Intel and Anaconda Extending Numba* to New highs

Sergey Maidanov (Intel), Engineering Manager for Intel® Distribution for Python*

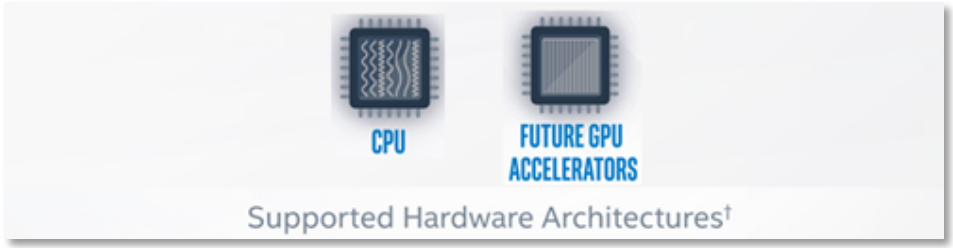
Diptorup Deb (Intel), Lead engineer for Python Compiler for GPUs

1

Scalable Dataframe Compiler

Numba* extension to compile Numpy* and Pandas* workflows

2



Extending Numba* to support Intel devices beyond CPUs

Intel® Scalable Dataframe Compiler

Extending Numba* to Compile Data Science workflows

Intel SDC in 1 minute

- Extension for Numba* to accelerate AI workflows
  - Goes beyond Numpy* operations to cover typical ETL
    - Lists, dictionaries, tuples
    - Supports more data types (series, dataframes, ASCII/Unicode strings)
- Compiler, not a library
  - Compiles Numpy, Pandas, **and the glue code** around it
  - Just-in-time and ahead-of-time compilation via Numba
- Scales from laptops to multi-core servers
  - Leverages Numba* parallel engine for auto-vectorization and multi-threading
- Open source project
  - Github page https://github.com/IntelPython/sdc
  - Documentation https://intelpython.github.io/sdc-doc/latest/index.html
- Available as conda package and pip wheels
  - `conda install -c intel/label/beta sdc`
  - `pip install -i https://pypi.anaconda.org/intel/label/beta/simple sdc`
  - Python 3.6, 3.7, Windows and Linux
  - 1700+ downloads in less than 2 months

```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)
```

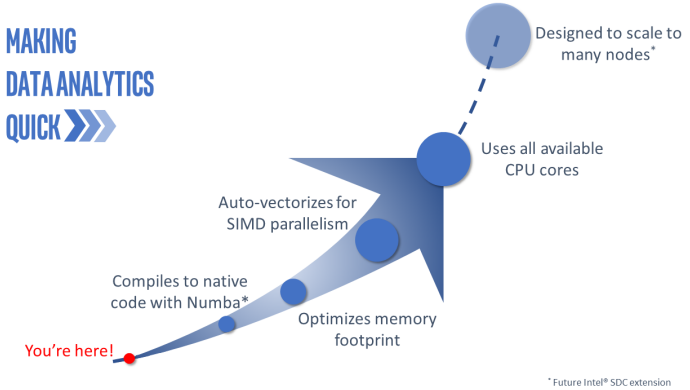
What's in Intel SDC

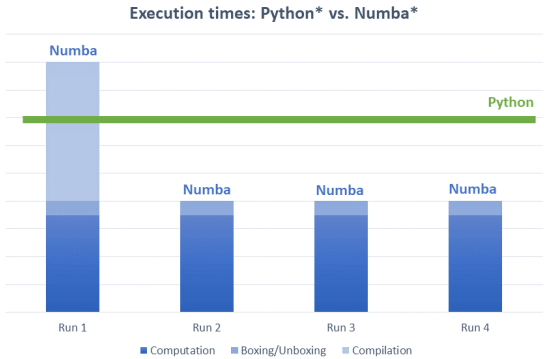
- Supports 170+ most important Pandas* APIs
  - Series, Dataframe, Window, GroupBy, read_csv
  - All major datatypes, including ASCII/Unicode strings
    - Coming soon time series and categoricals

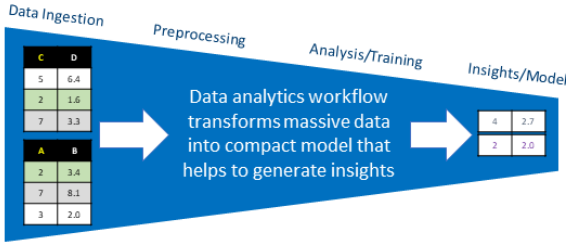
API Reference

- Input/output
  - Flat file
- Series: Columnar Data Structure
  - Constructor
  - Attributes/Operators
  - Type Conversions
  - Indexing and Iteration
  - Binary Operator Functions
  - User-Defined Functions, GroupBy, Window
  - Computations, Descriptive Statistics
  - Re-Indexing, Selection, Label Manipulation
  - Missing Data Handling
  - Re-Shaping, Sorting
  - Combining, Joining, Merging
  - Time Series
  - Accessors
  - Plotting
  - Serialization, Input-Output, Conversion
- Dataframe: Tabular Data Structure
  - Constructor
  - Attributes/Operators
  - Type Conversions
  - Indexing and Iteration
  - Binary Operator Functions
  - User-Defined Functions, GroupBy & Window
  - Computations, Descriptive Statistics
  - Re-Indexing, Selection, Label Manipulation
  - Missing Data Handling
  - Re-Shaping, Sorting, Transposing
  - Combining, Joining, Merging
  - Time Series
  - Plotting
  - Sparse Accessor
  - Serialization, Input-Output, Conversion
- Window
  - Standard moving window functions
  - Standard expanding window functions
  - Exponentially-weighted moving window functions
- GroupBy
  - Indexing, iteration
  - Function application
  - Computations / descriptive stats

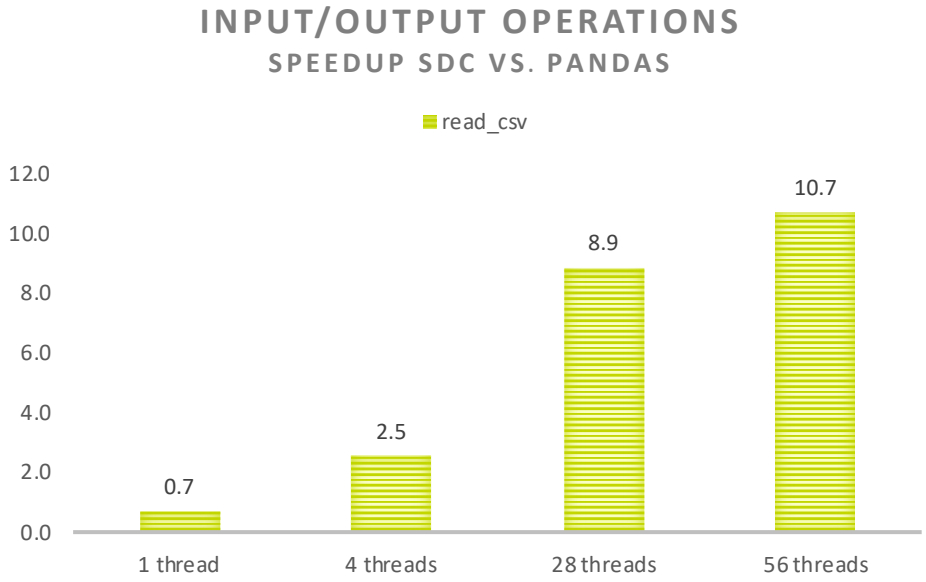
Numba/SDC performance aspects





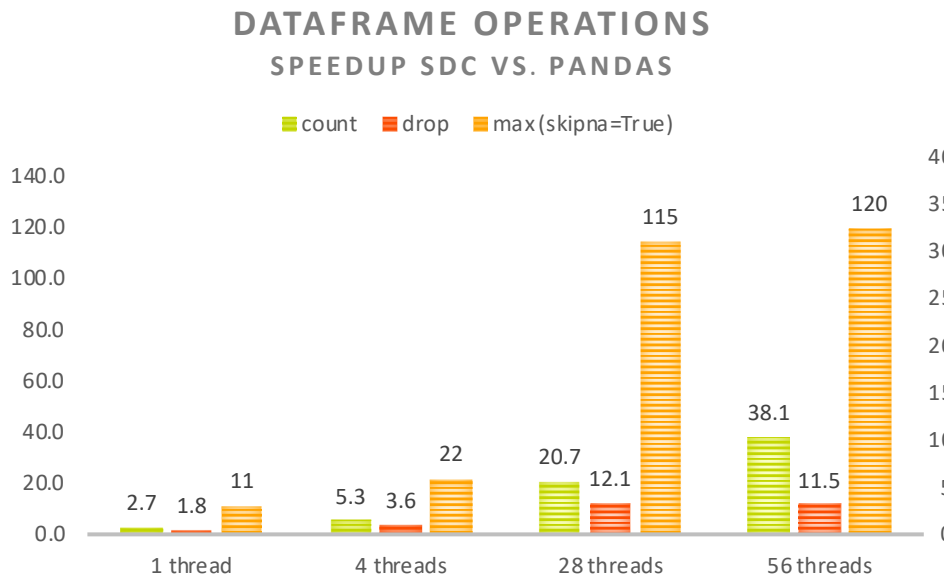


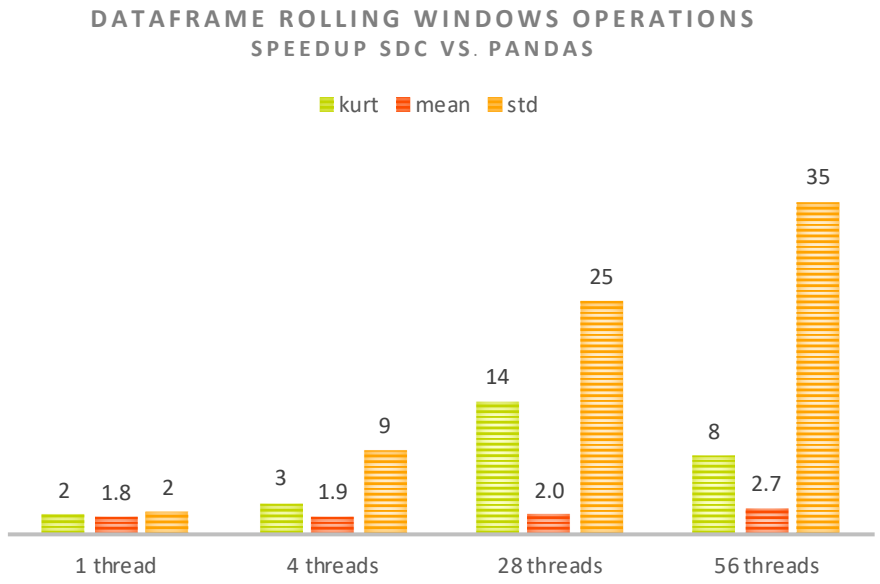
Intel SDC Performance – read_csv



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

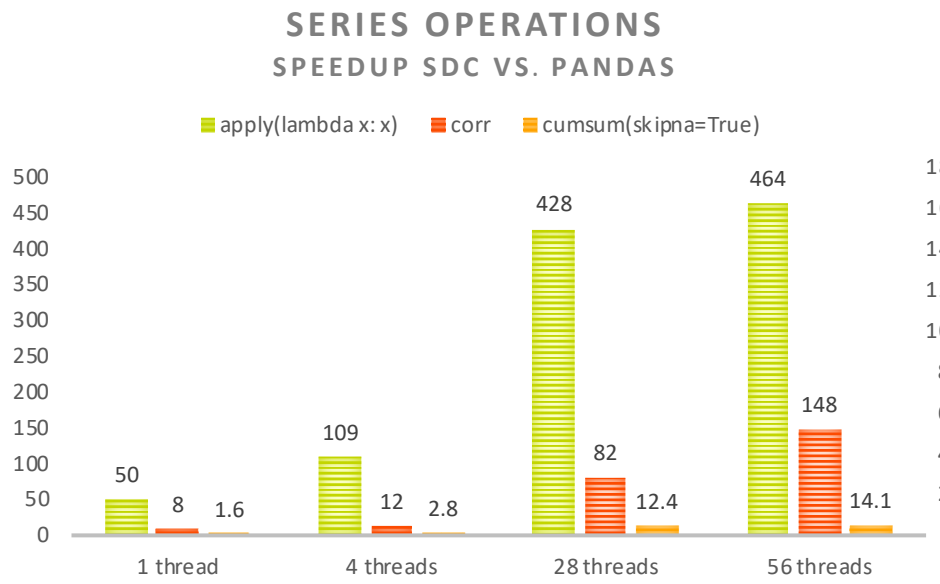
Intel SDC Performance – Dataframes

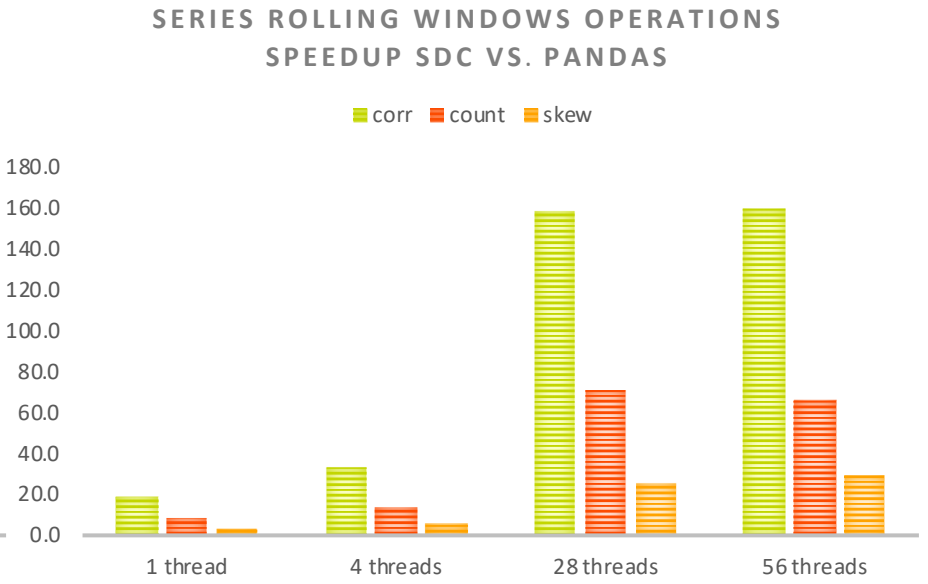




Intel® SDC Beta, Numba* 0.48, Pandas* 0.25.3
Intel® Xeon™ Platinum 8280L, 2.7 GHz, 2x28 cores, Hyperthreading=on, Turbo Mode=on

Intel SDC Performance – Series





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

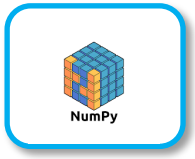
DPPY and Numba

Extending Numba* to support Intel accelerators

Envision a GPU-enabled Python Library Ecosystem

Say Hello to DPPY!!!

Data Parallel Python (DPPY)



daal4p
y

Your GPU-enabled library!

Intel is working to enable core Numpy* ecosystem with DPPY

- Lightweight Python wrapper for the SYCL runtime
- An interface for queues, events, buffers, USM, and kernels
- USM and sycl::buffer backed containers
- Allows different Python libraries to share SYCL runtime objects

Where does DPPY help me?

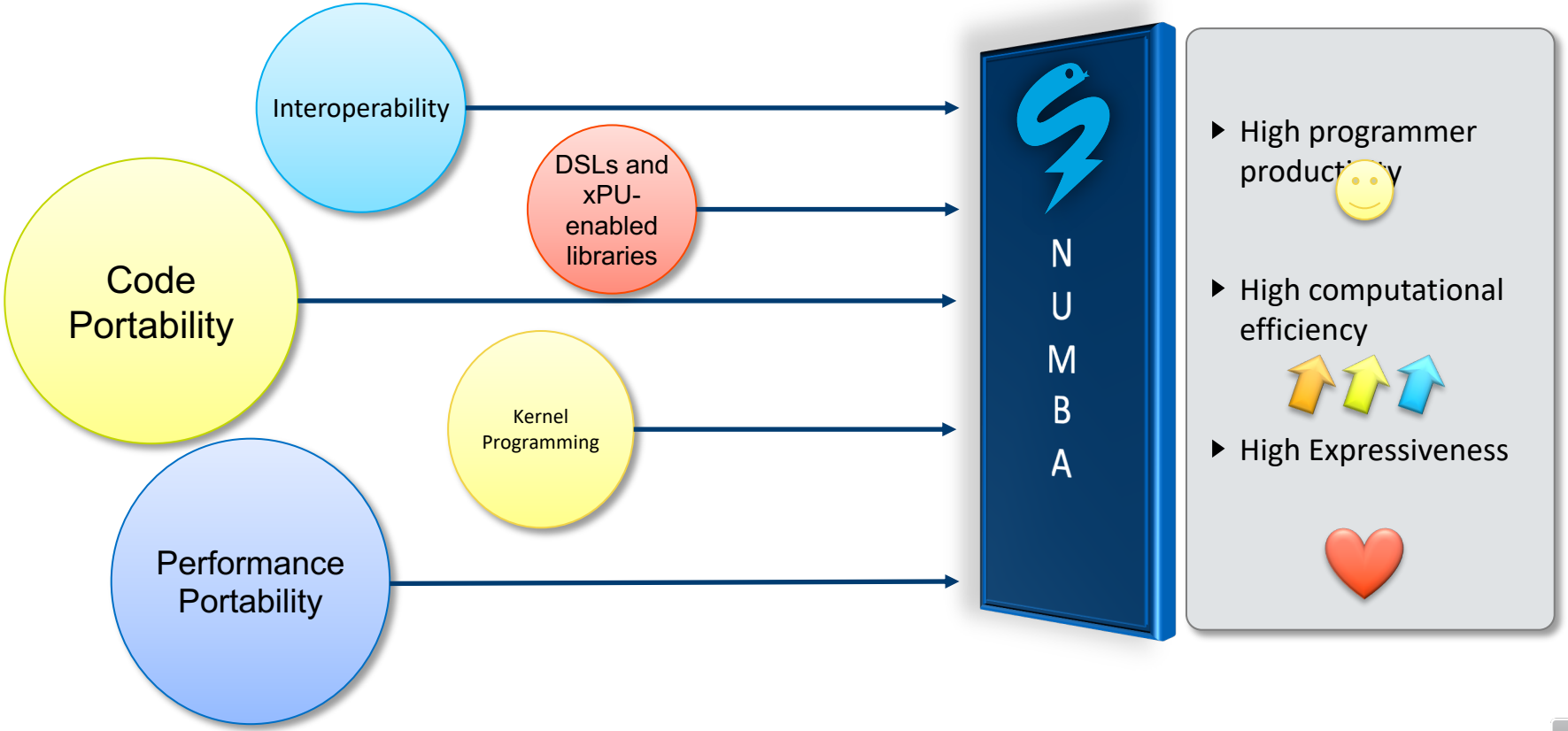
Seamless interoperability and sharing of resources

```
1  from dppy import device_context, DeviceArr
2  import daal4py as d4p
3  from numba import njit
4  import numpy as np
5
6  @njit
7  def numba_foo(a):
8      ...
9
10 a = np.array(np.random.random(1024), dtype=float32)
11
12 with device_context(gpu,0):
13     d_a = DeviceArr(a)
14     d_a = numba_foo(d_a)
15     d_a = d4p.bar(d_a)
16     a = d_a.asarray()
17
```

- Numba function
- daal4py function

Data remains on the device across different library calls!

Extending Numba* for tomorrow's Intel xPU



New Additions to Numba's Language Design

dppy.kernel

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

- Low-level kernel programming for expert ninjas

njit (target=dppy)

```
1  @njit(parallel=True, target=dppy.igpu)
2  def f1(a, b):
3      c = a + b
4      return c
```

- Domain-specific data-parallel code generation
- Compiler optimizations: kernel fusion, data movement optimizations
- Same code works for multiple devices and programming model (just change the target)

Let’s take an example

```
1   @numba.vectorize(nopython=True)
2   def cndf2(inp):
3       out = 0.5 + 0.5 * math.erf((math.sqrt(2.0)/2.0) * inp)
4       return out
5
6   @numba.njit(parallel=True, fastmath=True)
7   def blackscholes(sptprice, strike, rate, volatility, timev):
8       logterm = np.log(sptprice / strike)
9       powterm = 0.5 * volatility * volatility
10      den = volatility * np.sqrt(timev)
11      d1 = (((rate + powterm) * timev) + logterm) / den
12      d2 = d1 - den
13      NofXd1 = cndf2(d1)
14      NofXd2 = cndf2(d2)
15      futureValue = strike * np.exp(- rate * timev)
16      c1 = futureValue * NofXd2
17      call = sptprice * NofXd1 - c1
18      put  = call - futureValue + sptprice
19      return put
```

Blackscholes formula with Numba auto-parallelization on CPU using “parallel=True”

Default Numba behavior (added previously by Intel)

- **Fusion of all NumPy operations**
- **Auto-parallelization of the fused operation on CPUs using OpenMP**

Let’s take an example

```
1   @numba.vectorize(nopython=True)
2   def cndf2(inp):
3       out = 0.5 + 0.5 * math.erf((math.sqrt(2.0)/2.0) * inp)
4       return out
5
6   @numba.njit(parallel=True, fastmath=True)
7   def blackscholes(sptprice, strike, rate, volatility, timev):
8       logterm = np.log(sptprice / strike)
9       powterm = 0.5 * volatility * volatility
10      den = volatility * np.sqrt(timev)
11      d1 = (((rate + powterm) * timev) + logterm) / den
12      d2 = d1 - den
13      NofXd1 = cndf2(d1)
14      NofXd2 = cndf2(d2)
15      futureValue = strike * np.exp(- rate * timev)
16      c1 = futureValue * NofXd2
17      call = sptprice * NofXd1 - c1
18      put  = call - futureValue + sptprice
19      return put
```

Blackscholes formula with Numba auto-parallelization on CPU using “parallel=True”

Now let us make the same code work on Intel’s Gen9 GPUs…

… multiple options

Just change a decorator option

```
1   @numba.vectorize(nopython=True)
2   def cndf2(inp):
3       out = 0.5 + 0.5 * math.erf((math.sqrt(2.0)/2.0) * inp)
4       return out
5
6   @numba.njit(parallel=True, target=dppy.igpu)
7   def blackscholes(sptprice, strike, rate, volatility, timev):
8       logterm = np.log(sptprice / strike)
9       powterm = 0.5 * volatility * volatility
10      den = volatility * np.sqrt(timev)
11      d1 = (((rate + powterm) * timev) + logterm) / den
12      d2 = d1 - den
13      NofXd1 = cndf2(d1)
14      NofXd2 = cndf2(d2)
15      futureValue = strike * np.exp(- rate * timev)
16      c1 = futureValue * NofXd2
17      call = sptprice * NofXd1 - c1
18      put  = call - futureValue + sptprice
19      return put
```

- "target=dppy.igpu" triggers GPU code-generation and execution
- All other goodness like kernel fusion work out of the box!
- Numba handles all host-side control code (buffer creation, kernel enqueuing, data movement)

More power to the programmer

Type-driven JIT compilation

```
1   @numba.njit(parallel=True)
2   def blackscholes(sptprice, strike, rate, volatility, timev):
3
4   with device_context(gpu,0):
5       d_SP = DeviceArr(sptprice)
6       d_S = DeviceArr(strike)
7       d_R = DeviceArr(rate)
8       d_V = DeviceArr(volatility)
9       d_T = DeviceArr(timev)
10      blackscholes(d_SP, d_S, d_R, d_V, d_T)
11
```

- No change to njit options
- Define a GPU execution context
- Explicitly allocate memory (DeviceArr) on the GPU
- Since DeviceArr is passed in, Numba infers programmer wants GPU code

What about us ninjas?

Blackscholes dppy.kernel edition

```
@dppy.kernel
def black_scholes_dppy(callResult, putResult, S, X, T, R, V):
    i = dppy.get_global_id(0)
    if i >= S.shape[0]:
        return
    sqrtT = math.sqrt(T[i])
    d1 = (math.log(S[i] / X[i]) + (R + 0.5 * V * V) * T[i]) / (V * sqrtT)
    d2 = d1 - V * sqrtT

    K = 1.0 / (1.0 + 0.2316419 * math.fabs(d1))
    cndd1 = (RSQRT2PI * math.exp(-0.5 * d1 * d1) *
             (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5))))))
    if d1 > 0:
        cndd1 = 1.0 - cndd1

    K = 1.0 / (1.0 + 0.2316419 * math.fabs(d2))
    cndd2 = (RSQRT2PI * math.exp(-0.5 * d2 * d2) *
             (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5))))))
    if d2 > 0:
        cndd2 = 1.0 - cndd2

    expRT = math.exp((-1. * R) * T[i])
    callResult[i] = (S[i] * cndd1 - X[i] * expRT * cndd2)
    putResult[i] = (X[i] * expRT * (1.0 - cndd2) - S[i] * (1.0 - cndd1))

```

and Python*

The breadth of Intel technologies for entire Python ecosystem

Industry Initiative

oneAPI

Intel Product

Besides best native tools **all** oneAPI™ toolkits include
Intel® Distribution for Python* (IDP) ★ with **all core packages optimized**

Toolkits Tailored to Your Needs

Domain-specific sets of tools to get your job done quickly.

Intel® oneAPI Base Toolkit ★

A core set of high-performance tools for building Data Parallel C++ applications and oneAPI library based applications

Intel® oneAPI HPC Toolkit ★

Everything HPC developers need to deliver fast C++, Fortran, & OpenMP* applications that scale

Intel® oneAPI IoT Toolkit ★

Tools for building high-performing, efficient, reliable solutions that run at the network's edge

Intel® oneAPI DL Framework Developer Toolkit ★

Optimized libraries to build deep learning frameworks or customize existing ones

Intel® oneAPI Rendering Toolkit ★

Powerful rendering libraries to create high-performance, high-fidelity visualization applications

Toolkits Powered by oneAPI

Intel® System Bring-Up Toolkit ★

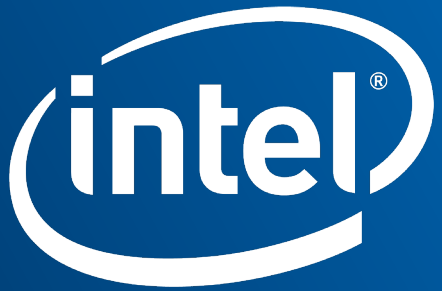
Tools to debug & tune power & performance in pre- & post-silicon development

Intel® Distribution of OpenVINO™ Toolkit ★

Deploy high-performance inference applications from device to cloud. (production-level tool)

Intel® AI Analytics Toolkit ★

Speed AI development with tools for DL training, inference, and data 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.

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