Data Parallel Essentials For Python

Interfacing oneAPI and Python

Diptorup Deb
Oleksandr Pavlyk

Intel Corp

March 08, 2022

Agenda

Duration	Topics
5 minutes	Goals
20 minutes	Current ecosystem and core packages
5 minutes	Q&A

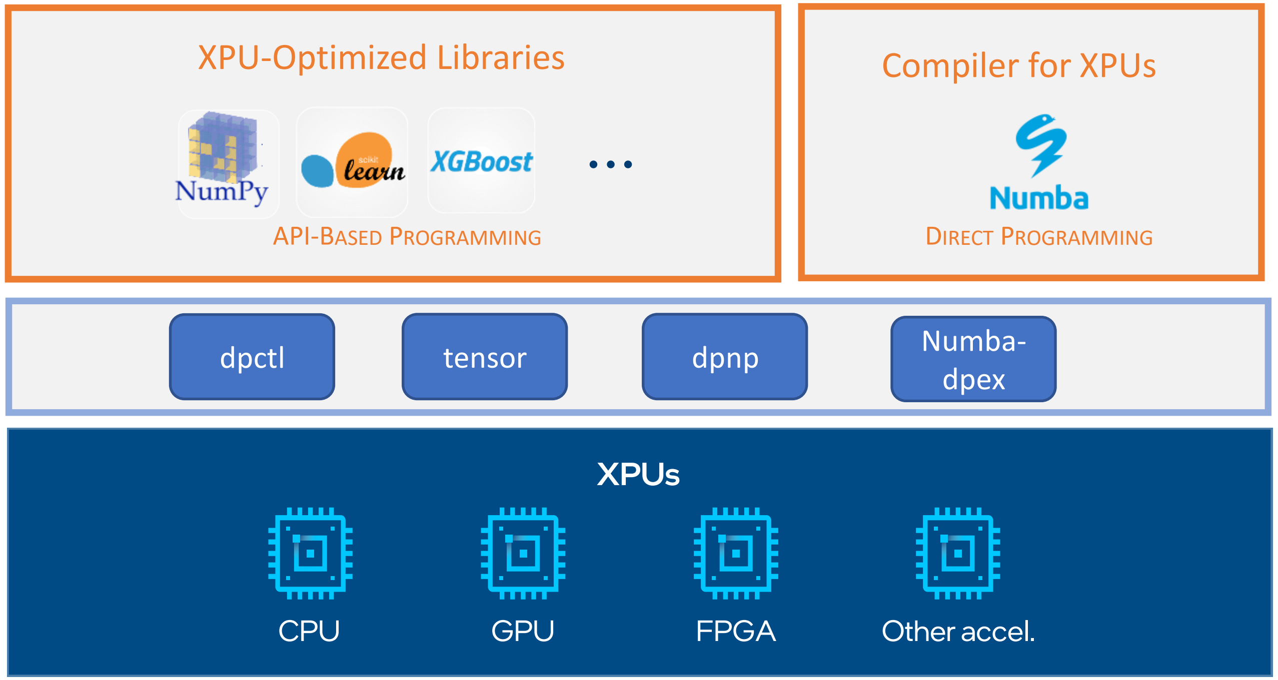
Data Parallel Essentials for Python

Fostering a oneAPI/SYCL-based ecosystem for PyDATA

PyData Ecosystem

Data Parallel Essentials for Python

oneAPI + SYCL



Core Goals

oneAPI™

- Prescribe a Pythonic offload model and interoperability API
  - offload model (compute follows data)
  - data interchange and interoperation specification
- Building blocks to foster a SYCL-based ecosystem in Python
  - SYCL USM-based Python array library (Array API standard)
  - Compilation of Python bytecode to SPIR-V for SYCL/DPC++
  - SYCL-based drop-in replacement for NumPy
- Ease Python native extension development for oneAPI and SYCL libraries

Programming Model Goals

Offload Model

- Pythonic offload model following array API spec (https://data-apis.org/array-api/latest/)
- Offload happens where data currently resides ("compute follows data")

```
X = dp.array([1,2,3])
Y = X * 4
```

executed on default device

```
X = dp.array([1,2,3], device="gpu:0")
Y = X * 4
```

executed on "gpu:0" device

```
X = dp.array([1,2,3], device="gpu:0")
Y = dp.array([1,2,3], device="gpu:1")
Z = X + Y
```

Error! Arrays are on different devices

Interoperability

Native extensions

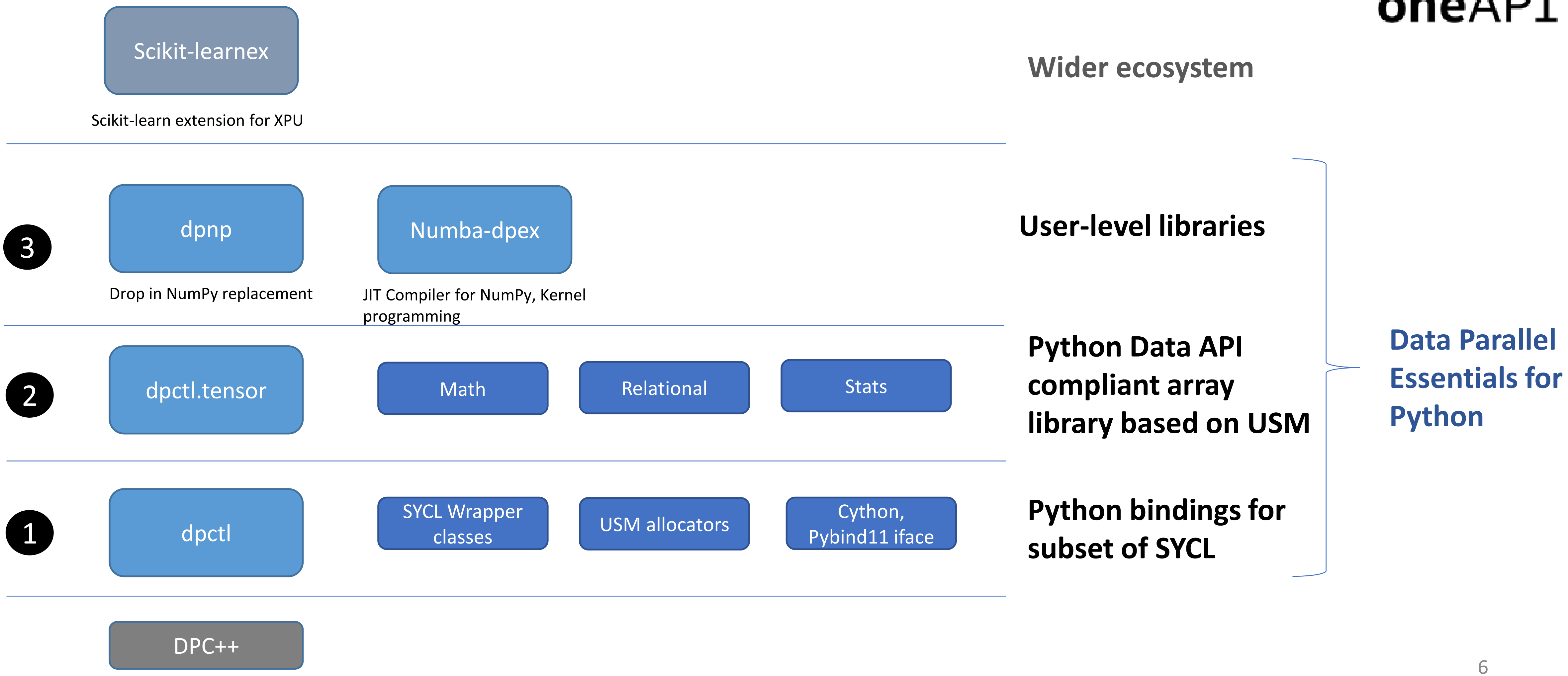
- Extend the dlpack standard (https://github.com/dmlc/dlpack)

Pure Python modules

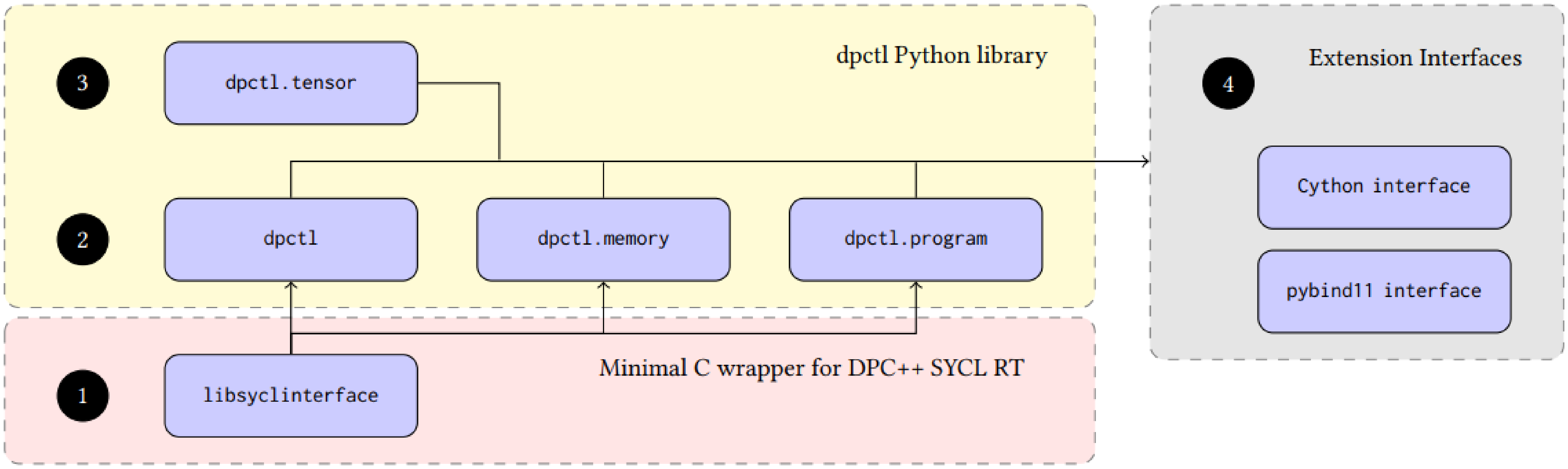
- Define a protocol like NumPy's __array_interface__ and CuPy's __cuda_array_interface__

Current Ecosystem

oneAPI



dpctl – Data parallel control



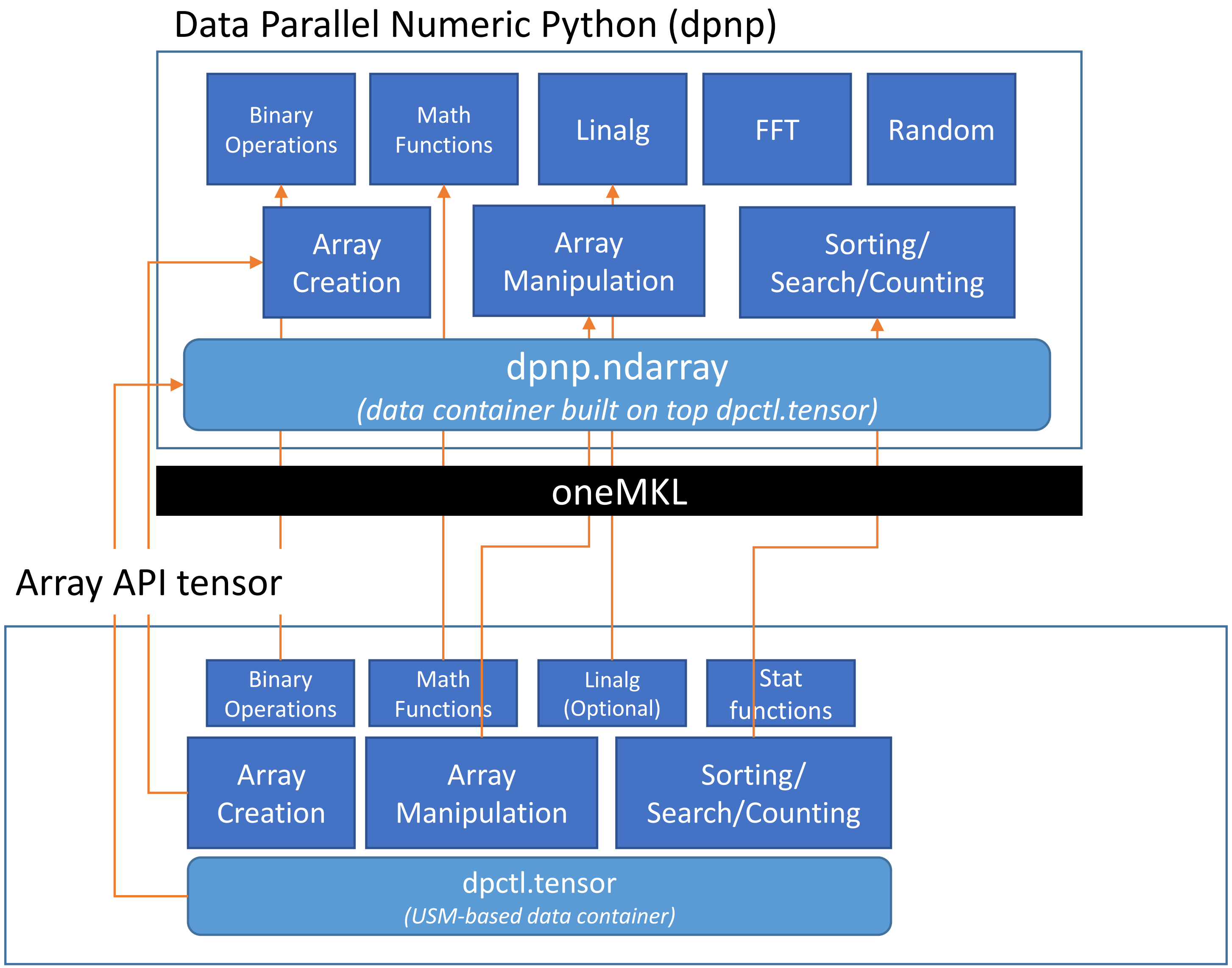
1 Library providing a minimal C API for the main DPC++ SYCL runtime classes

2 Python modules exposing SYCL runtime classes, USM allocators, and kernel bundle

3 A data API standard complaint array library supporting USM allocated memory

4 Native API to use dpctl objects in Cython and pybind11 extensions modules

Array Libraries



oneAPI™

Original CPU script

```
import numpy as np

x = np.array([[1, 1], [1, 1]])
y = np.array([[1, 1], [1, 1]])

res = np.matmul(x, y)
```

Modified XPU script

```
import dpnp as dp

x = dp.array([[1, 1], [1, 1]], device="gpu")
y = dp.array([[1, 1], [1, 1]], device="gpu")

res = dp.matmul(x, y)  # res resides on gpu
```

Extension Interfaces

```cpp
#include "dpctl4pybind11.hpp"
#include <CL/sycl.hpp>
#include <oneapi/mkl.hpp>
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>

void gemv_blocking(sycl::queue q,
                   dpt::usm_ndarray m,
                   dpt::usm_ndarray v,
                   dpt::usm_ndarray r,
                   const std::vector<sycl::event> &deps = {})
{
    auto n = m.get_shape(0);
    auto m = m.get_shape(1);
    int mat_typenum = m.get_typenum();
    /* various legality checks omitted */
    sycl::event res_ev;

    if (mat_typenum == UAR_DOUBLE) {
        auto *mat_ptr = m.get_data<double>());
        auto *v_ptr = v.get_data<double>());
        auto *r_ptr = r.get_data<double>();
        res_ev = oneapi::mkl::blas::row_major::gemv(
            q, oneapi::mkl::transpose::nontrans, n, m, 1,
            mat_ptr, m, v_ptr, 1, 0, r_ptr, 1, depends);
    }
    else
        throw std::runtime_error("unsupported");

    res_ev.wait();
}

PYBIND11_MODULE(_onemkl, m)
{
    // Import the dpctl extensions
    import_dpctl();
    m.def("gemv_blocking", &gemv_blocking, "oneMKL gemv wrapper");
}
```

- **Create a Python ext. to call onemkl::gemv in < 40 loc (fits on a slide)**
- **Invoke it seamless from Python using dpctl, dpctl.tensor**

```python
import dpctl;
import numpy as np
import dpctl.tensor as dpt
import onemkl4py

# Programmatically select a device
d = select_device()
# Create an execution queue for the selected device
q = dpctl.SyclQueue(d)
# Allocate matrices and vectors objects using NumPy
Mnp, vnp = np.random.randn(5, 3), np.random.randn(3)
# Copy data to a USM allocation
M = convert_numpy_to_tensor(Mnp, q)
v = convert_numpy_to_tensor(vnp, q)
r = dpt.empty((5,), dtype="d", sycl_queue=q)
# Invoke a binding for the oneMKL gemv kernel.
onemkl4py.gemv_blocking(M.sycl_queue, M, v, r, [])
```

Numba-dpex

oneAPI

Array-style programming

```
@njit(parallel=True)
def l2_distance(a, b, c)
    return np.sum((a-b)**2)
```

NumPy (array) style programming. Requires minimum code changes to compile existing Numpy code for XPU.
Nvidia cuPy offers this programming model with JIT fusion capabilities via cupy.fuse()

Explicit prange (parfor) loops

```
@njit(parallel=True)
def l2_distance(a, b, c)
    s = 0.0
    for i in prange(len(a))
        s += (a[i]-b[i])**2
    return s
```

Parfor-style programming. Preferred by some users when iteration space requires complex indexing.
Unique for CPU. Intel extends to XPU via numba-dpex. No CUDA alternatives to date

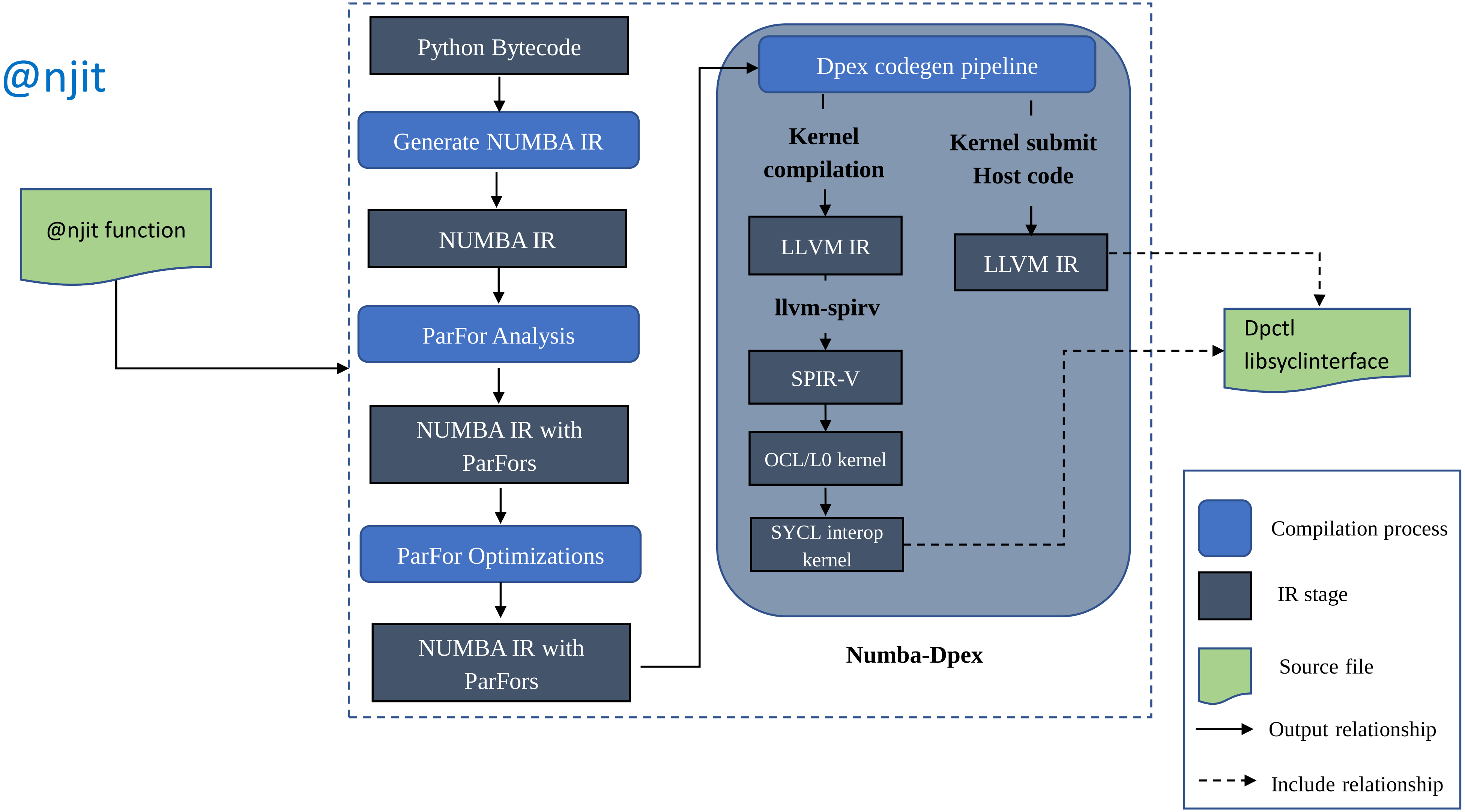
OpenCl-style kernel programming

```
@kernel(access_type={"read_only": ["a", "b"], write_only:["c"]})
def l2_distance(a, b, c)
    i = numba_dpex.get_global_id(0)
    j = numba_dpex.get_global_id(1)
    sub = a[i,j] - b[i,j]
    sq = sub ** 2
    atomic.add(c, 0, sq)
```

Most advanced programming model. Recommended to get highest performance on XPU yet avoiding DPC++.
Nvidia @cuda.jit offers this programming model in Numba

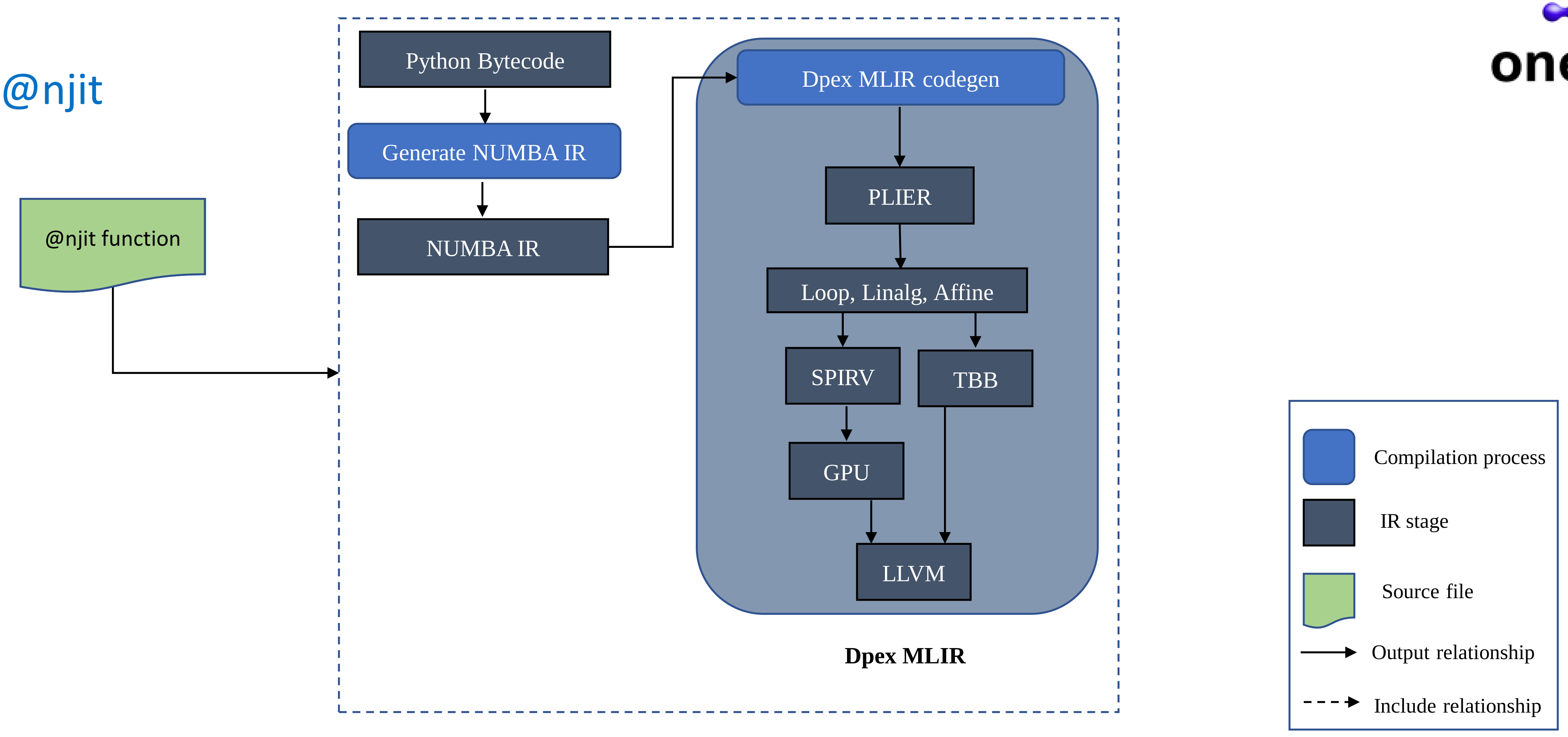
Numba-dpex codegen

@njit



Numba-dpex codegen (MLIR)

@njit



Current Status

- Included in oneAPI Basekit and Intel Distribution for Python* (IDP)
- Open-source development on github.com/IntelPython
- Packages available from Anaconda cloud and PyPi

Programming Model

Compute Follows Data

- Pythonic offload model following array API spec
- Explicit control over execution based on data placement

Interoperability

- __sycl_usm_array_interface__ modeled after NumPy's __array_interface__
- Dlpack for exchanging data across native extensions

```
import dpnp as dp
    # Case 1
    #  Allocate X on the default device
    X = dp.array([1,2,3])
    #  scaling of X executed on device of X, result
         placed into Y
    Y = X * 4
    # Case 2
    #  Allocate X on "gpu:1"
    X = dp.array([1,2,3], device="gpu:1")
    # Executed on "gpu:1"
    Y = X * 4
    # Case 3
    X1 = dp.array([1,2,3], device="gpu:1")
    X2 = dp.array([1,2,3], device="gpu:0")
    #  error!
    Y = X1 + X2

    # Arrays can be associated with another device
    # (copy is performed if needed)
    X1a = X1.to_divice(device=dev)
```