Choose the Best Accelerated Technology

How to accelerate Classical Machine Learning on Intel Architecture

Roy Allela—Deep Learning SW Engineer
roy.m.allela@intel.com
15.06.2021
Agenda

- Recap: Intel AI Analytics Toolkit
- Intel Distribution for Python
- Intel Distribution of Modin
- Intel(R) Extension for Scikit-learn
- XGBoost Optimization
- Data Parallel Python
Accelerate end-to-end AI and data analytics pipelines with libraries optimized for Intel® architectures

Who Uses It?
Data scientists, AI researchers, ML and DL developers, AI application developers

Top Features/Benefits
- Deep learning performance for training and inference with Intel optimized DL frameworks and tools
- Drop-in acceleration for data analytics and machine learning workflows with compute-intensive Python packages

Deep Learning
- Intel® Optimization for TensorFlow
- Intel® Optimization for PyTorch
- Intel® Low Precision Optimization Tool
- Model Zoo for Intel® Architecture

Data Analytics & Machine Learning
- Accelerated Data Frames
  - Intel® Distribution of Modin
  - OmniSci Backend
  - Intel® Distribution for Python
    - XGBoost
    - Scikit-learn
    - Daal-4Py
    - NumPy
    - SciPy
    - Pandas

Samples and End2End Workloads
- CPU
- GPU

Supported Hardware Architectures

Hardware support varies by individual tool. Architecture support will be expanded over time. Other names and brands may be claimed as the property of others.

Get the Toolkit HERE or via these locations
- Intel Installer
- Docker
- Apt, Yum
- Conda
- Intel® DevCloud

Learn More: software.intel.com/oneapi/ai-kit
AI Software Stack for Intel® XPU

Intel offers a robust software stack to maximize performance of diverse workloads.
AI Software Stack for Intel® XPU

Intel offers a robust software stack to maximize performance of diverse workloads.

- **Intel® Software Stack**
  - E2E Workloads (Census, NYTaxi, Mortgage...)
  - Intel® Low Precision Optimization Tool
  - Model Zoo for Intel® Architecture
  - Open Model Intel® OpenVINO™ Toolkit
  - Intel® oneAPI Base Toolkit
    - DPC++ / DPPY
    - oneMKL
    - oneDNN
    - oneVPL

- **AI Analytics Toolkit**
  - Develop DL models in Frameworks, ML & Analytics in Python
  - pandas, numpy, xgboost, TensorFlow, PyTorch, DPC++ / DPPY, oneMKL, oneDNN, oneVPL

- **DL/ML Tools**
  - Model Optimizer
  - Inference Engine
  - Intel® Low Precision Optimization Tool
  - Intel® oneAPI Base Toolkit

- **DL/ML Middleware & Frameworks**
  - Intel® AI Analytics Toolkit
  - Develop DL models in Frameworks, ML & Analytics in Python
  - Intel® OpenVINO™ Toolkit
  - Deploy DL models

- **Model Zoo for Intel® Architecture**

- **Full Set of AI ML and DL Software Solutions Delivered with Intel’s oneAPI Ecosystem**
Executive Summary

- Intel® Distribution for Python covers major usages in HPC and Data Science
- Achieve faster Python application performance — right out of the box — with minimal or no changes to a code
- Accelerate NumPy*, SciPy*, and scikit-learn* with integrated Intel® Performance Libraries such as Intel® oneMKL (Math Kernel Library) and Intel® oneDAL (Data Analytics Library)
- Access the latest vectorization and multithreading instructions, Numba* and Cython*, composable parallelism with Threading Building Blocks, and more

Analysts
Data Scientists
Machine Learning Developers
Intel® Distribution for Python

oneAPI Powered

Develop fast, performant Python code with this set of essential computational packages

Who Uses It?

- Machine Learning Developers, Data Scientists, and Analysts can implement performance-packed, production-ready scikit-learn algorithms
- Numerical and Scientific Computing Developers can accelerate and scale the compute-intensive Python packages NumPy, SciPy, and mpi4py
- High-Performance Computing (HPC) Developers can unlock the power of modern hardware to speed up your Python applications

Initial GPU support enabled with Data Parallel Python

Hardware support varies by individual tool. Architecture support will be expanded over time. Other names and brands may be claimed as the property of others.
Intel® Distribution for Python Architecture

**Command Line**

```python
> python script.py
```

**Scientific Environments**

- PyCharm
- Spyder
- Eclipse
- Sublime Text
- Visual Studio
- Visual Studio Code

**Developer Environments**

**Intel® Distribution for Python Packages**

- NumPy
- SciPy
- Learn
- daal4py
- oneDAL
- tbb4py
- smp
- mpi4py
- TBB
- iomp
- impi

**Intel® Distribution for Python Native Technologies**

- oneMKL
- DPC++
- oneDAL

**Parallelism**

- Community technology
- Intel technology
# Intel® Distribution for Python

## Developer Benefits

<table>
<thead>
<tr>
<th>Maximize Performance</th>
<th>Minimize Development Cost</th>
<th>Vast Ecosystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Libraries, Parallelism, Multithreading, Language Extensions</td>
<td>Drop-in Python Replacement</td>
<td>Familiar usage and compatibility</td>
</tr>
<tr>
<td>Near-native performance comes through acceleration of core Python numerical packages</td>
<td>Prebuilt optimized packages for numerical computing, machine/deep learning, HPC, &amp; data analytics</td>
<td>Supports Python 3</td>
</tr>
<tr>
<td>Accelerated NumPy/SciPy/scikit-learn with oneMKL &amp; oneDAL</td>
<td>Data-Parallel Python provides cross-architecture XPU support</td>
<td>Supports conda &amp; pip package managers</td>
</tr>
<tr>
<td>Data analytics, machine learning &amp; deep learning with scikit-learn, XGBoost, Modin, daal4py</td>
<td>Conda build recipes included in packages</td>
<td>Packages available via conda, pip YUM/APT, Docker image on DockerHub</td>
</tr>
<tr>
<td>Scale with Numba*, Cython*, tbb4py, mpi4py, SDC</td>
<td>Free download &amp; free for all uses including commercial deployment</td>
<td>Commercial support through the Intel® oneAPI Base Toolkit</td>
</tr>
<tr>
<td>Optimized for latest Intel® architectures</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operating Systems:** Windows*, Linux*, MacOS*

**Intel® Architecture Platforms:**

- [CPU](#)
- [GPU](#)
- [Other accel.](#)
Choose Your Download Option

Python Solutions

Tools and frameworks to accelerate end-to-end data science and analytics pipelines

Intel® AI Analytics Toolkit

Develop fast, performant Python code with essential computational packages

Intel® Distribution for Python

Optimized Python packages from package managers and containers

Conda | YUM | APT | Docker

Develop in the Cloud

Intel® DevCloud | Intel® DevCloud

* Also available in the Intel® oneAPI Base Toolkit
Data Science Landscape: Today

Tools efficient for O(1MB)

- Usable but not scalable
  - pandas
  - $y_i = \beta' x_i + \epsilon_i$

Tools efficient for O(100s GB+)

- Scalable but not usable
  - Apache Spark

Intel® AI Analytics Toolkit

Powered by oneAPI

Choose the best accelerated technology – the software doesn't decide for you

Learn More:
intel.com/oneAPI
- AIKit
Data Science Landscape: Today

Tools efficient for O(1MB)

Usable and scalable

MODIN

Tools efficient for O(100s GB+)

Scalable and usable

\[ y_{il} = \beta' x_{il} + \mu_i + \epsilon_{il} \]

pandas
Intel distribution of Modin

- Accelerate your Pandas* workloads across multiple cores and multiple nodes
- No upfront cost to learning a new API
  - import modin.pandas as pd
- In the backend, Intel Distribution of Modin is supported by Omnisci*, a performant framework for end-to-end analytics that has been optimized to harness the computing power of existing and emerging Intel® hardware
Intel distribution of Modin

- Recall: No upfront cost to learning a new API
  - import modin.pandas as pd
- Integration with the Python* ecosystem
- Integration with Ray*/Dask* clusters (Run on what you have, even on laptop!)
- To use Modin, you do not need to know how many cores your system has, and you do not need to specify how to distribute the data

Pandas* on Big Machine

Modin on Big Machine
Modin Layered Architectural Diagram - NOW

- Pandas*
- SQL* (WIP)
- Modin API Query Compiler (WIP)
- Dataframe Algebra API (WIP)
- Pyarrow Kernels (WIP)
- Pandas* Kernels
- Execution API Layer
- Dask*
- Ray*
- Multiprocessing
- CPU
Modin

```python
import modin.pandas as pd
import numpy as np

def run_etl():
    def cat_converter(x):
        if x is None:
            return np.int32(0)
        else:
            return np.int32(int(x, 16))

    names = [f'column_{i}' for i in range(40)]
    converter = {names[i]: cat_converter for i in range(14, 40)}

    df = pd.read_csv('data.csv', delimiter='\t', names=names, converters=converter)

    count_y = df.groupby("column_0")["0"].count()

    return df, count_y

df, count_y = run_etl()
```

- Dataset size: 2.4GB

Execution time Pandas vs. Modin[ray]

![Chart showing speedup of 10.8 between Pandas and Modin]

Intel® Xeon™ Gold 6248 CPU @ 2.50GHz, 2x20 cores
End-to-End Data Pipeline Acceleration

- **Workload**: Train a model using 50yrs of Census dataset from IPUMS.org to predict income based on education

- **Solution**: Intel Modin for data ingestion and ETL, Daal4Py and Intel scikit-learn for model training and prediction

- **Perf Gains**:
  - Read_CSV (Read from disk and store as a dataframe) : **6x**
  - ETL operations : **38x**
  - Train Test Split : **4x**
  - ML training (fit & predict) with Ridge Regression : **21x**

Intel(R) Extension for Scikit-learn
The most popular ML package for Python*

scikit-learn
Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification
Identifying to which category an object belongs to.
Applications: Spam detection, Image recognition.
Algorithms: SVM, nearest neighbors, random forest, ...

Regression
Predicting a continuous-valued attribute associated with an object.
Applications: Drug response, Stock prices.
Algorithms: SVR, ridge regression, Lasso, ...

Clustering
Automatic grouping of similar objects into sets.
Applications: Customer segmentation, Grouping experiment outcomes.
Algorithms: k-Means, spectral clustering, mean-shift, ...

*scikit-learn is a popular ML library for Python.
oneAPI Data Analytics Library (oneDAL)
Optimized building blocks for all stages of data analytics on Intel Architecture

GitHub: https://github.com/oneapi-src/oneDAL
Intel(R) Extension for Scikit-learn

Common Scikit-learn

- from sklearn.svm import SVC
- X, Y = get_dataset()
- clf = SVC().fit(X, y)
- res = clf.predict(X)

Scikit-learn mainline

Scikit-learn with Intel CPU opts

```python
import daal4py as d4p
d4p.patch_sklearn()
from sklearn.svm import SVC
X, Y = get_dataset()
clf = SVC().fit(X, y)
res = clf.predict(X)
```

Available through Intel conda
(conda install daal4py –c intel)

Same Code, Same Behavior

- Scikit-learn, not scikit-learn-like
- Scikit-learn conformance (mathematical equivalence) defined by Scikit-learn Consortium, continuously vetted by public CI

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

```bash
> python -m daal4py <your-scikit-learn-script>
```
Available algorithms

- Accelerated IDP Scikit-learn algorithms:
  - Linear/Ridge Regression
  - Logistic Regression
  - ElasticNet/LASSO
  - PCA
  - K-means
  - DBSCAN
  - SVC
  - train_test_split(), assume_all_finite()
  - Random Forest Regression/Classification - DAAL 2020.3
  - kNN (kd-tree and brute force) - DAAL 2020.3
Intel optimized Scikit-Learn

Same Code, Same Behavior

- Scikit-learn, not scikit-learn-like
- Scikit-learn conformance (mathematical equivalence) defined by Scikit-learn Consortium, continuously vetted by public CI

Speedup of Intel® oneDAL powered Scikit-Learn over the original Scikit-Learn

<table>
<thead>
<tr>
<th>Task</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-means fit 1M x 20, k=1000</td>
<td>3,6</td>
</tr>
<tr>
<td>K-means predict, 1M x 20, k=1000</td>
<td>4,0</td>
</tr>
<tr>
<td>PCA fit, 1M x 50</td>
<td>27,2</td>
</tr>
<tr>
<td>PCA transform, 1M x 50</td>
<td>38,3</td>
</tr>
<tr>
<td>Random Forest fit, higgs1m</td>
<td>55,4</td>
</tr>
<tr>
<td>Random Forest predict, higgs1m</td>
<td>53,4</td>
</tr>
<tr>
<td>Ridge Reg fit 10M x 20</td>
<td>91,8</td>
</tr>
<tr>
<td>Linear Reg fit 2M x 100</td>
<td>50,9</td>
</tr>
<tr>
<td>LASSO fit, 9M x 45</td>
<td>29,0</td>
</tr>
<tr>
<td>SVC fit, ijcnn</td>
<td>95,3</td>
</tr>
<tr>
<td>SVC predict, ijcnn</td>
<td>82,4</td>
</tr>
<tr>
<td>SVC fit, mnist</td>
<td>221,0</td>
</tr>
<tr>
<td>SVC predict, mnist</td>
<td>17,3</td>
</tr>
<tr>
<td>DBSCAN fit, 500K x 50</td>
<td>9,4</td>
</tr>
<tr>
<td>train_test_split, 5M x 20</td>
<td>113,4</td>
</tr>
<tr>
<td>kNN predict, 100K x 20, class=2, k=5</td>
<td>131,4</td>
</tr>
<tr>
<td>kNN predict, 20K x 50, class=2, k=5</td>
<td>113,8</td>
</tr>
</tbody>
</table>

HW: Intel Xeon Platinum 8276L CPU @ 2.20GHz, 2 sockets, 28 cores per socket;
Details: https://medium.com/intel-analytics-software/accelerate-your-scikit-learn-applications-a06cacf44912
Competitor’s Relative Performance vs. Intel® Distribution for Python* (IDP) with Scikit-learn* from the Intel® AI Analytics Toolkit

(Intel = 1)

Testing Date: Performance results are based on testing by Intel as of October 23, 2020 and may not reflect all publicly available security updates.

Configuration Details and Workload Setup: Intel® oneDAL beta10, Scikit-learn 0.23.1, Intel® Distribution for Python 3.7, Intel® AI Analytics Toolkit 2021.1, Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz, 2 sockets, 28 cores per socket, microcode: 0x4003003, total available memory 376 GB, 12X32GB modules, DDR4. AMD Configuration: AMD Rome 7742 @2.25 GHz, 2 sockets, 64 cores per socket, microcode: 0x8301038, total available memory 376 GB, 12X32GB modules, DDR4, Intel® Distribution for Python 3.7. NVIDIA Configuration: Nvidia Tesla V100-16Gb, total available memory 376 GB, 12X32GB modules, DDF4, Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz, 2 sockets, 28 cores per socket, microcode: 0x5003003, cuDF 0.15, cuML 0.15, CUDA 10.2.89, driver 440.33.01, Operation System: CentOS Linux 7 (Core), Linux 4.19.36 kernel.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex. Your costs and results may vary.
Demo
XGBoost
Gradient Boosting - Overview

Gradient Boosting:

• Boosting algorithm (Decision Trees - base learners)
• Solve many types of ML problems (classification, regression, learning to rank)
• Highly-accurate, widely used by Data Scientists
• Compute intensive workload
• Known implementations: XGBoost*, LightGBM*, CatBoost*, Intel® oneDAL, ...
Gradient Boosting Acceleration – gain sources

Pseudocode for XGBoost* (0.81) implementation

```
def ComputeHist(node):
    hist = []
    for i in samples:
        for f in features:
            bin = bin_matrix[i][f]
            hist[bin].g += g[i]
            hist[bin].h += h[i]
    return hist

def BuildLvl:
    for node in nodes:
        ComputeHist(node)
    for node in nodes:
        FindBestSplit(node, f)
    for node in nodes:
        SamplePartition(node)
```

- Memory prefetching to mitigate irregular memory access
- Usage of AVX-512, vcompress instruction (from Skylake)
- Usage uint8 instead of uint32
- SIMD instructions instead of scalar code
- Nested parallelism
- Advanced parallelism, reducing seq loops
- Already available in Intel® DAAL, potential optimizations for XGBoost*
- Moved from Intel® oneDAL to XGBoost (v1.3)

Pseudocode for Intel® oneDAL implementation

```
def ComputeHist(node):
    hist = []
    for i in samples:
        for f in features:
            bin = bin_matrix[i][f]
            bin_value = load(hist[2*bin])
            bin_value = add(bin_value, gh[i])
            store(hist[2*bin], bin_value)
    return hist

def BuildLvl:
    parallel for node in nodes:
        ComputeHist(node)
    parallel for node in nodes:
        FindBestSplit(node, f)
    parallel for node in nodes:
        SamplePartition(node)
```

Legend:
XGBoost* fit CPU acceleration ("hist" method)

XGBoost fit - acceleration against baseline (v0.81) on Intel CPU

+ Reducing memory consumption

<table>
<thead>
<tr>
<th>memory, Kb</th>
<th>Airline</th>
<th>Higgs1m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>28311860</td>
<td>1907812</td>
</tr>
<tr>
<td>#5334</td>
<td>16218404</td>
<td>1155156</td>
</tr>
<tr>
<td>reduced:</td>
<td>1.75</td>
<td>1.65</td>
</tr>
</tbody>
</table>

**CPU configuration:**
c5.24xlarge AWS Instance, CLX 8275 @ 3.0GHz, 2 sockets, 24 cores per socket, HT:on, DRAM (12 slots / 32GB / 2933 MHz)
**XGBoost* CPU vs. GPU**

* XGBoost* fit v1.1 CPU vs GPU speed-up, (higher is better for Intel)


**CPU:** c5.18xlarge AWS Instance (2 x Intel® Xeon Platinum 8124M @ 18 cores, OS: Ubuntu 20.04.2 LTS, 193 GB RAM.

**GPU:** p3.2xlarge AWS Instance (GPU: NVIDIA Tesla V100 16GB, 8 vCPUs), OS: Ubuntu 18.04.2 LTS, 61 GB RAM.

**SW:** XGBoost 1.1: build from sources. compiler – G++ 7.4, nvcc 9.1. Intel DAAL: 2019.4 version, downloaded from conda. Python env: Python 3.6, Numpy 1.16.4, Pandas 0.25, Scikit-learn 0.21.2.

**Testing Date:** 5/18/2020
XGBoost* and LightGBM* Prediction Acceleration with Daal4Py

- Custom-trained XGBoost* and LightGBM* Models utilize Gradient Boosting Tree (GBT) from Daal4Py library for performance on CPUs

- No accuracy loss; 23x performance boost by simple model conversion into daal4py GBT:

```python
# Train common XGBoost model as usual
xgb_model = xgb.train(params, X_train)
import daal4py as d4p  
# XGBoost model to DAAL model
daal_model = d4p.get_gbt_model_from_xgboost(xgb_model)  
# make fast prediction with DAAL
daal_prediction = d4p.gbt_classification_prediction(...).compute(X_test, daal_model)
```

- Advantages of daal4py GBT model:
  - More efficient model representation in memory
  - Avx512 instruction set usage
  - Better L1/L2 caches locality

For more complete information about performance and benchmark results, visit [www.intel.com/benchmarks](http://www.intel.com/benchmarks).
See backup for configuration details.
Demo
QnA
Backup slides
Envision a GPU-enabled Python Library Ecosystem

Data Parallel Python

Extending PyData ecosystem for XPU

Unified Python Offload Programming Model

```
with device_context("gpu"):
    a_dparray = dpnp.random.random(1024, 3)
    X_dparray = numba.njit(compute_embedding)(a_dparray)
    res_dparray = daal4py.kmeans().compute(X_dparray)
```

Optimized Packages for Intel CPUs & GPUs

- NumPy
- XGBoost
- TensorFlow
- PyTorch

Jit Compilation

Unified Data & Execution Infrastructure

- zero-copy USM array interface
- common device execution queues

DPC++ RUNTIME

- OpenCL
- Level 0
- CUDA

Envision a GPU-enabled Python Library Ecosystem

Data Parallel Python

NDA Presentation

ndarray

numpy → dpnp

→ dparray

→ XPU

Unified Python Offload Programming Model

```
with device_context("gpu"):
    a_dparray = dpnp.random.random(1024, 3)
    X_dparray = numba.njit(compute_embedding)(a_dparray)
    res_dparray = daal4py.kmeans().compute(X_dparray)
```

Optimized Packages for Intel CPUs & GPUs

- NumPy
- XGBoost
- TensorFlow
- PyTorch

Jit Compilation

Unified Data & Execution Infrastructure

- zero-copy USM array interface
- common device execution queues

DPC++ RUNTIME

- OpenCL
- Level 0
- CUDA

Envision a GPU-enabled Python Library Ecosystem

Data Parallel Python

NDA Presentation

ndarray

numpy → dpnp

→ dparray

→ XPU

Unified Python Offload Programming Model

```
with device_context("gpu"):
    a_dparray = dpnp.random.random(1024, 3)
    X_dparray = numba.njit(compute_embedding)(a_dparray)
    res_dparray = daal4py.kmeans().compute(X_dparray)
```

Optimized Packages for Intel CPUs & GPUs

- NumPy
- XGBoost
- TensorFlow
- PyTorch

Jit Compilation

Unified Data & Execution Infrastructure

- zero-copy USM array interface
- common device execution queues

DPC++ RUNTIME

- OpenCL
- Level 0
- CUDA
Scikit-Learn on XPU

**Stock on Host:**

```python
from sklearn.svm import SVC
X, Y = get_dataset()
clf = SVC().fit(X, y)
res = clf.predict(X)
```

**Optimized on Host:**

```python
import daal4py as d4p
d4p.patch_sklearn()

from sklearn.svm import SVC
X, Y = get_dataset()
clf = SVC().fit(X, y)
res = clf.predict(X)
```

**Offload to XPU:**

```python
import daal4py as d4p
d4p.patch_sklearn()
import dpctl

from sklearn.svm import SVC
X, Y = get_dataset()
with dpctl.device_context("gpu"):
    clf = SVC().fit(X, y)
    res = clf.predict(X)
```

SAME NUMERIC BEHAVIOR as defined by Scikit-learn Consortium & continuously validated by CI

[Passed]
Installing Intel® Distribution for Python* 2021

Anaconda.org
https://anaconda.org/intel/packages

> conda create -n idp -c intel intelpython3_core python=3.x
> conda activate idp
> conda install intel::numpy

YUM/APT


docker pull intelpython/intelpython3_full

Docker Hub


oneAPI


Standalone Installer


PyPI

> pip install intel-numpy
> pip install intel-scipy
> pip install mkl_fft
> pip install mkl_random

+ Intel library Runtime packages
+ Intel development packages

Linux* Windows*

OS X*
New Additions to Numba’s Language Design

@dppy.kernel

def sum(a, b, c):
    i = dppy.get_global_id(0)
    c[i] = a[i] + b[i]

    a = np.ones(1024, dtype=np.float32)
    b = np.ones(1024, dtype=np.float32)
    c = np.zeros_like(a)

    with dpctl.device_context("gpu"):
        sum[1024, dppy.DEFAULT_LOCAL_SIZE](a, b, c)

@njit

def f1(a, b):
    c = a + b
    return c

    a = np.ones(1024, dtype=np.float32)
    b = np.ones(1024, dtype=np.float32)

    with dpctl.device_context("gpu"):
        c = f1(a, b)

Explicit kernels, Low-level kernel programming for expert ninjas

NumPy-based array programming, auto-offload, high-productivity
Seamless interoperability and sharing of resources

```
import dpctl, numba, dpnp, daal4py

@numba.njit
def compute(a):
    ...

with dpctl.device_context("gpu"):
    a_dparray = dpnp.random.random(1024, 3)
    X_dparray = compute(a_dparray)
    res_dparray = daal4py.kmeans().compute(X_dparray)
```

- Different packages share same execution context
- Data can be exchanged without extra copies and kept on the device
import numba
import numpy as np
import math

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

@numba.njit(parallel={"offload": 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

# Runs on CPU by default
blackscholes(...)

# Runs on GPU
with dpctl.device_context("gpu"):  
    blackscholes(...)

# In future
with dpctl.device_context("cuda:gpu"):  
    blackscholes(...)

Portability Across Architectures