THE PAINLESS ROUTE IN PYTHON TO FAST AND SCALABLE MACHINE LEARNING
Victoriya Fedotova, Frank Schlimbach
# The Reality of “Data Centric Computing”

## Software Challenges:

| Performance Limited | • Software is slow and single-node for many organizations  
<table>
<thead>
<tr>
<th></th>
<th>• Only sample a small portion of the data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Productivity Limited</td>
<td>• More performant/scalable implementations require significantly more development &amp; deployment skills &amp; time</td>
</tr>
<tr>
<td>Compute Limited</td>
<td>• Performance bottleneck often in compute, not storage/memory</td>
</tr>
</tbody>
</table>

A typical data scientist analyzes only a small portion of data that they think has the most potential of bringing the great insights. This means they may miss out on valuable insights from the remaining bigger portion of the data — insights that may be crucial for the business.
Easy, out-of-the-box access to high performance Python

- Prebuilt accelerated solutions for data analytics, numerical computing, etc.
- Drop in replacement for your existing Python. No code changes required.

Learn More: software.intel.com/distribution-for-python
TWO INGREDIENTS TO GET CLOSE-TO-NATIVE PERFORMANCE IN PYTHON

- **Pure Python**
  - Serial
  - Interpreted

- **Python + Libraries**
  - Partially Ninja-level
  - Partially Interpreted

- **Libraries + JIT**
  - Largely Ninja-level
  - 100% native

- **C++**
  - 100% Ninja-level
  - 100% native
DATA ANALYSIS AND MACHINE LEARNING

Data Input
- Pandas
- Spark
- SDC

Data Preprocessing

Model Creation
- Scikit-learn
- Spark
- DL frameworks
daal4py

Prediction

more nodes, more cores, more threads, wider vectors, ...
INTEL® DATA ANALYTICS ACCELERATION LIBRARY (DAAL)
ACCELERATING MACHINE LEARNING

- Efficient memory layout
- Chunking for optimal cache performance
- Computations mapped to most efficient matrix operations (in MKL)
- Parallelization via TBB
- Vectorization

Try it out! conda install --cintel scikit-learn
CLOSE TO NATIVE SCIKIT-LEARN PERFORMANCE WITH INTEL PYTHON 2020
COMPARSED TO STOCK PYTHON PACKAGES ON INTEL® XEON PROCESSORS

Performance efficiency measured against native code with Intel® DAAL

Configuration: Testing by Intel as of November 27, 2019. Stock Python: Python 3.7.5 h_0371630_0 installed from conda, numpy 1.17.4, numba 0.46.0, llvmlite 0.30.0, scipy 1.3.2, scikit-learn 0.21.3 installed from pip; Intel Python: Intel® Distribution for Python* 2020 Gold: Python 3.7.4 hf484d3e_3, numpy 1.17.3 py37ha68da19_4, mlk 2020 intel_133, mkl_fft 1.0.15 py37ha68da19_3, mkl_random 1.1.0 py37ha68da19_0, numa 0.45.1 rp117py37_1, llvmlite 0.29.0 py37hf484d3e_9, scipy 1.3.1 py37ha68da19_2, scikit-learn 0.21.3 py37ha68da19_14, daal 2020 intel_133, daal4py 2020 py37ha68da19_4; CentOS Linux 7.3.1611, kernel 3.10.0-516.e7.x86_64; Hardware: Intel Xeon® Platinum® 8280 CPU @ 2.70 GHz (2 sockets, 28 cores/socket, HT off), 256 GB of DDR4 RAM, 16 DIMMs of 16 GB@2666MHz.

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

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
ACCELERATING SCIKIT-LEARN THROUGH DAAL4PY

> python -m daal4py <your-scikit-learn-script>

import daal4py.sklearn
daal4py.sklearn.patch_sklearn('kmeans')

Monkey-patch any scikit-learn on the command-line

Monkey-patch any scikit-learn programmatically

Scikit-learn with daal4py patches applied passes Scikit-learn test-suite
SCALING MACHINE LEARNING BEYOND A SINGLE NODE

- scikit-learn
- daal4py

-Powered by DAAL
-Scalable to multiple nodes
-Open source

Try it out! conda install -c intel daal4py
K-MEANS USING SCIKIT-LEARN AND DAAL4PY

**Scikit-learn**

```python
from sklearn.cluster import KMeans
import pandas as pd

data = pd.read_csv("./kmeans.csv")

algo = KMeans(n_clusters=20,
              init='k-means++',
              max_iter=5)

result = algo.fit(data)

result.labels_
result.cluster_centers_
```

**daal4py**

```python
from daal4py import kmeans_init, kmeans
import pandas as pd

data = pd.read_csv("./kmeans.csv")

init = kmeans_init(nClusters=20,
                   method="plusPlusDense").compute(data)

algo = kmeans(nClusters=20,
              maxIterations=5, assignFlag=True)

result = algo.compute(data,
                       init.centroids)

result.assignments
result.centroids
```

# Load the data

# Compute initial centroids

# Configure K-means main object

# Compute the clusters and labels

# Print the results
DISTRIBUTED K-MEANS USING DAAL4PY

```python
from daal4py import kmeans_init, kmeans, daalinit, daalfini, my_procid
import pandas as pd

# Optionally initialize distributed execution environment
daalinit()

# Load the data. Daal4py accepts data as CSV files, numpy arrays or pandas dataframes
data = pd.read_csv("./kmeans_dense_{}.csv".format(my_procid() + 1))

# compute initial centroids
init_res = kmeans_init(nClusters=10, method="plusPlusDense", distributed=True).compute(data)

# configure kmeans main object: we also request the cluster assignments
algo = kmeans(nClusters=10, maxIterations=25, distributed=True)

# compute the clusters/centroids
result = algo.compute(data, init_res.centroids)

daalfini()

mpirun -n 4 python kmeans_distributed.py```

Semi-automatic API generation process:

- Parse C++ headers to generate Cython code
- Use jinja2 to generate Python classes for algorithms, models, results, etc.

100X fewer LOC for multi-node algorithms
**EFFECTIVE DATA TRANSFER: PYTHON ↔ NATIVE**

**Python data type**
- numpy.ndarray
- Homogeneous dense array

**DAAL data type**
- Memory layout: Homogeneous
  - Sample 1
  - Sample 2
  - Sample n

**Python data type**
- pandas.DataFrame
- Heterogeneous data

**DAAL data type**
- Memory layout: Structure-of-Arrays (SOA)
  - Feature 1
  - Feature 2
  - Feature p

**daal4py mostly avoids data copies and works optimally with various data layouts**
On a 32-node cluster (1280 cores) daal4py computed K-Means (10 clusters) of 1.12 TB of data in 107.4 seconds and 35.76 GB of data in 4.8 seconds.

Configuration: Intel® Xeon® Gold 6148 CPU @ 2.40GHz, EIST/Turbo on 2 sockets, 20 cores per socket, 192 GB RAM, 16 nodes connected with Infiniband, Oracle Linux Server release 7.4, using 64-bit floating point numbers.
INTEL® SCALABLE DATAFRAME COMPILER (SDC)
It's a compiler
A just-in-time compiler

Uses all available CPU cores
Auto-vectorizes for SIMD parallelism
Compiles to native code with Numba*
Optimizes memory footprint

You are here!

```python
import pandas as pd
from numba import njit

# Dataset for analysis
FNAME = "employees.csv"

# This function gets compiled by Numba*
@njit
def get_analyzed_data():
    df = pd.read_csv(FNAME)
    s_bonus = pd.Series(df['Bonus %'])
    s_first_name = pd.Series(df['First Name'])
    m = s_bonus.mean()
    names = s_first_name.sort_values()
    return m, names

# Printing names and their average bonus percent
mean_bonus, sorted_first_names = get_analyzed_data()
print(sorted_first_names)
print('Average Bonus %:', mean_bonus)
```
COMPILATION PIPELINE (HIGH-LEVEL VIEW)

Decorator @numba.jit

Type Analysis

Parallel Analysis

Compile

Efficient binary

@hpat.jit
def get_stats():
    ...
df[‘latency’].sum()
df[‘latency’].mean()
    ...

vcomisd %xmm0, %xmm0
setnp %dl
jp .LBB0_11
vaddsd %xmm0, %xmm2, %xmm2
.LBB0_11:
vaddsd %xmm0, %xmm3, %xmm1
vcmpunordsd %xmm0, %xmm0, %xmm0
vblendvpd %xmm0, %xmm3, %xmm1,
import numpy as np
import pandas as pd
from numba import njit

# This function gets compiled by Numba*
@njit
def get_analyzed_data(file_name):
    df = pd.read_csv(file_name,
                     dtype={'Bonus %': np.float64, 'First Name': str},
                     usecols=['Bonus %', 'First Name'])
    s_bonus = pd.Series(df['Bonus %'])
    s_first_name = pd.Series(df['First Name'])
    m = s_bonus.mean()
    names = s_first_name.sort_values()
    return m, names

mean_bonus, sorted_first_names = get_analyzed_data('employees.csv')
print(sorted_first_names)
print('Average Bonus %:', mean_bonus)
Input code to SDC must be statically compilable (type stable)

- Dynamically typed code examples (rare in analytics):

<table>
<thead>
<tr>
<th>Untypable variable</th>
<th>Unresolvable function</th>
<th>Nonstatic DataFrame schema</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if flag1:</code></td>
<td></td>
<td><code>if flag3:</code></td>
</tr>
<tr>
<td><code>a = 2</code></td>
<td></td>
<td><code>df = pd.DataFrame({'A': [1, 2, 3]})</code></td>
</tr>
<tr>
<td><code>else:</code></td>
<td></td>
<td><code>else:</code></td>
</tr>
<tr>
<td><code>a = np.ones(n)</code></td>
<td><code>if flag2:</code></td>
<td><code>df = pd.DataFrame({'A': ['a', 'b', 'c']})</code></td>
</tr>
<tr>
<td><code>if isinstance(a, np.ndarray):</code></td>
<td><code>f = np.zeros</code></td>
<td><code>else:</code></td>
</tr>
<tr>
<td><code>doWork(a)</code></td>
<td><code>f = np.ones</code></td>
<td><code>b = f(m)</code></td>
</tr>
<tr>
<td></td>
<td><code>b = f(m)</code></td>
<td><code>b = f(m)</code></td>
</tr>
</tbody>
</table>
Getting Performance with Intel® SDC

- Compile parts of code where parallelism resides
- Compile functions that are called multiple times
- Minimize number of columns in dataframes in the regions being compiled
Intel SDC Performance – read_csv

INPUT/OUTPUT OPERATIONS
SPEEDUP SDC VS. PANDAS

<table>
<thead>
<tr>
<th>Threads</th>
<th>SDC Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.7</td>
</tr>
<tr>
<td>4</td>
<td>2.5</td>
</tr>
<tr>
<td>28</td>
<td>8.9</td>
</tr>
<tr>
<td>56</td>
<td>10.7</td>
</tr>
</tbody>
</table>

Intel® SDC Beta, Numba® 0.48, Pandas® 0.25.3
Intel® Xeon® Platinum 8280L, 2.7 GHz, 2x28 cores, Hyperthreading=on, Turbo Mode=on
Intel SDC Performance – Dataframes

**DATAFRAME OPERATIONS**

**SPEEDUP SDC VS. PANDAS**
- count
- drop
- max (skipna=True)

**DATAFRAME ROLLING WINDOWS OPERATIONS**

**SPEEDUP SDC VS. PANDAS**
- kurt
- mean
- std

Intel® SDC Beta, Numba* 0.48, Pandas* 0.25.3
Intel® Xeon™ Platinum 8280L, 2.7 GHz, 2x28 cores, Hyperthreading=on, Turbo Mode=on
Intel SDC Performance – Series

SERIES OPERATIONS
SPEEDUP SDC VS. PANDAS

SERIES ROLLING WINDOWS OPERATIONS
SPEEDUP SDC VS. PANDAS

Intel® SDC Beta, Numba* 0.48, Pandas* 0.25.3
Intel® Xeon™ Platinum 8280L, 2.7 GHz, 2x28 cores, Hyperthreading=on, Turbo Mode=on
## INTEL® SDC FUNCTIONALITY

### Operations
- Python/Numpy/Pandas* basics
- Statistical operations (max, std, median, ...)
- Relational operations (filter, groupby)
- Rolling window (rolling)

### Data
- Missing value
- Dates
- ASCII/Unicode strings
- Data-Frames, Series, Lists, Dictionaries, Tuples

### Interoperability
- I/O integration (CSV)

Coming soon: time series and categoricals
INTEL® SCALABLE DATAFRAME COMPILER (SDC)
EVOLVED FROM HIGH-PERFORMANCE ANALYTICS TOOLKIT (HPAT)

Open source project

• https://github.com/IntelPython/sdc
• https://intelpython.github.io/sdc-doc/latest/index.html

In Beta till end of 2020

Available as conda packages and pip wheels (Python 3.6/3.7, Windows/Linux)

- conda install -c intel/label/beta sdc
- pip install -i https://pypi.anaconda.org/intel/label/beta/simple sdc
WHAT'S NEXT?

More Pandas features

Auto-scale-out to clusters of workstations

Compiling to and running on GPUS/accelerators
Compiling to and Running on GPUs/Accelerators

```python
@numba.njit(parallel=True, fastmath=True)
def blackscholes(sptprice, strike, rate, volatility, timev):
    logterm = np.log(sptprice / strike)
    powterm = 0.5 * volatility * volatility
    den = volatility * np.sqrt(timev)
    d1 = ((rate + powterm) * timev + logterm) / den
    d2 = d1 - den
    NofXd1 = cndf2(d1)
    NofXd2 = cndf2(d2)
    futureValue = strike * np.exp(- rate * timev)
    c1 = futureValue * NofXd2
    call = sptprice * NofXd1 - c1
    put = call - futureValue + sptprice
    return put
```

```python
@numba.vectorize(nopython=True)
def cndf2(inp):
    out = 0.5 + 0.5 * math.erf((math.sqrt(2.0)/2.0) * inp)
    return out
```

```python
@dppy.kernel
def data_parallel_sum(a, b, c):
    i = dppy.get_global_id(0)
    c[i] = a[i] + b[i]
```

```python
with device_context(gpu, 0):
    black_scholes(SP, S, R, V, T)
```
END-TO-END PERFORMANCE OF ANALYTICS

daal4py and SDC help accelerate, scale-up and scale-out the entire analytics process in Python from preprocessing through machine learning

- https://anaconda.org/intel
- https://software.intel.com/distribution-for-python
- https://intelpython.github.io/daal4py
- https://github.com/IntelPython/sdc
- https://medium.com/intel-analytics-software
LEGAL DISCLAIMER & OPTIMIZATION NOTICE

Performance results are based on testing as of November 27, 2019, May 18, 2020 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.

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

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.

Copyright © 2020, Intel Corporation. All rights reserved. Intel, Xeon, Core, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

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