
intel
SC16
FUEL YOUR INSIGHT

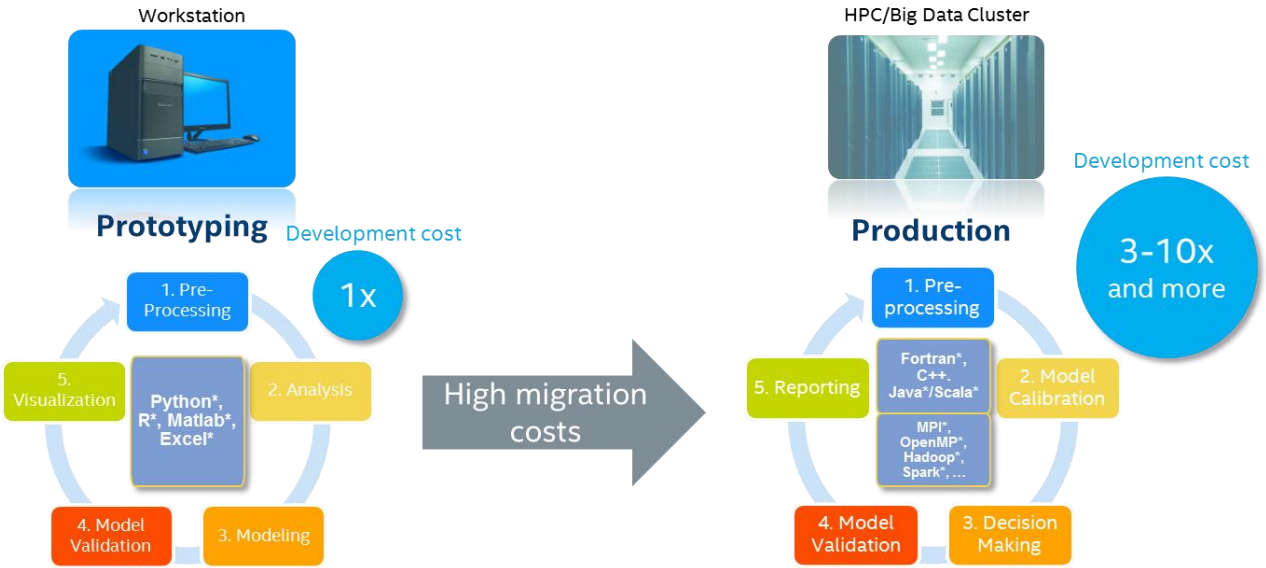
Scaling Out Python* To HPC and Big Data

Sergey Maidanov

Software Engineering Manager for

Intel® Distribution for Python*

What Problems We Solve: Scalable Performance



Make Python usable beyond prototyping environment by scaling out to HPC and Big Data environments

What Problems We Solve: Ease of Use

Developer Zone https://software.intel.com/en-us/forums/intel-math-kernel-library/topic/280832

Join Today › Log in

Development › Tools › Resources ›

powered by Google

Home › Forums › Intel® Software Development Products › Intel® Math Kernel Library

compiling and linking MKL with numpy/scipy

Xavier Barthelemy Tue, 11/22/2011 - 15:28

dear everyone,

I am hard trying to compile numpy / and scipy with mkl.

unfortunately it does not work. i have tried a lot of solution, and the closest for me to work is:

Developer Zone https://software.intel.com/en-us/articles/numpyscipy-with-intel-mkl

Join Today › Log in

Development › Tools › Resources ›

powered by Google

Numpy/Scipy with Intel® MKL and Intel® Compilers

By Vipin Kumar E K (Intel), Added June 28, 2012 Translate

Developer Zone https://software.intel.com/en-us/articles/building-numpyscipy-with-intel-mkl-and-intel-fortran-on-windows

Join Today › Log in

Development › Tools › Resources ›

powered by Google

Building Numpy/Scipy with Intel® MKL and Intel® Compilers on Windows

By Yuan C. (Intel), Added February 12, 2015 Translate

Share Tweet Share

Forums ›

Intel® Math Kernel Library ›

NumPy/SciPy Application Note

Step 1 - Overview

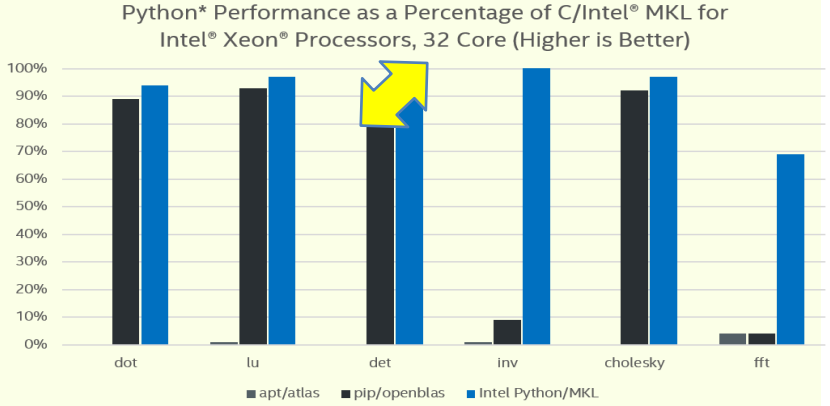
This guide is intended to help current NumPy/SciPy users to take advantage of Intel® Math Kernel Library (Intel® MKL), Intel® Fortran and Intel® C++ Compilers on Microsoft Windows platform.

"Any articles I found on your site that related to actually using the MKL for compiling something were overly technical. ***I couldn't figure out what the heck some of the things were doing or talking about****."* – Intel® Parallel Studio 2015 Beta Survey Response

Why Yet Another Python Distribution?

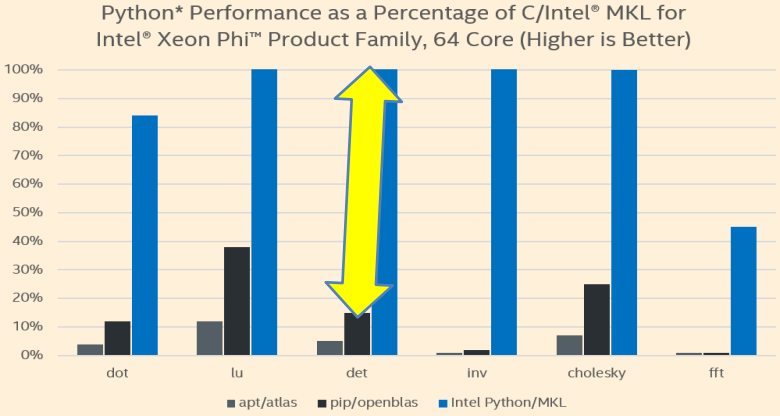
Mature AVX2 instructions based product

Intel® Xeon® Processors



New AVX512 instructions based product

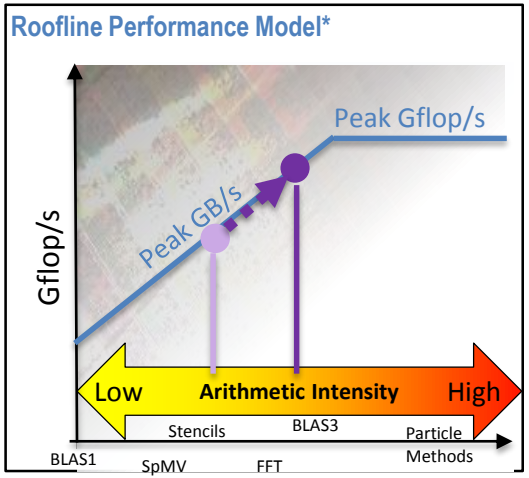
Intel® Xeon Phi™ Product Family

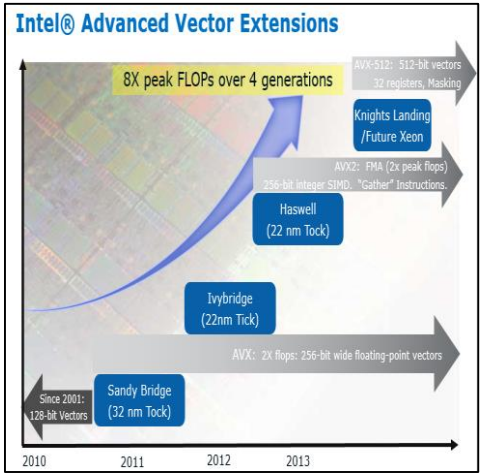


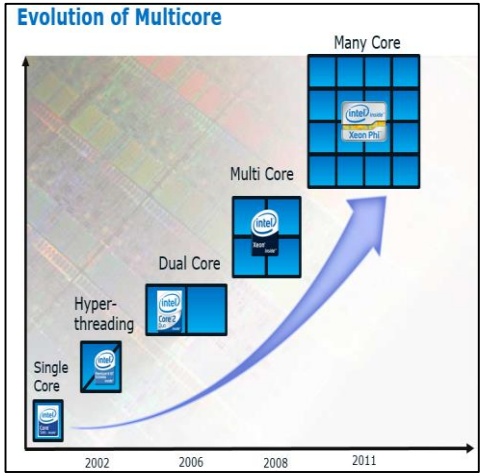
Configuration Info: apt/atlas: installed with apt-get, Ubuntu 16.10, python 3.5.2, numpy 1.11.0, scipy 0.17.0; pip/openblas: installed with pip, Ubuntu 16.10, python 3.5.2, numpy 1.11.1, scipy 0.18.0; Intel Python: Intel Distribution for Python 2017;. Hardware: Xeon: Intel Xeon CPU E5-2698 v3 @ 2.30 GHz (2 sockets, 16 cores each, HT=off), 64 GB of RAM, 8 DIMMS of 8GB@2133MHz; Xeon Phi: Intel Intel® Xeon Phi™ CPU 7210 1.30 GHz, 96 GB of RAM, 6 DIMMS of 16GB@1200MHz

Scaling To HPC & Big Data Environments

- Hardware and software efficiency crucial in production (Perf/Watt, etc.)
- Efficiency = Parallelism
  - Instruction Level Parallelism with effective memory access patterns
  - SIMD
  - Multi-threading
  - Multi-node







* Roofline Performance Model https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

Efficiency = Parallelism and Python?

- CPython as interpreter inhibits parallelism but…
- … Overall Python tools evolved far toward unlocking parallelism

Native extensions numpy*, scipy*, scikit-learn* accelerated with Intel® MKL, Intel® DAAL, Intel® IPP

Composable multi-threading with Intel® TBB and Dask*

Multi-node parallelism with mpi4py* accelerated with Intel® MPI*

Language extensions for vectorization & multi-threading (Cython*, Numba*)

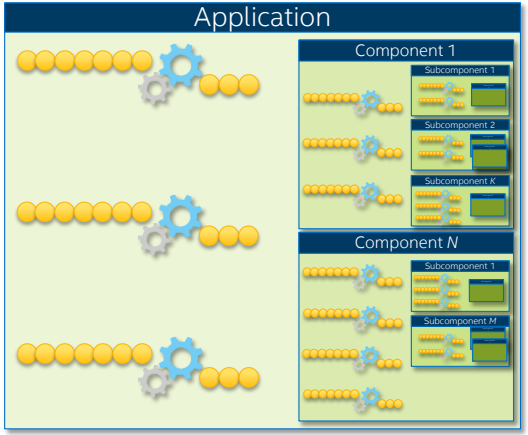
Integration with Big Data platforms and Machine Learning frameworks (pySpark*, Theano*, TensorFlow*, etc.)

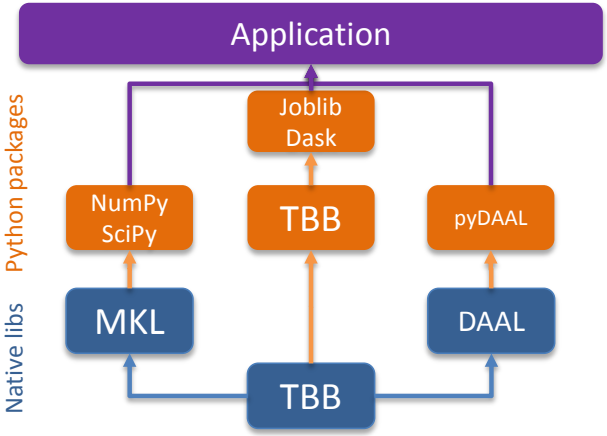
Mixed language profiling with Intel® VTune™ Amplifier

Composable Multi-Threading With Intel® TBB

- Amhdal's law suggests extracting parallelism at all levels
- If software components do not coordinate on threads use it may lead to oversubscription
- Intel TBB dynamically balances HW thread loads and effectively manages oversubscription
- Intel engineers extended Cpython* and Numba* thread pools with support of Intel® TBB

```
>python -m TBB myapp.py
```

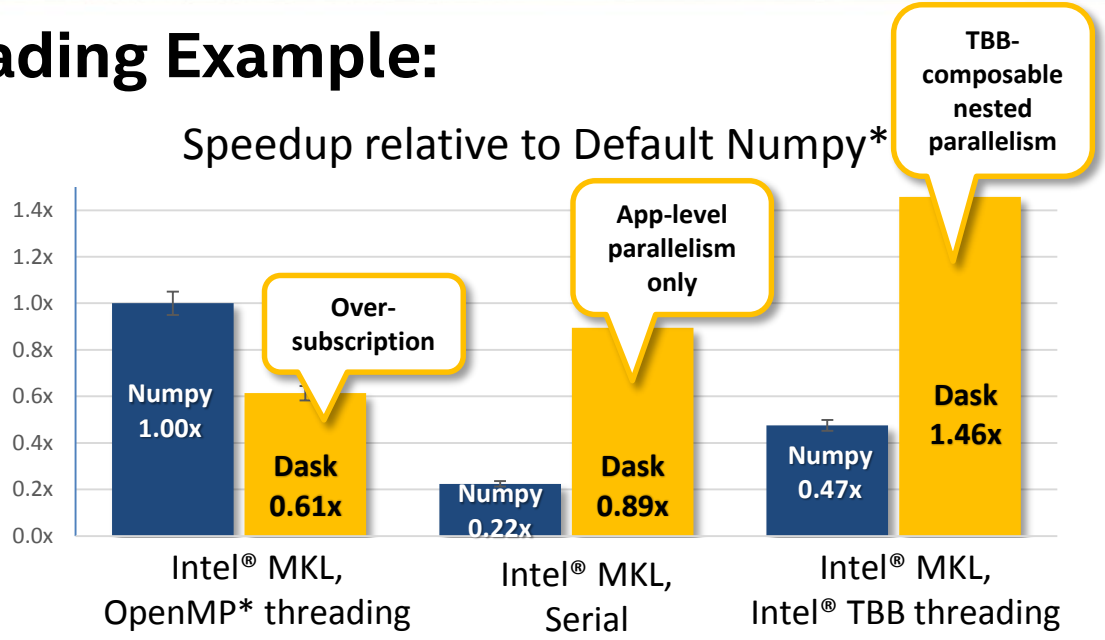




Composable Multi-Threading Example: Batch QR Performance

```python
import time, numpy as np
x = np.random.random((100000, 2000))
t0 = time.time()
q, r = np.linalg.qr(x)
test = np.allclose(x, q.dot(r))
assert(test)
print(time.time() - t0)
```

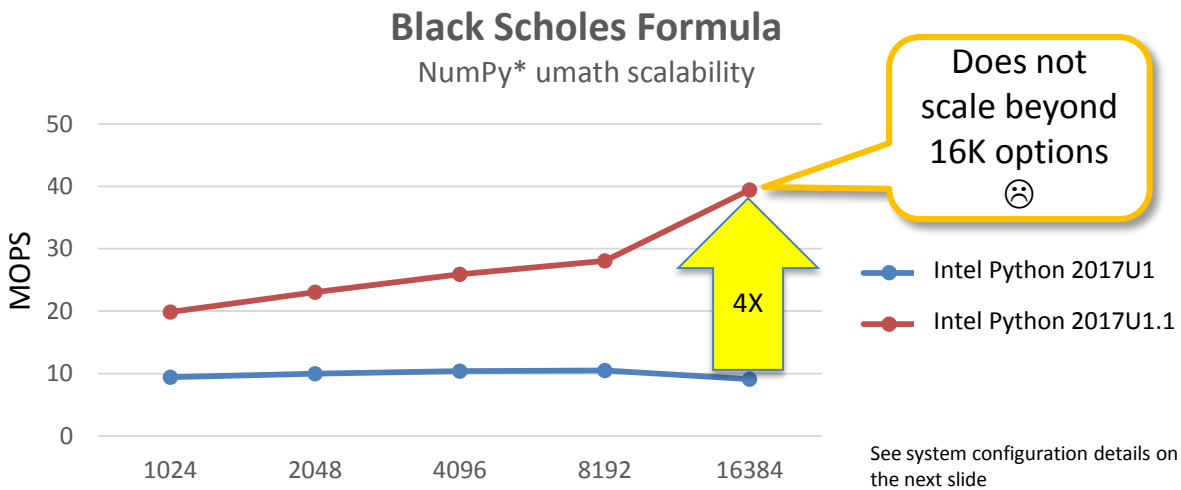
```python
import time, dask, dask.array as da
x = da.random.random((100000, 2000),
                     chunks=(10000, 2000))
t0 = time.time()
q, r = da.linalg.qr(x)
test = da.all(da.isclose(x, q.dot(r)))
assert(test.compute()) # threaded
print(time.time() - t0)
```



System info: 32x Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz, disabled HT, 64GB RAM; Intel(R) MKL 2017.0 Beta Update 1 Intel(R) 64 architecture, Intel(R) AVX2; Intel(R)TBB 4.4.4; Ubuntu 14.04.4 LTS; Dask 0.10.0; Numpy 1.11.0.

NumPy* and SciPy* Continuous Improvement

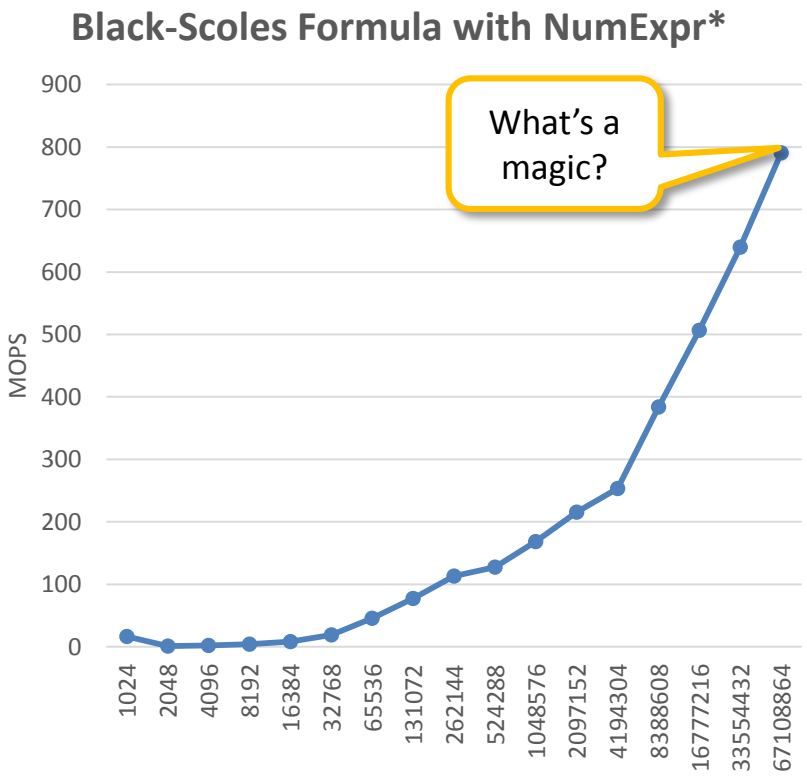
- MKL-level performance for dense linear algebra and FFT
- NumPy random, umath & NumExpr* exploit SIMD and multi-threading out-of-the-box
- Better memory allocation and copy in NumPy



See system configuration details on the next slide

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

    return call, put
```

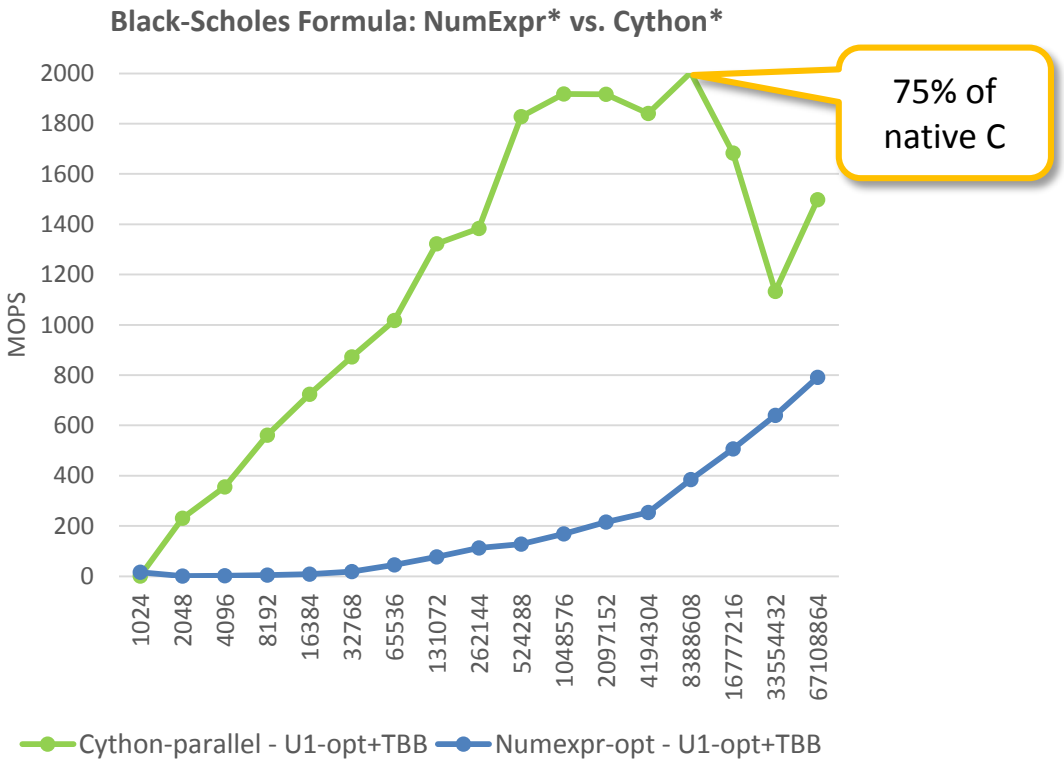
NumExpr* Scalability: Black-Scholes Formula




```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
```

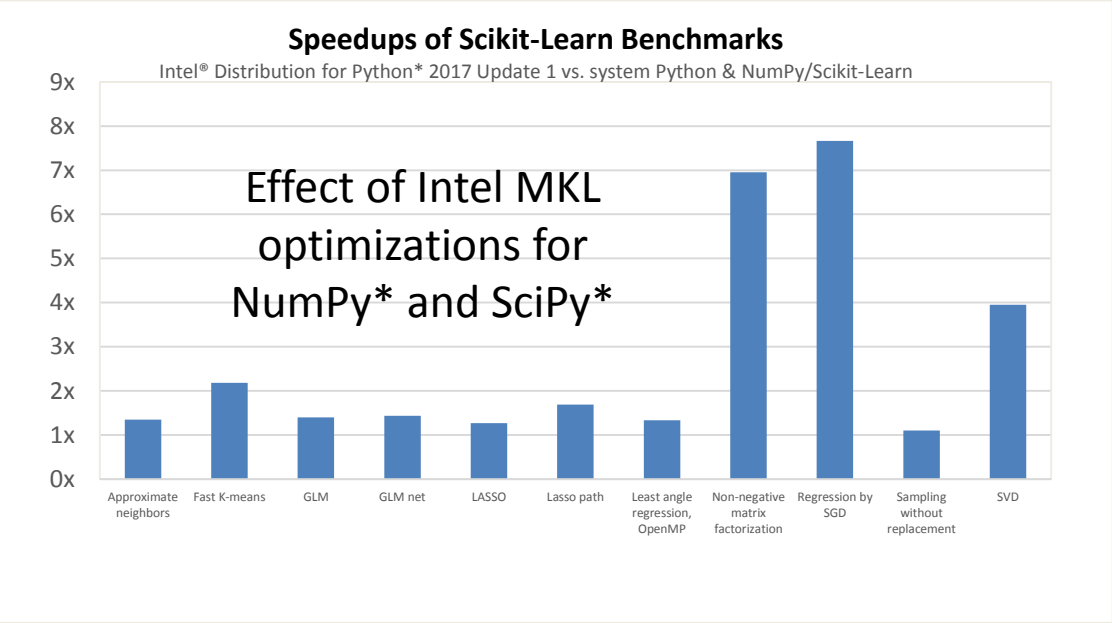
System info: 32x Intel® Xeon® CPU E5-2698 v3 @ 2.30GHz, disabled HT, 64GB RAM; Intel® Distribution for Python* 2017 Update

Cython* Scalability: Black-Scholes Formula

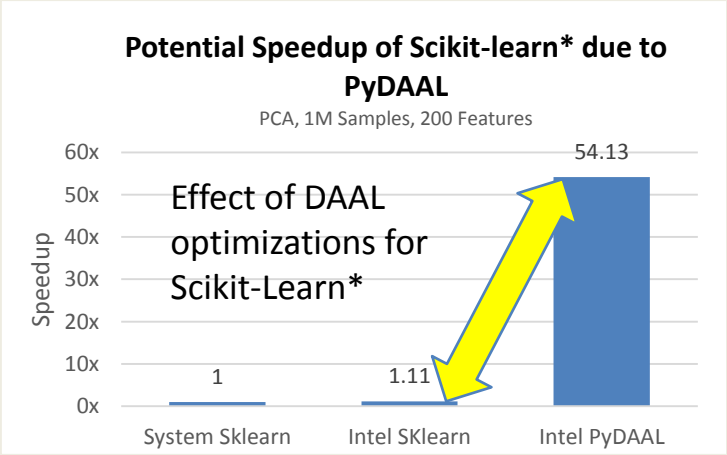



```
# In order to release GIL for a parallel loop, the code in this block cannot
# manipulate Python objects in any way.
@boundscheck(False)
@wraparound(False)
@cdivision(True)
@initializedcheck(False)
def black_scholes(int nopt,
                  double[:] price,
                  double[:] strike,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

Skt-Learn* Optimizations With Intel® MKL… And Intel® DAAL



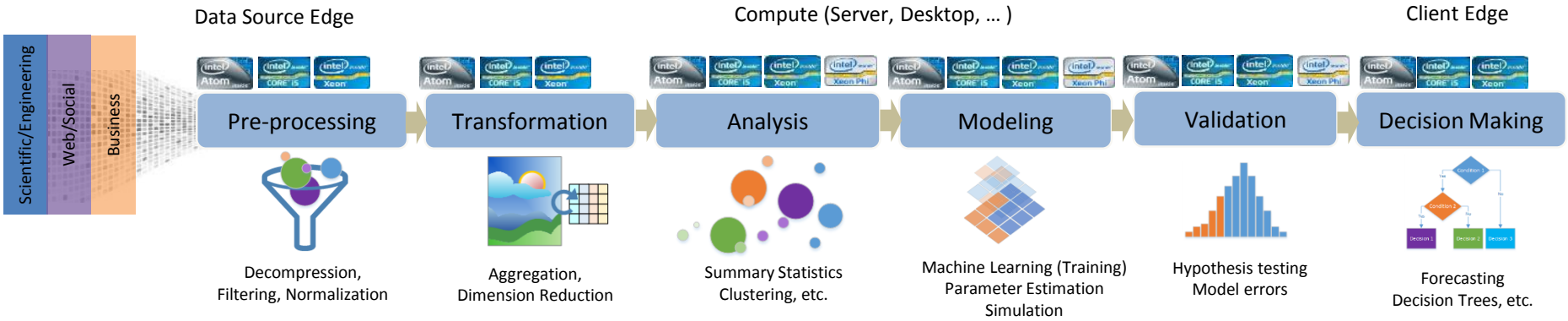
Intel® Distribution for Python* ships Intel® Data Analytics Acceleration Library with Python interfaces, a.k.a. pyDAAL



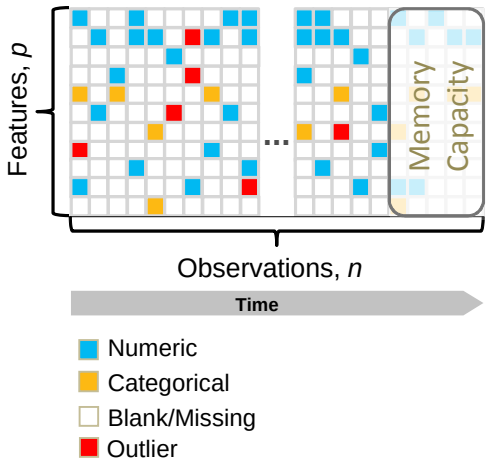
System info: 32x Intel® Xeon® CPU E5-2698 v3 @ 2.30GHz, disabled HT, 64GB RAM; Intel® Distribution for Python* 2017 Gold; Intel® MKL 2017.0.0; Ubuntu 14.04.4 LTS; Numpy 1.11.1; scikit-learn 0.17.1.

Ideas Behind Intel® DAAL: Heterogeneous Analytics

- Data is different, data analytics pipeline is the same
- Data transfer between devices is costly, protocols are different
  - Need data analysis proximity to Data Source
  - Need data analysis proximity to Client
  - Data Source device ≠ Client device
  - Requires abstraction from communication protocols



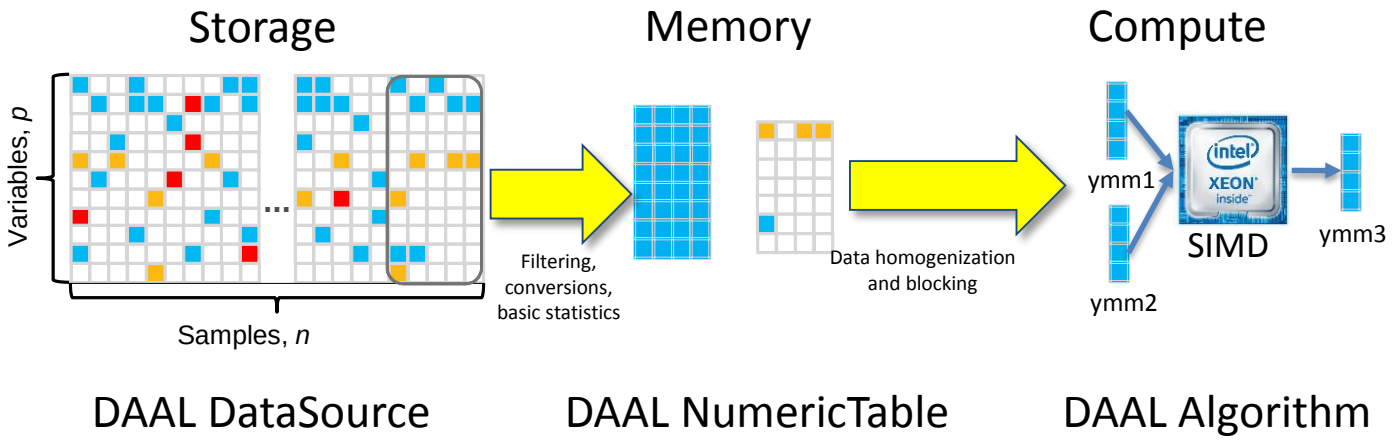
Ideas Behind Intel® DAAL: Effective Data Management, Streaming and Distributed Processing



Big Data Attributes	Computational Solution
Distributed across different devices	•Distributed processing with communication-avoiding algorithms
Huge data size not fitting into device memory	•Distributed processing •Streaming algorithms
Data coming in time	•Data buffering & asynchronous computing •Streaming algorithms
Non-homogeneous data	•Categorical→Numeric (counters, histograms, etc) •Homogeneous numeric data kernels • Conversions, Indexing, Repacking
Sparse/Missing/Noisy data	•Sparse data algorithms •Recovery methods (bootstrapping, outlier correction)

Ideas Behind Intel® DAAL: Storage & Compute

- Optimizing storage ≠ optimizing compute
  - Storage: efficient non-homogeneous data encoding for smaller footprint and faster retrieval
  - Compute: efficient memory layout, homogeneous data, contiguous access
  - Easier manageable for traditional HPC, much more challenging for Big Data



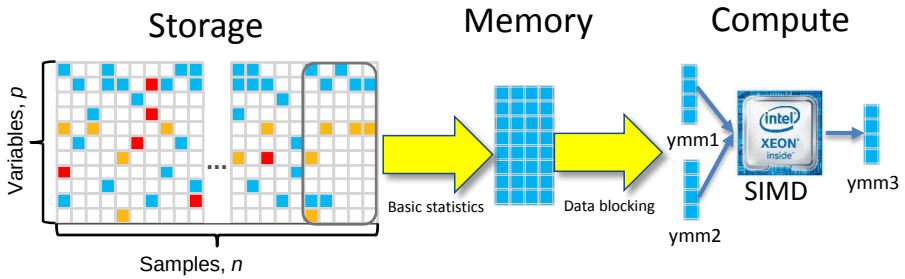
Ideas Behind Intel® DAAL: Languages & Platforms

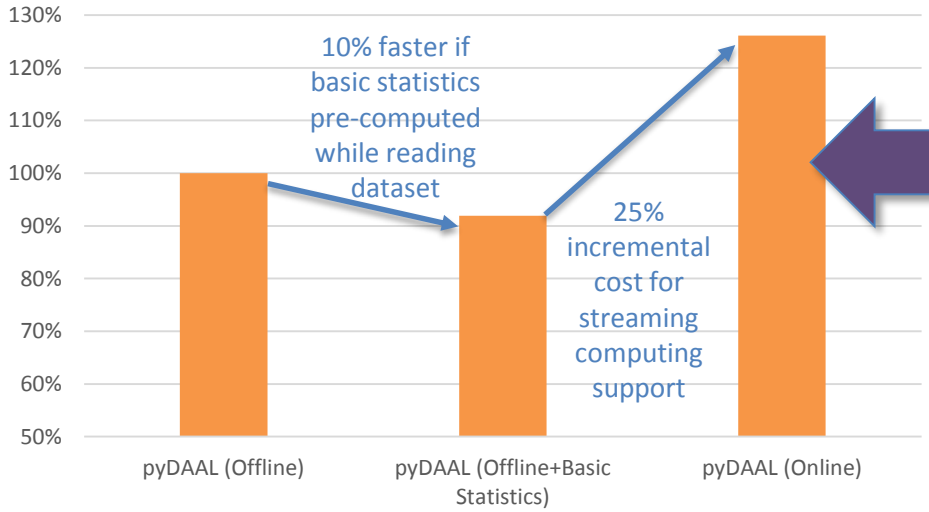
DAAL has multiple programming language bindings

- C++ – ultimate performance for real-time analytics with DAAL
- Java*/Scala* – easy integration with Big Data platforms (Hadoop*, Spark*, etc)
- Python* – advanced analytics for data scientist

Ex 1. Compute Basic Statistics While Reading Dataset
Ex 2. Offline *vs*. Online

- $cov(X,Y) = E((X - E(X))(Y - E(Y))$
- Algorithm: Maximum Likelihood Estimator
- Dataset: CSV file *n*=1M, *p*=100





Better memory management in pyDAAL allows bigger datasets even for offline processing

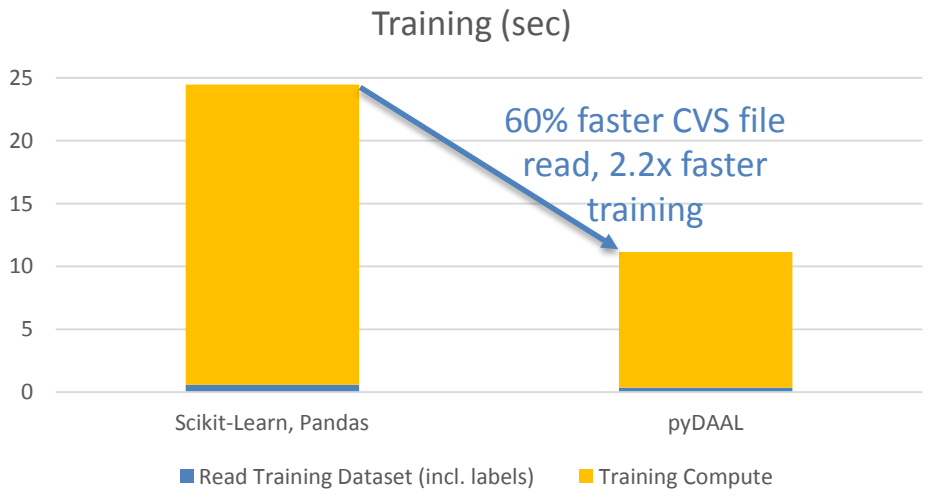
- Ex.: Scikit-learn fails to process *n*=50M, *p*=1000 while pyDAAL successfully completes work

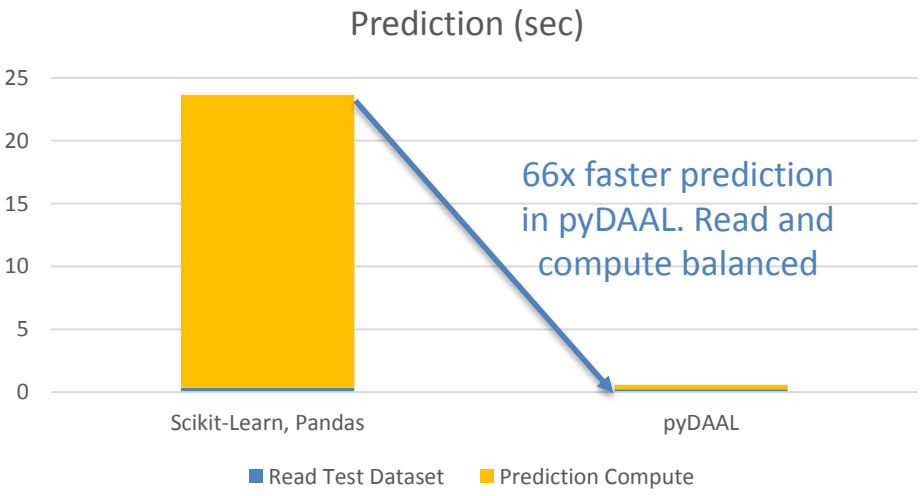
Online processing removes dataset size limitation

System Info: Intel(R) Xeon(R) 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)

Ex 3: Read *vs.* Compute

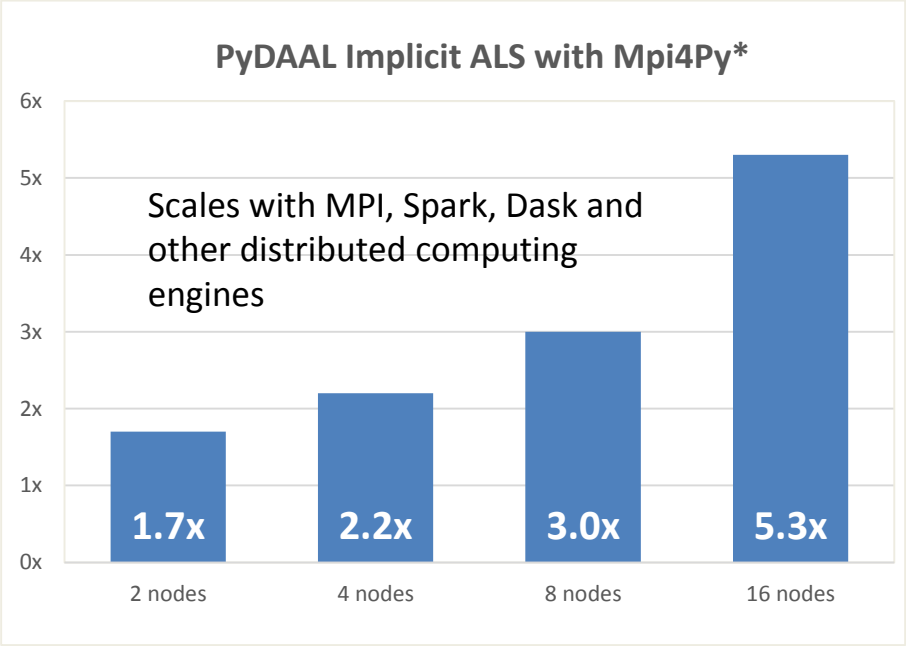
- Algorithm: SVM Classification with RBF kernel
- Training dataset: CSV file (PCA-preprocessed MNIST, 40 principal components) n=42000, p=40
- Testing dataset: CSV file (PCA-preprocessed MNIST, 40 principal components) n=28000, p=40





Distributed Parallelism

- Intel® MPI* accelerates Intel® Distribution for Python (mpi4py*, ipyparallel*)
- Intel Distribution for Python also supports
  - PySpark* - Python* interfaces for Spark*, an engine for large-scale data processing
  - Dask* - flexible parallel computing library for numerical computing



Configuration Info: Hardware (each node): Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz, 2x18 cores, HT is ON, RAM 128GB; Versions: Oracle Linux Server 6.6, Intel® DAAL 2017 Gold, Intel® MPI 5.1.3; Interconnect: 1 GB Ethernet.

Summary And Call To Action

- Intel created the Python* distribution for out-of-the-box performance and scalability on Intel® Architecture
  - With minimum to no code modification Python aims to scale
- Multiple technologies applied to unlock parallelism at all levels
  - Numerical libraries, libraries for parallelism, Python code compilation/JITing, profiling
  - Enhancing mature Python packages and bringing new technologies, e.g. pyDAAL, TBB
- With multiple choices available Python developer needs to be conscious what will scale best
  - Intel® VTune™ Amplifier helps making conscious decisions

Intel Distribution for Python is free!

https://software.intel.com/en-us/intel-distribution-for-python

Commercial support included for Intel® Parallel Studio XE customers!

Easy to install with Anaconda* https://anaconda.org/intel/

@IntelHPC | intel.com/SC16