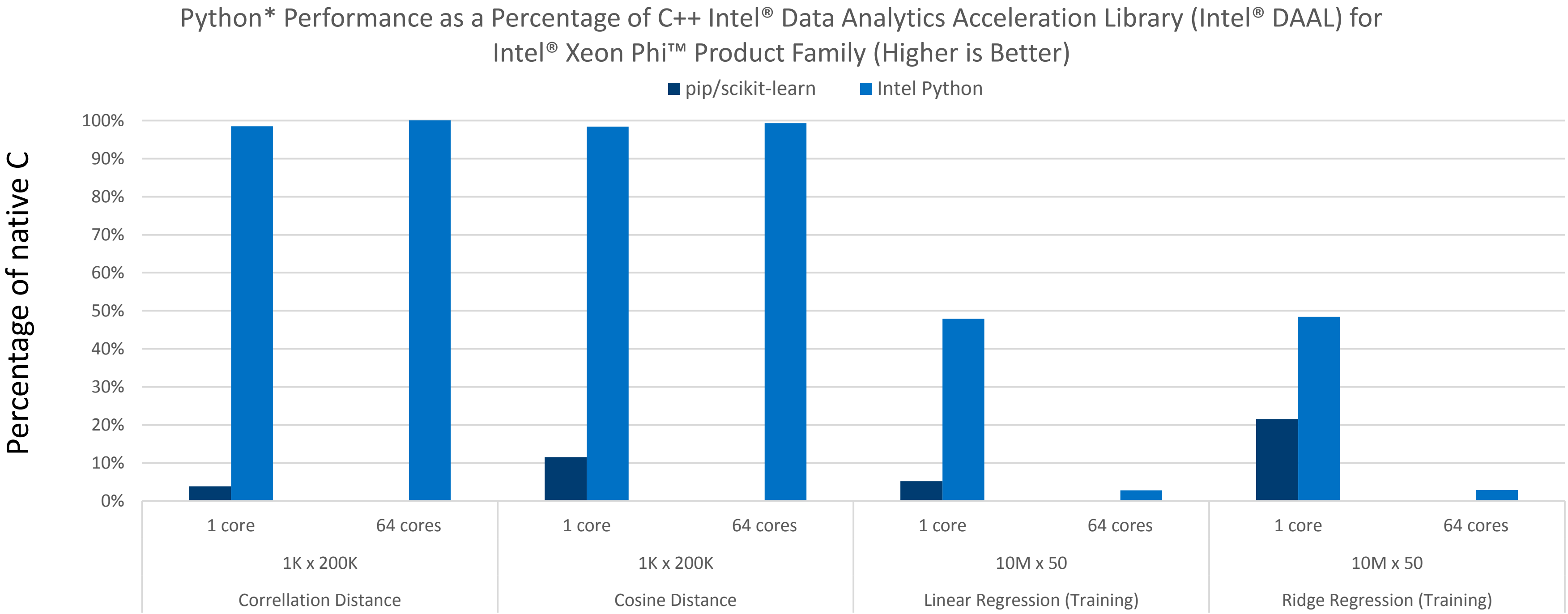
Xeon



For more complete information about compiler optimizations, see our Optimization Notice.

SCIKIT-LEARN* BENCHMARKS

Xeon Phi



CONFIGURATION INFORMATION

Software

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

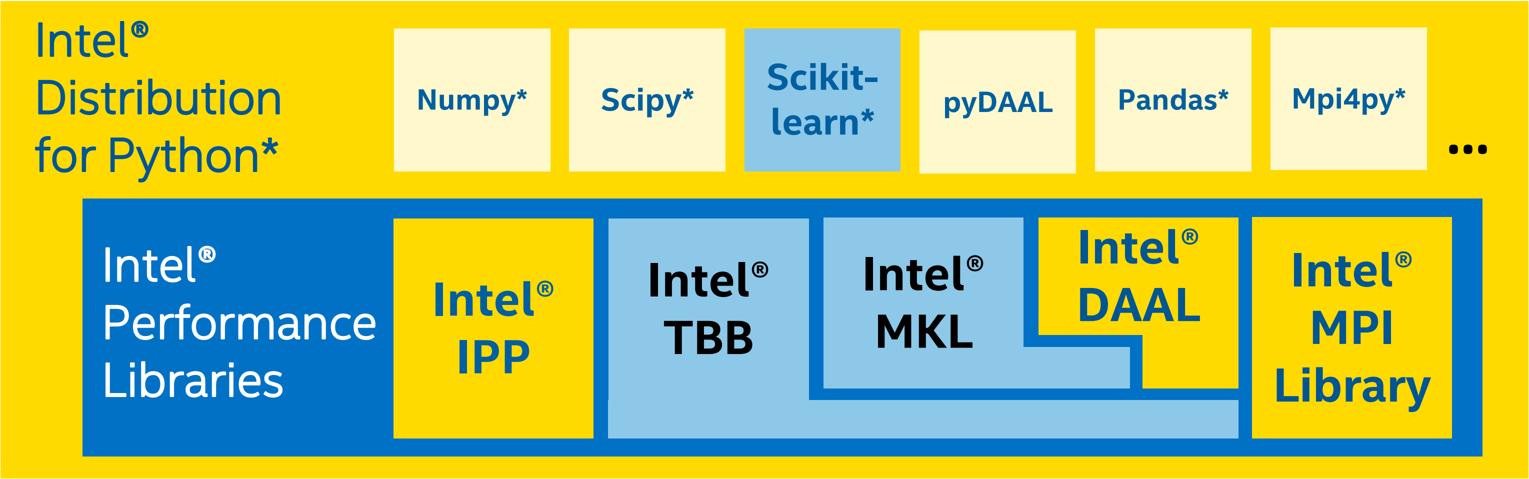
Hardware

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

Modifications

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

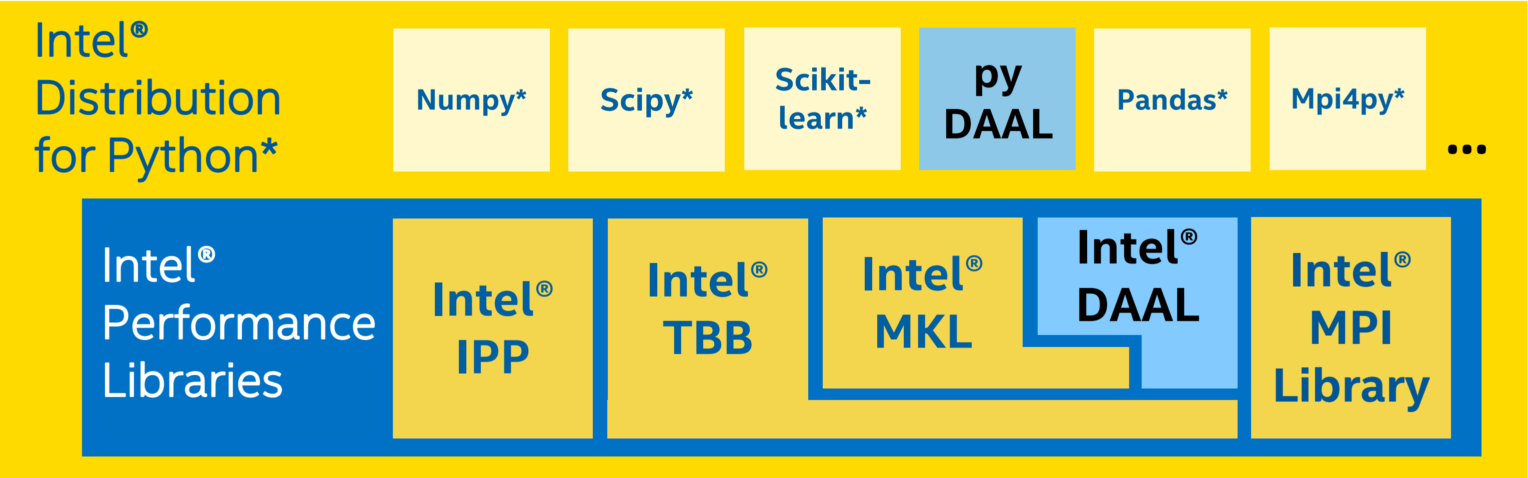
INTEL PYTHON LANDSCAPE



FASTER DATA ANALYTICS AND MACHINE LEARNING WITH PYDAAL

Software

INTEL PYTHON LANDSCAPE



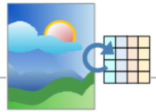
INTEL® DAAL: HETEROGENEOUS ANALYTICS

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

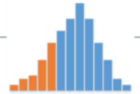
- Targets both data centers (Intel® Xeon® and Intel® Xeon Phi™) and edge-devices (Intel® Atom™)
- Perform analysis close to data source (sensor/client/server) to optimize response latency, decrease network bandwidth utilization, and maximize security
- Offload data to server/cluster for complex and large-scale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

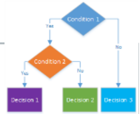
Pre-processing → **Transformation** → **Analysis** → **Modeling** → **Validation** → **Decision Making**

(De-)Compression
(De-)Serialization



PCA
Statistical moments
Quantiles
Variance matrix
QR, SVD, Cholesky
Apriori
Outlier detection





Regression
- Linear
- Ridge

Classification
- Naïve Bayes
- SVM
- Classifier boosting
- kNN

Clustering
- Kmeans
- EM GMM

Collaborative filtering
- ALS

Neural Networks

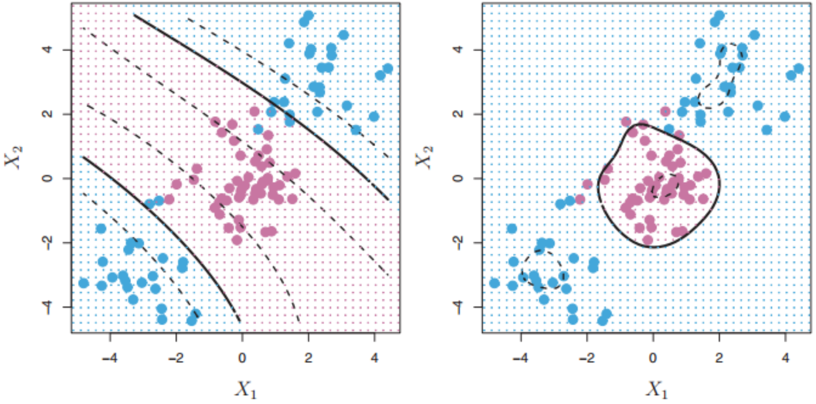
CLASSIFICATION

Problems

- An emailing service provider wants to build a spam filter for the customers
- A postal service wants to implement handwritten address interpretation

Solution: Support Vector Machine (SVM)

- Works well for non-linear decision boundary
- Two kernel functions are provided:
  - Linear kernel
  - Gaussian kernel (RBF)
- Multi-class classifier
  - One-vs-One



Source: Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani. (2014). *An Introduction to Statistical Learning.* Springer

PYDAAL EXAMPLE – SUPPORT VECTOR MACHINE

Model training

```
from daal.algorithms.svm import training, prediction
import daal.algorithms.kernel_function.linear
import daal.algorithms.classifier.training
kernel = kernel_function.linear.Batch()

def trainModel():
  dataSource   = FileDataSource('train_data.csv', …)
  labelsSource = FileDataSource('train_labels.csv', …)
  dataSource.loadDataBlock()
  labelsSource.loadDataBlock()

  algorithm = training.Batch()

  algorithm.parameter.kernel    = kernel
  algorithm.parameter.cacheSize = 600000000
  algorithm.input.set(classifier.training.data,
                      dataSource.getNumericTable())
  algorithm.input.set(classifier.training.labels,
                      labelsSource.getNumericTable())
  return algorithm.compute()
```

Kernel function to use with SVM algorithm

Initialize data source to retrieve data from CSV files

Retrieve the data from the input files

Create an algorithm object to train the SVM model

Set the parameters of the algorithm

Pass the training data set and labels to the algorithm

Build the SVM model

PYDAAL EXAMPLE – SUPPORT VECTOR MACHINE

Model-based prediction

```
def testModel(trainingResult):
  dataSource    = FileDataSource('test_data.csv', …)
  dataSource.loadDataBlock()

  algorithm = prediction.Batch()

  algorithm.parameter.kernel = kernel

  svmModel = trainingResult.get(classifier.training.model)
  algorithm.input.setTable(classifier.prediction.data,
                           dataSource.getNumericTable())
  algorithm.input.setModel(classifier.prediction.model, svmModel)

  return algorithm.compute()
```

Initialize data source to retrieve data from CSV file and retrieve the data from the input file

Create an algorithm object to predict the results

Set the parameters of the algorithm

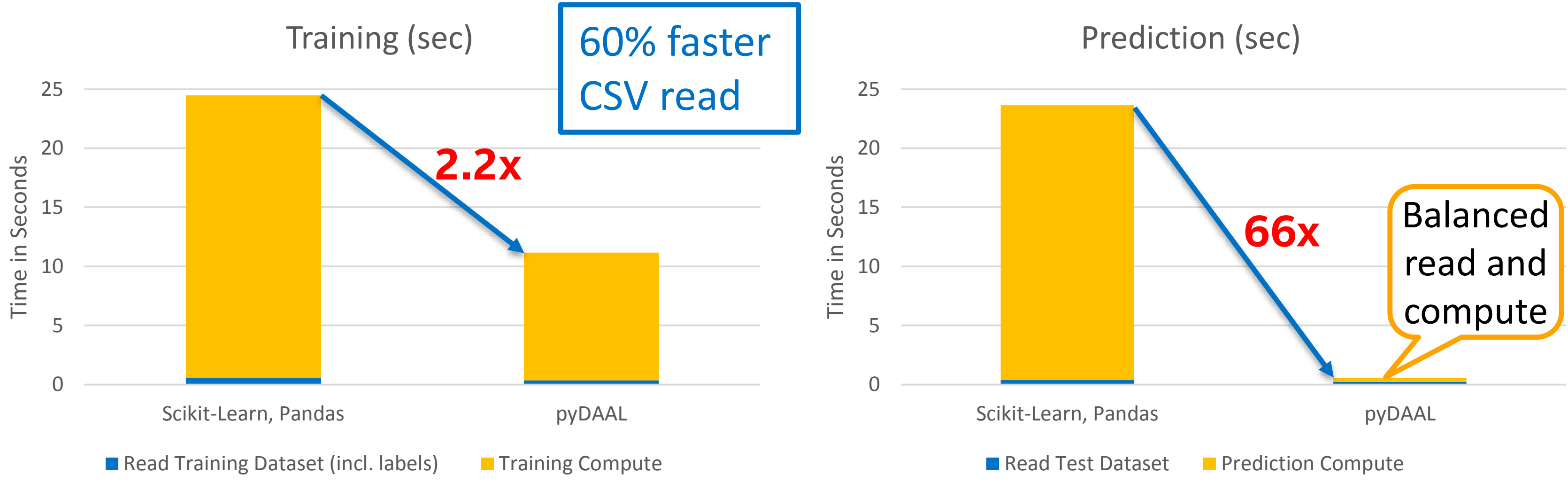
Pass a testing data set and the trained model to the algorithm

Predict the SVM results

PERFORMANCE EXAMPLE : READ AND COMPUTE

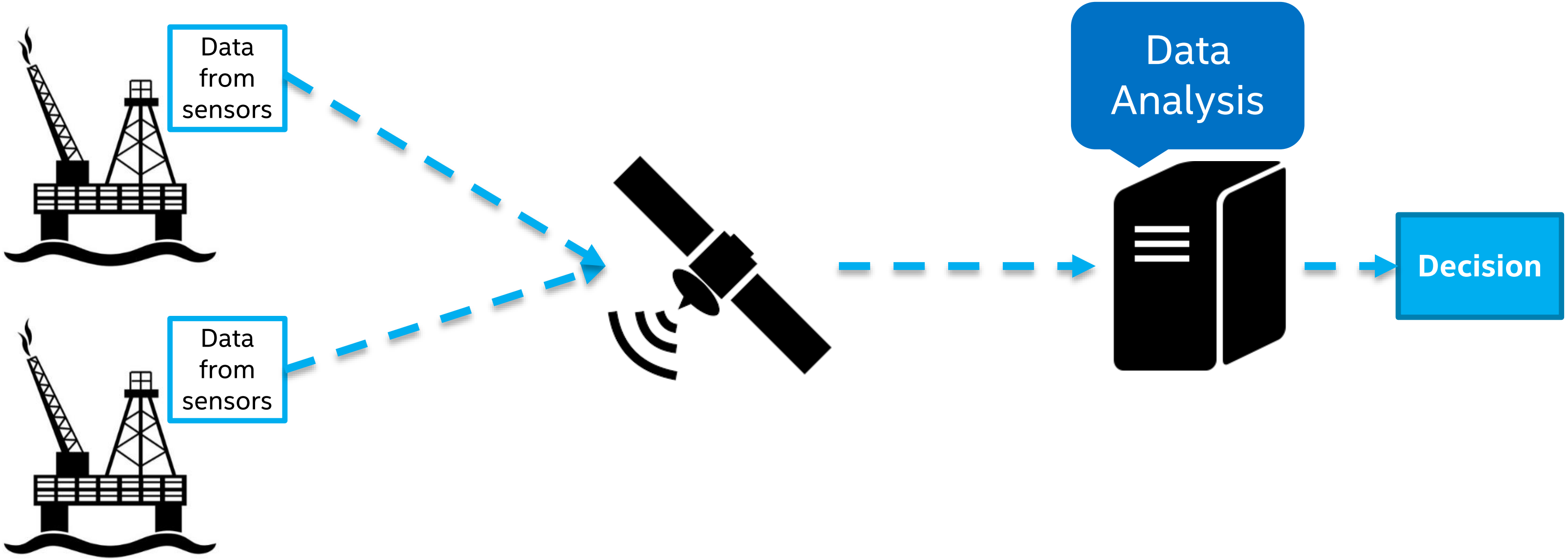
SVM Classification with RBF kernel

- Training dataset: CSV file (PCA-preprocessed MNIST, 40 principal components) *n*=42000, *p*=40
- Testing dataset: CSV file (PCA-preprocessed MNIST, 40 principal components) *n*=28000, *p*=40



System Info: Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz, 504GB, 2x24 cores, HT=on, OS RH7.2 x86_64, Intel® Distribution for Python* 2017 Update 1 (Python* 3.5)

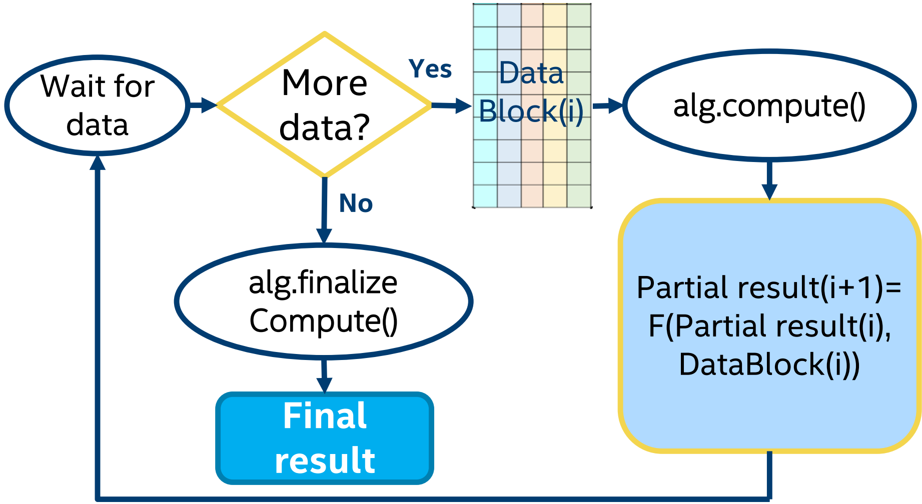
ANOMALY DETECTION PROBLEM EXAMPLE



- State of the art solution
  - The data collected by sensors sent to mainland for analysis and decision making
    - Excessive amount of data is transferred, communication channel is overloaded

SOLUTION: ONLINE PROCESSING

- Update the decision when the new portion of data is available
- The whole data set may not fit into memory but still can be processed on one machine



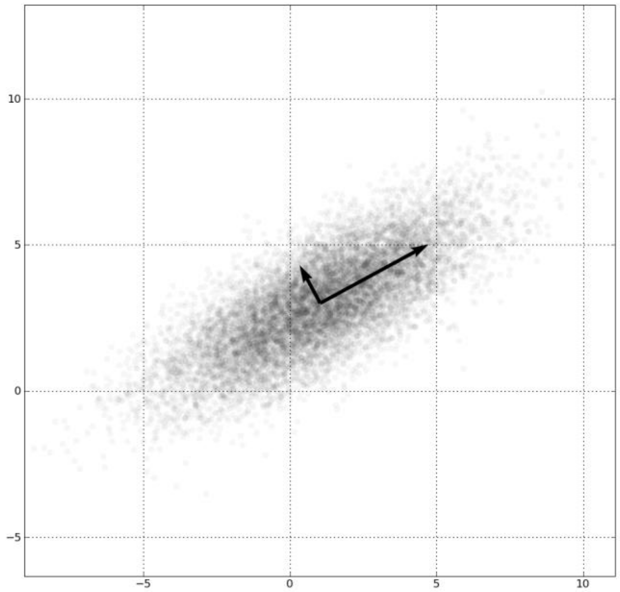
PROJECTION METHODS FOR OUTLIER DETECTION

Principal Component Analysis (PCA)

- Computes principal components: the directions of the largest variance, the directions where the data is mostly spread out

PCA for outlier detection

- Project new observation on the space of the first k principal components
- Calculate score distance for the projection using first k singular values
- Compare the distance against threshold



http://i.stack.imgur.com/uYaTv.png

ONLINE PROCESSING

Data sets that does not fit into memory could be processed effectively on a single machine

```
from daal.algorithms.pca import (
    Online_Float64CorrelationDense, data,
    eigenvalues, eigenvectors
)
```

```
dataSource = FileDataSource(
    dataFileName,
    DataSourceIface.doAllocateNumericTable,
    DataSourceIface.doDictionaryFromContext
)
```

Initialize data source to retrieve data from CSV files

```
algorithm = Online_Float64CorrelationDense()
```

Create a PCA algorithm using correlation method

```
while(dataSource.loadDataBlock(nVectorsInBlock) > 0):
```

Load next block of data from the input file

```
    algorithm.input.setDataset(data,
        dataSource.getNumericTable())
```

Set input data to the algorithm

```
    algorithm.compute()
```

Update partial results of the PCA

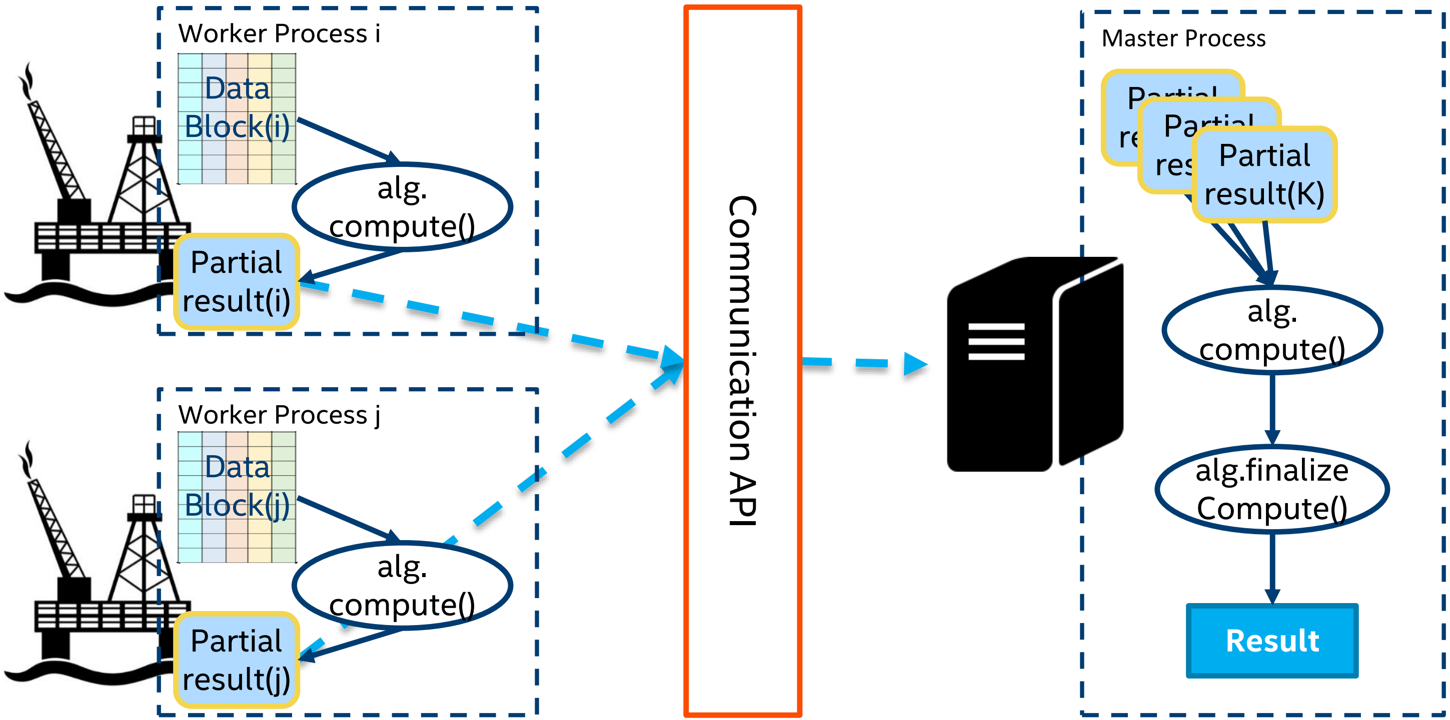
```
result = algorithm.finalizeCompute()
```

Retrieve the results of the algorithm

```
printNumericTable(result.get(eigenvalues))
printNumericTable(result.get(eigenvectors))
```

Print the results

SOLUTION: DISTRIBUTED PROCESSING



DISTRIBUTED COMPUTING LANDSCAPE

mpi4py

pySpark

Dask/distributed

. . .

- New distributed computing technologies appear almost every year
- Intel® DAAL provides communication layer agnostic building blocks to create distributed algorithms

DISTRIBUTED PROCESSING: STEP 1 ON WORKER

```
from daal import step1Local, step2Master
import daal.algorithms.pca as pca
from daal.data_management import (
    OutputDataArchive, InputDataArchive,
    FileDataSource, DataSourceIface
)
```

`dataSource = FileDataSource(datasetFileNames[rankId], …)`	Initialize data source to retrieve data from CSV files
`dataSource.loadDataBlock()`	Retrieve the input data
`algorithm = pca.Distributed(step1Local)`	Create a PCA algorithm for using the correlation method on the local node
`algorithm.input.setDataset(pca.data, dataSource.getNumericTable())`	Set input data to the algorithm
`pres = algorithm.compute()`	Compute partial results of the PCA algorithm

`dataArch = InputDataArchive()`
`pres.serialize(dataArch)`
`nodeResults = dataArch.getArchiveAsArray()`
	Serialize the object that contains the partial results into the array of bytes

`serializedData = comm.gather(nodeResults)`	Gather the partial results on master node

DISTRIBUTED PROCESSING: STEP 2 ON MASTER

```
if rankId == MPI_ROOT:
  masterAlgorithm = pca.Distributed(step2Master)
```

Create a PCA algorithm for using the correlation method on the master node

```
  for i in range(nBlocks):
    dataArch =
      OutputDataArchive(serializedData[i])
    dataForStep2FromStep1 =
      pca.PartialResult(pca.correlationDense)
    dataForStep2FromStep1.deserialize(dataArch)
```

De-serialize partial results that were gathered from the local nodes

```
    masterAlgorithm.input.add(
      pca.partialResults, dataForStep2FromStep1)
```

Set local partial results as inputs for the master algorithm

```
  masterAlgorithm.compute()
```

Compute partial results of the PCA algorithm

```
  res = masterAlgorithm.finalizeCompute()
```

Compute final results of the PCA algorithm

```
  printNumericTable(res.get(pca.eigenvalues))
  printNumericTable(res.get(pca.eigenvectors))
```

Print the results

REFERENCES

- Intel® DAAL User's Guide and Reference Manual
  - https://software.intel.com/sites/products/documentation/doclib/daal/daal-user-and-reference-guides/index.htm
- Intel® Distribution for Python* Documentation
  - https://software.intel.com/en-us/intel-distribution-for-python-support/documentation

WHAT'S NEXT - TAKEAWAYS

- Learn more about Intel® DAAL
  - It supports C++ and Java*, too!
  - We want you to use Intel® DAAL in your data analytics projects
- Learn more about Intel® Distribution for Python*
  - Beyond machine learning, many more benefits
- Keep an eye on the tutorial repository
  - https://github.com/daaltces/pydaal-tutorials
  - We'll be adding more labs, samples, etc.

LEGAL DISCLAIMER & OPTIMIZATION NOTICE

intel

Software