Numba-dpex: Towards SYCL-like kernel programming in Python*

February 6th, 2024

Diptorup Deb

intel®

Agenda

- Overview of Data Parallel Extensions for Python* software stack
- Deep dive into direct kernel programming in Python

Data Parallel Extensions for Python* (DPEP)

Set of Python packages that extend the oneAPI programming model to the PyData ecosystem

PyData

SciPy NumPy scikit learn XGBoost Numba matplotlib

Data Parallel Extensions for Python*

dpctl tensor dpnp numba-dpex

oneAPI

DPC++ oneMKL oneDPL

Unified Runtime API

Level Zero OpenCL CUDA ROCm

Execution Model

- Pythonic execution model that follows Python Array API standard [1]:

execution 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="cpu")
Y = X * 4
```

executed on a “cpu” device

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

executed on a “gpu” device

[1] https://data-apis.org/array-api/latest/design_topics/device_support.html

`dpctl`: Data Parallel Control

`dpctl`

What?

Low-level Python and C bindings for a sub-set of DPC++/SYCL runtime

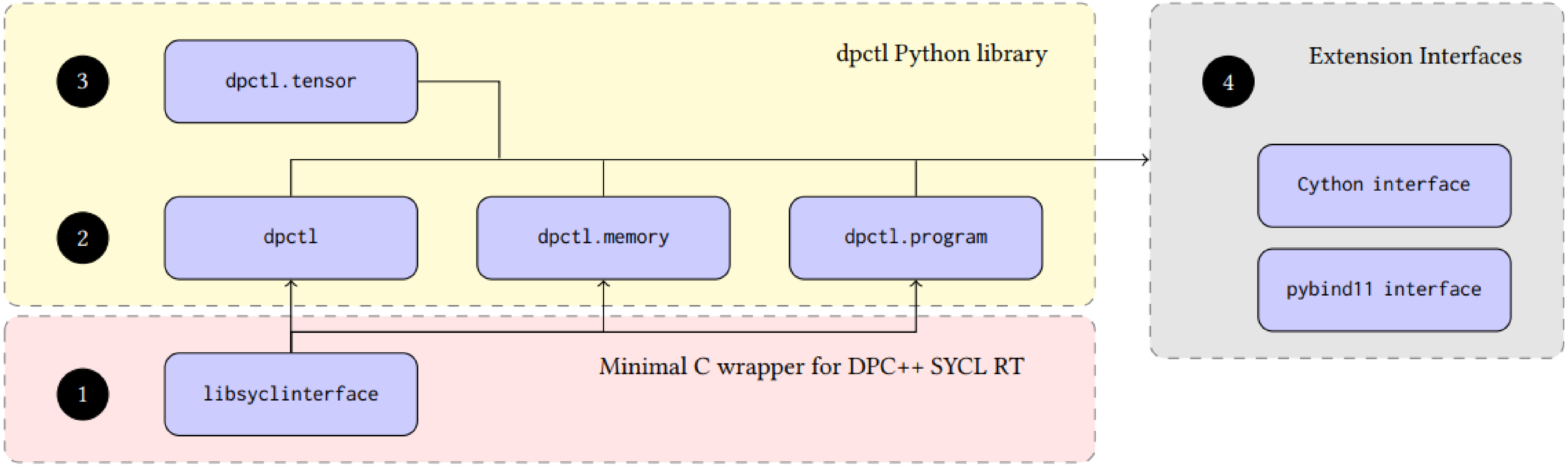
Why?

- Allow device selection, execution queue management, and device memory allocation from Python
- To support DPC++/SYCL library developers in building Python native extensions in pybind11/Cython for their applications

Benefits

- Productivity from code portability
- Freedom of using standards-based libraries

Architecture



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

A quick demo

Query all available computational platforms

```
>>> import dpctl
>>> dpctl.lsplatform()
Intel(R) OpenCL OpenCL 3.0 LINUX
Intel(R) Level-Zero 1.3
NVIDIA CUDA BACKEND CUDA 11.4
```

Three platforms were found: Intel OpenCL CPU, Intel Level Zero GPU, Nvidia GPU

```
>>> d0 = dpctl.SyclDevice()
>>> d1 = dpctl.SyclDevice("cuda:gpu")
>>> d2 = dpctl.SyclDevice("cpu")
>>> d3 = dpctl.SyclDevice("level_zero:gpu")
>>> [d.name for d in [d0, d1, d2, d3]]
['Intel(R) UHD Graphics 770', 'NVIDIA GeForce GT 1030', '12th Gen Intel(R) Core(TM) i9-
12900', 'Intel(R) UHD Graphics 770']
```

Query devices and show their properties

Note: The demo was generated using a custom dpctl build with CUDA support.
Refer https://github.com/IntelPython/dpctl/discussions/1124

Easily bind your SYCL library to Python

```
#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`**

```
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, [])
```

https://github.com/IntelPython/sample-data-parallel-extensions

Developing portable code in Pure Python

```
>>> from dpctl import tensor as dpt
>>> def foo(x):
...     d2 = dpt.remainder(x, 2) == 0
...     d3 = dpt.remainder(x, 3) == 0
...     d5 = dpt.remainder(x, 5) == 0
...     return x[ d2 & d3 & d5 ]
```

A platform independent implementation that works on any supported device

```
>>> foo(dpt.arange(1, 100))
usm_ndarray([30, 60, 90])
>>> _.device
Device(level_zero:gpu:0)

>>> foo(dpt.arange(1, 100, device="cuda"))
usm_ndarray([30, 60, 90])
>>> _.device
Device(cuda:gpu:0)

>>> foo(dpt.arange(1, 100, device="cpu"))
usm_ndarray([30, 60, 90])
>>> _.device
Device(opencl:cpu:0)
```

Portability across platforms by only changing data allocation

numba-dpex: Data Parallel Extension for Numba*

Direct kernel programming in Python

numba-dpex

What?

- JIT compiler providing a SYCL-like kernel programming API
- Partial JIT compilation for Numba array expressions to SYCL devices

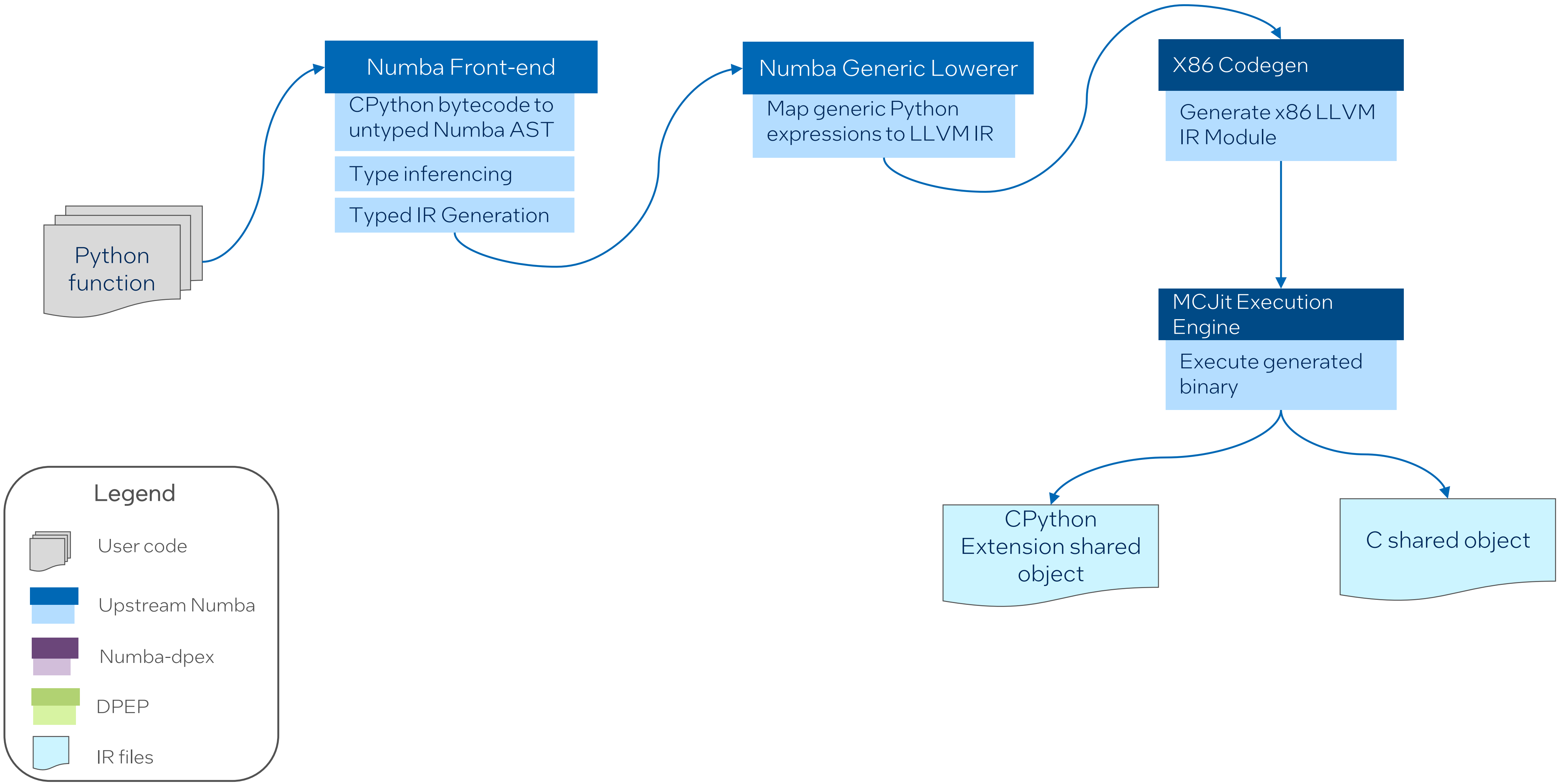
Why?

No truly heterogeneous kernel programming interface exists in Python

Benefits

- Direct kernel programming in Python for rapid prototyping and near-native (DPC++) performance
- Frees developers from developing and maintaining native extension bindings
- Compiler-driven optimizations

Numba


Numba Front-end
CPython bytecode to untyped Numba AST
Type inferencing
Typed IR Generation
Numba Generic Lowerer
Map generic Python expressions to LLVM IR
X86 Codegen
Generate x86 LLVM IR Module
MCJit Execution Engine
Execute generated binary
Python function
CPython Extension shared object
C shared object
Legend
User code
Upstream Numba
Numba-dpex
DPEP
IR files

SYCL-like API in Pure Python

Numba-dpex kernel API

```python
import math
import dpnp
import numba_dpex as dpex
from numba_dpex import kernel_api as kapi

@dpex.kernel
def pwd_kernel(item: kapi.Item, data, dist):
  i = item.get_id(0)
  j = item.get_id(1)
  data_dims = data.shape[1]
  d = data.dtype.type(0.0)
  for k in range(data_dims):
    tmp = data[i, k] - data[j, k]
    d += tmp * tmp
  dist[i, j] = math.sqrt(d)

data = <elided>
dist = <elided>
dpex.call_kernel(
  pwd_kernel, # JIT compile
  dpex.Range(data.shape[0], data.shape[1]),
  data, dist # kernel arguments
)
```

SYCL

```cpp
void pwd_sycl(queue Queue,
              size_t x1,
              size_t x2,
              size_t ndims,
              const float *p1,
              const float *p2,
              float *dist)
{
    Queue.submit([&](handler &h) {
      h.parallel_for(
        range<2>{x1, x2}, [=](id<2> myID) {
          auto i = myID[0];
          auto j = myID[1];
          float d = 0.;
          for(auto k = 0; k < ndims; k++) {
            auto tmp = p1[i*ndims+k]-p2[j*ndims+k];
            d += tmp * tmp;
          }
          dist[i * x2 + j] = sycl::sqrt(d);
        });
    });
    Queue.wait();
}
```

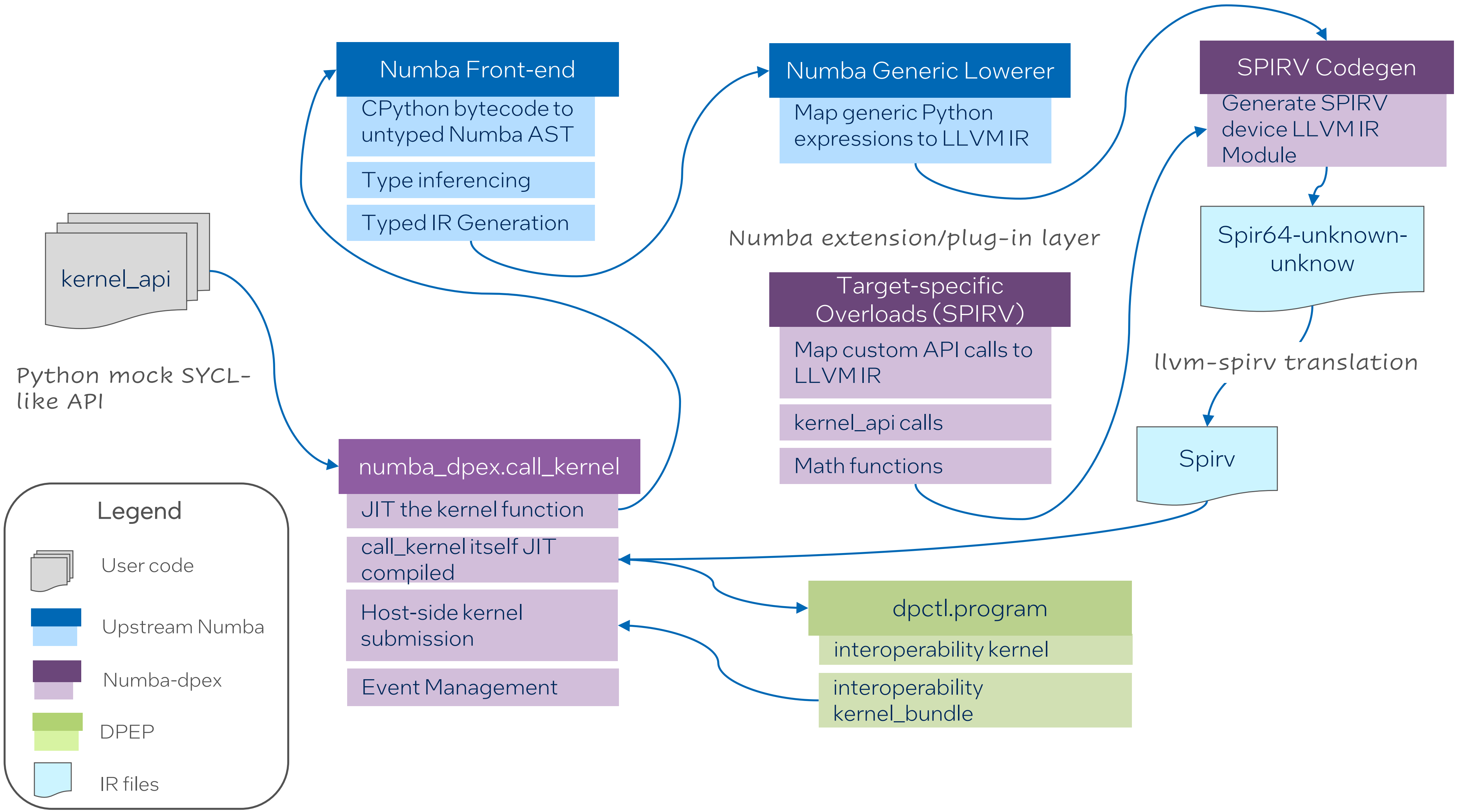
SYCL-like API in Pure Python (cont.) [as of 01/31/2024]

	SYCL classes	numba_dpex.kernel_api
Ranges	range, nd_range	Range, NdRange
Index Space ID	item, nd_item, group	Item, NdItem, Group
Local Accessors	local_accessor	dpex.local.array, LocalAccessor(in design)
Synchronizations and Atomics	group_barrier, atomic_fence, atomic_ref	group_barrier, atomic_fence, AtomicRef
Event	event	*dpctl.SyclEvent*

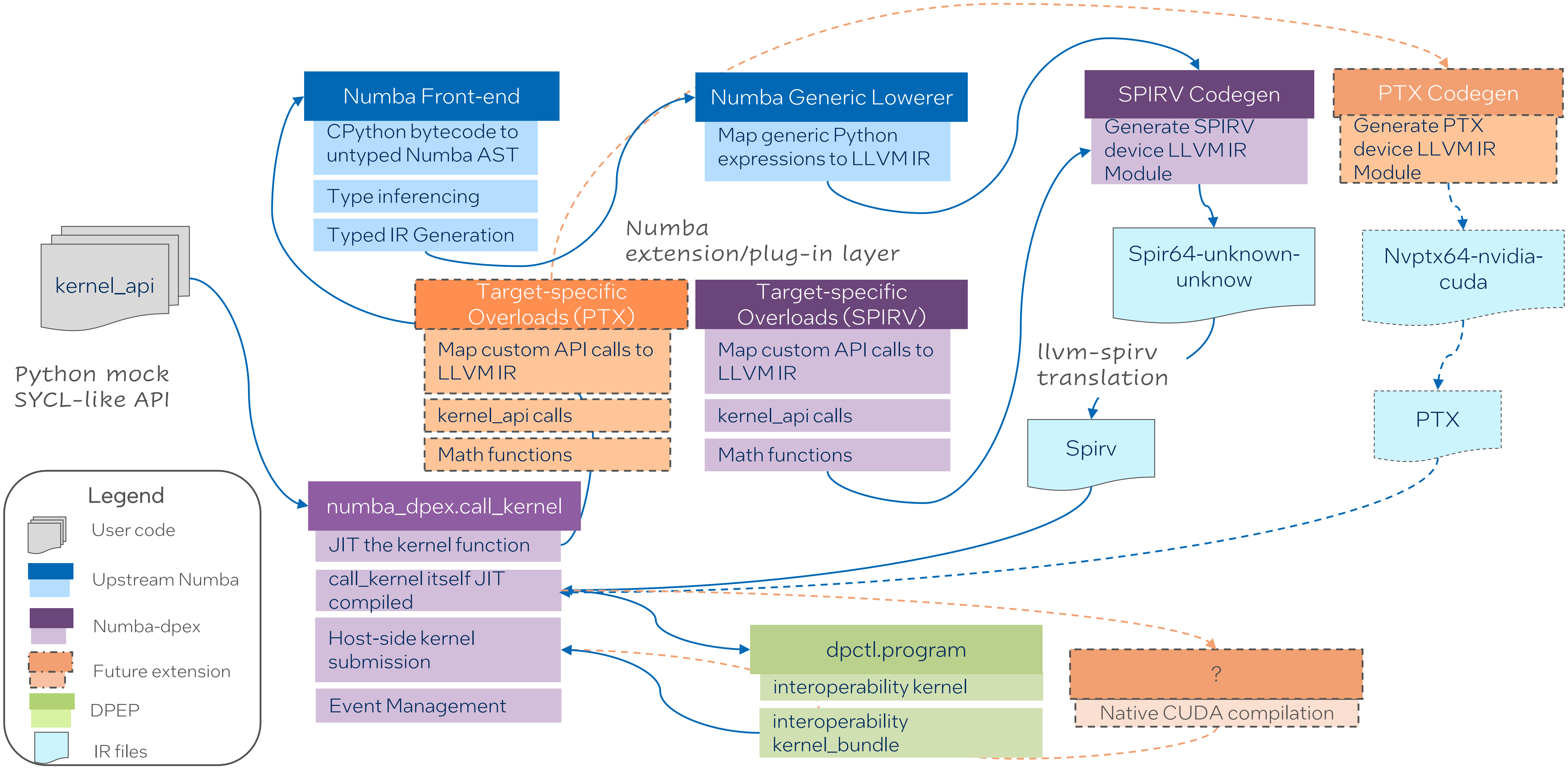
Note:

- Math functions in Python `math` and `dpnp` are supported in a kernel
- Most Python built-in operators are supported in a kernel

System Diagram



Codegen can be extended to other devices



Beyond Direct Kernel Programming

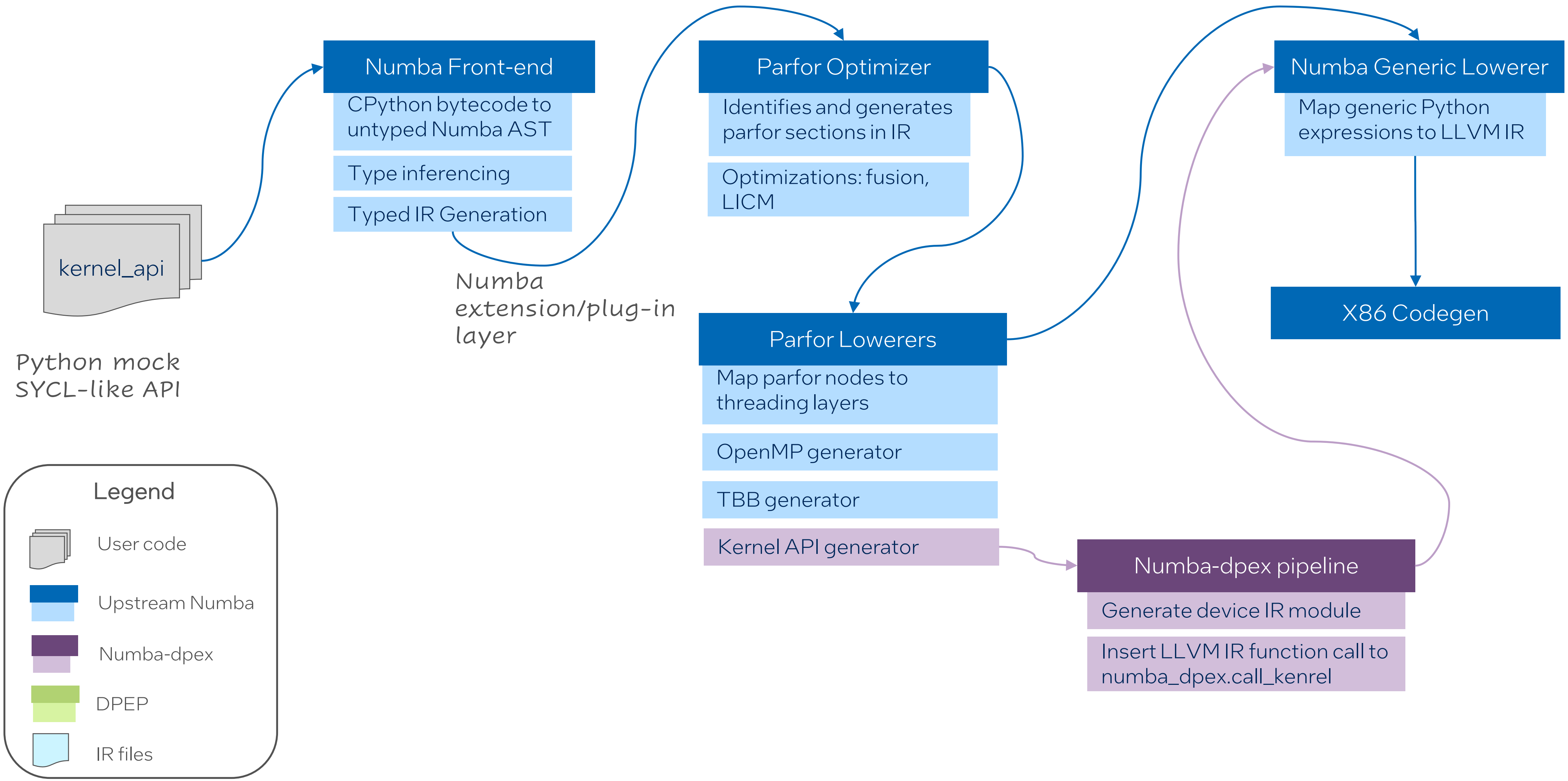
```
import dpnp
import numba_dpex as dpex

@dpex.dpjit
def pwd_fn(data, dist):
  for i in dpex.prange(data.shape[0]):
    for j in range(data.shape[0]):
      d = data.dtype.type(0.0)
      for k in range(data.shape[1]):
        d += (data[i, k] - data[j, k])**2
      dist[I, j] = dpnp.sqrt(d)

data = <elided>
dist = <elided>
pwd_fn(data, dist)
```

Numba-dpex experimental *do-all (prange)* API

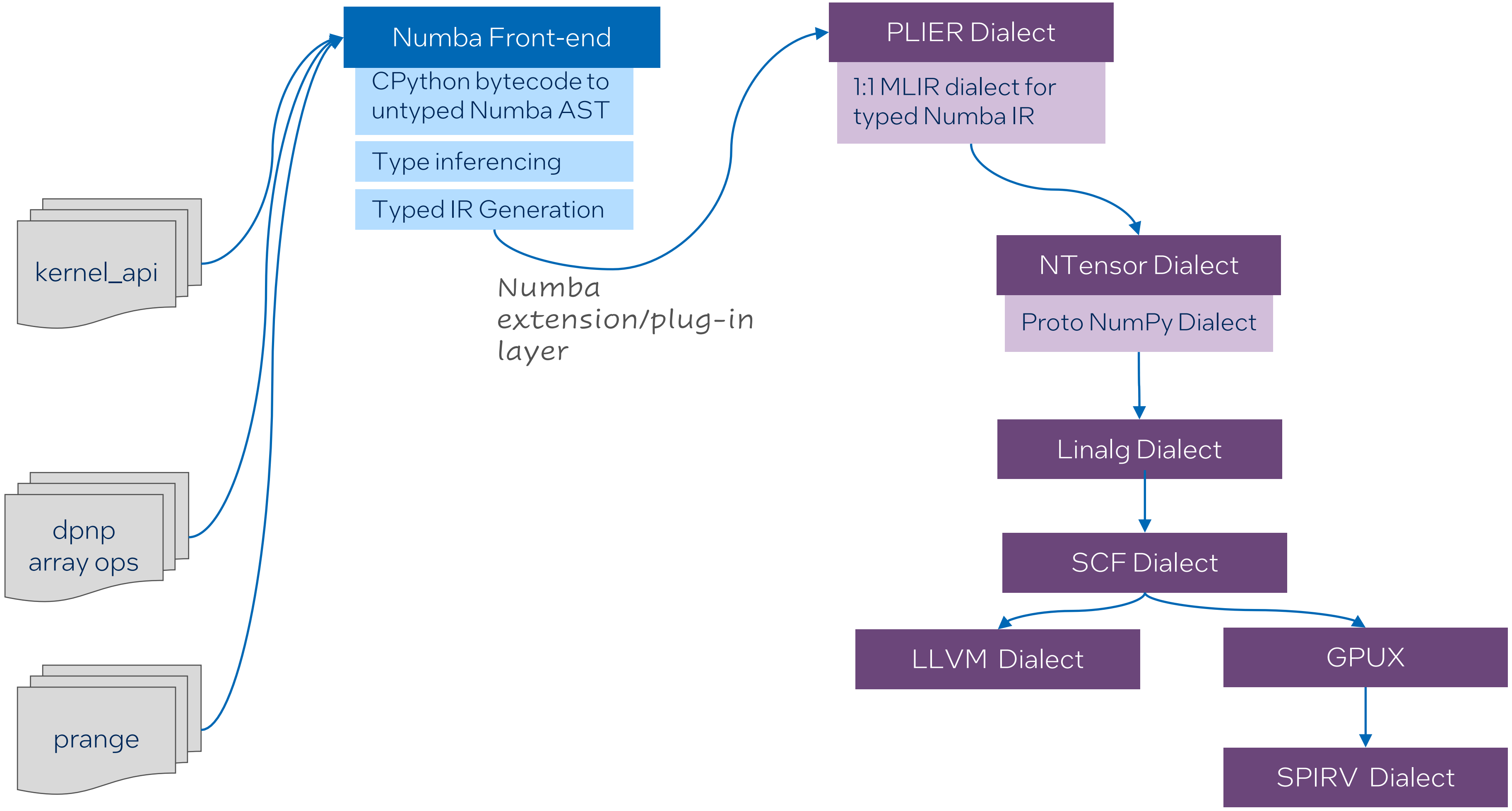
```
import dpnp
import numba_dpex as dpex

@dpex.dpjit
def pwd_fn(data):
  data_sqr = dpnp.sum(dpnp.square(data), axis=1)
  dist = dpnp.dot(data, data.T)
  dist *= -2
  dist = dist + data_sqr.reshape(data_sqr.size, 1)
  dist = dist + data_sqr
  return dpnp.sqrt(dist)


data = <elided>
dist = pwd_fn(data)
```

Proposed pure array-programming API

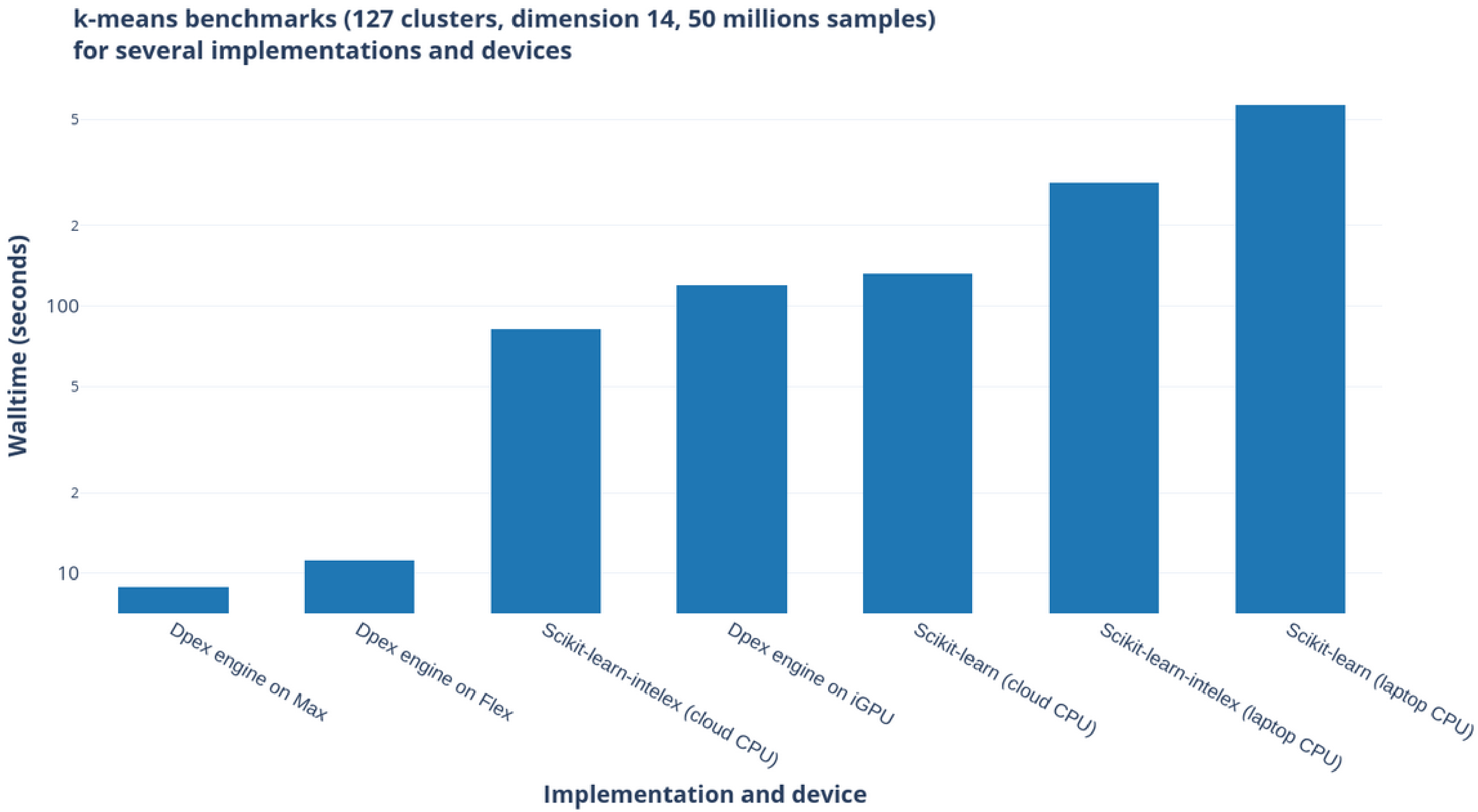
Par-for compilation



Exploration in building an MLIR-based backend



User story (INRIA/Scikit-learn community)



Source: https://team.inria.fr/soda/exploring-a-oneapi-based-gpu-powered-backend-for-scikit-learn/
https://github.com/soda-inria/sklearn-numba-dpex

Status as of 01/31/2024

- Support for OpenCL CPU, OpenCL GPU, Level Zero GPU devices
  - *Including all current generations of Intel GPU: Max, Iris Xe, Gen 9*
- GDB debugging support on both CPU and GPU devices
- Production grade kernel API
- Partial array programming compilation using dpnp tensors
- Proof of concept MLIR backend

Getting to the Code

GitHub repositories

- https://github.com/IntelPython/numba-dpex
- https://github.com/IntelPython/dpctl

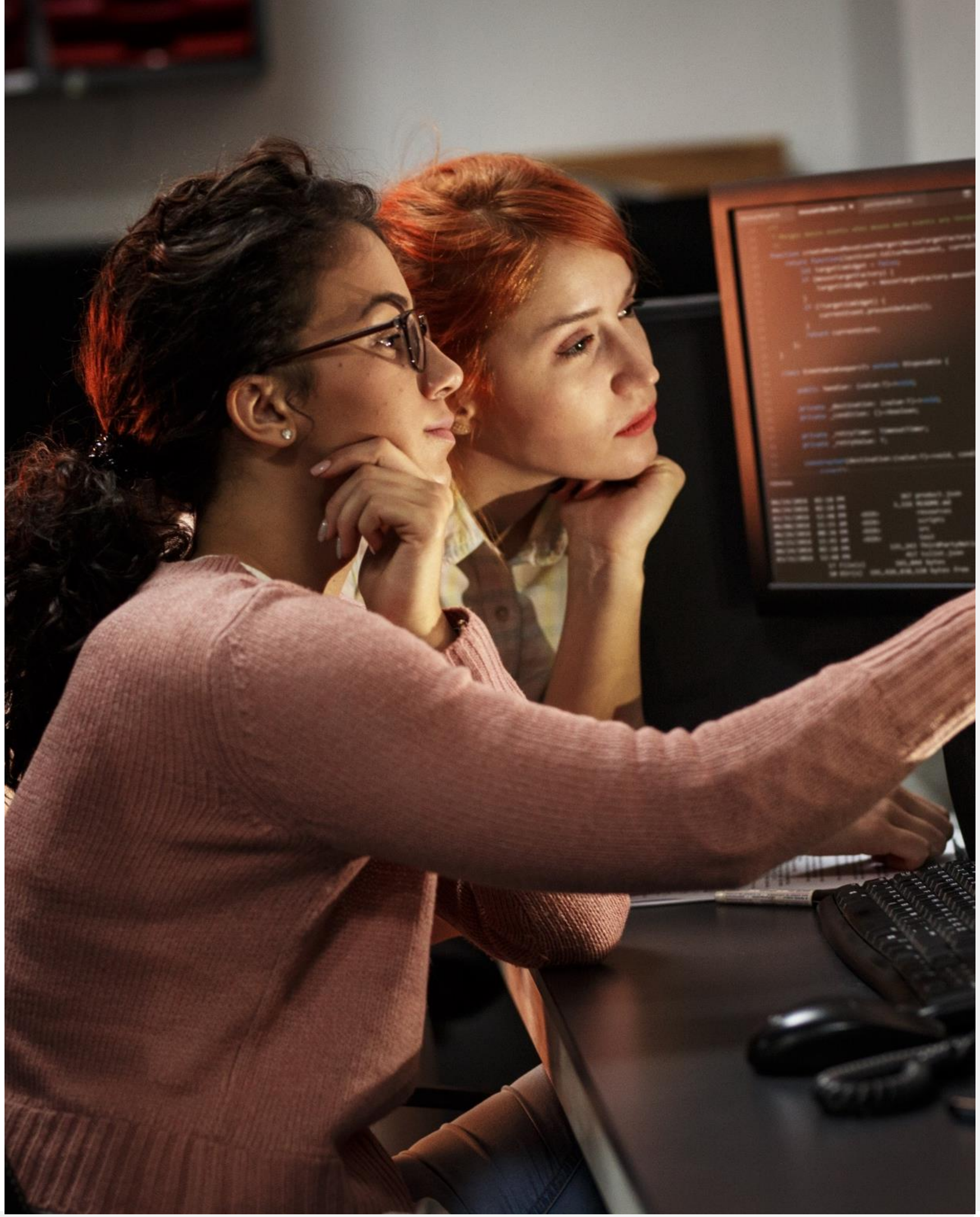
Installable using pip or conda

```
conda install numba-dpex -c intel -c conda-forge
```

```
Python -m pip install numba-dpex
```

Chat

```
https://app.gitter.im/#/room/#Data-Parallel-Python_community:gitter.im
```



Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

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

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

intel®

tensor

Data-parallel Python Array API standard reference implementation

tensor

What?

A SYCL-based data-parallel array library conforming to Python Array API standard

Array API standard (2022.12) conformance
(As of 01/31/2024)
Passed: 909
Failed: 0
Skipped: 86

Why?

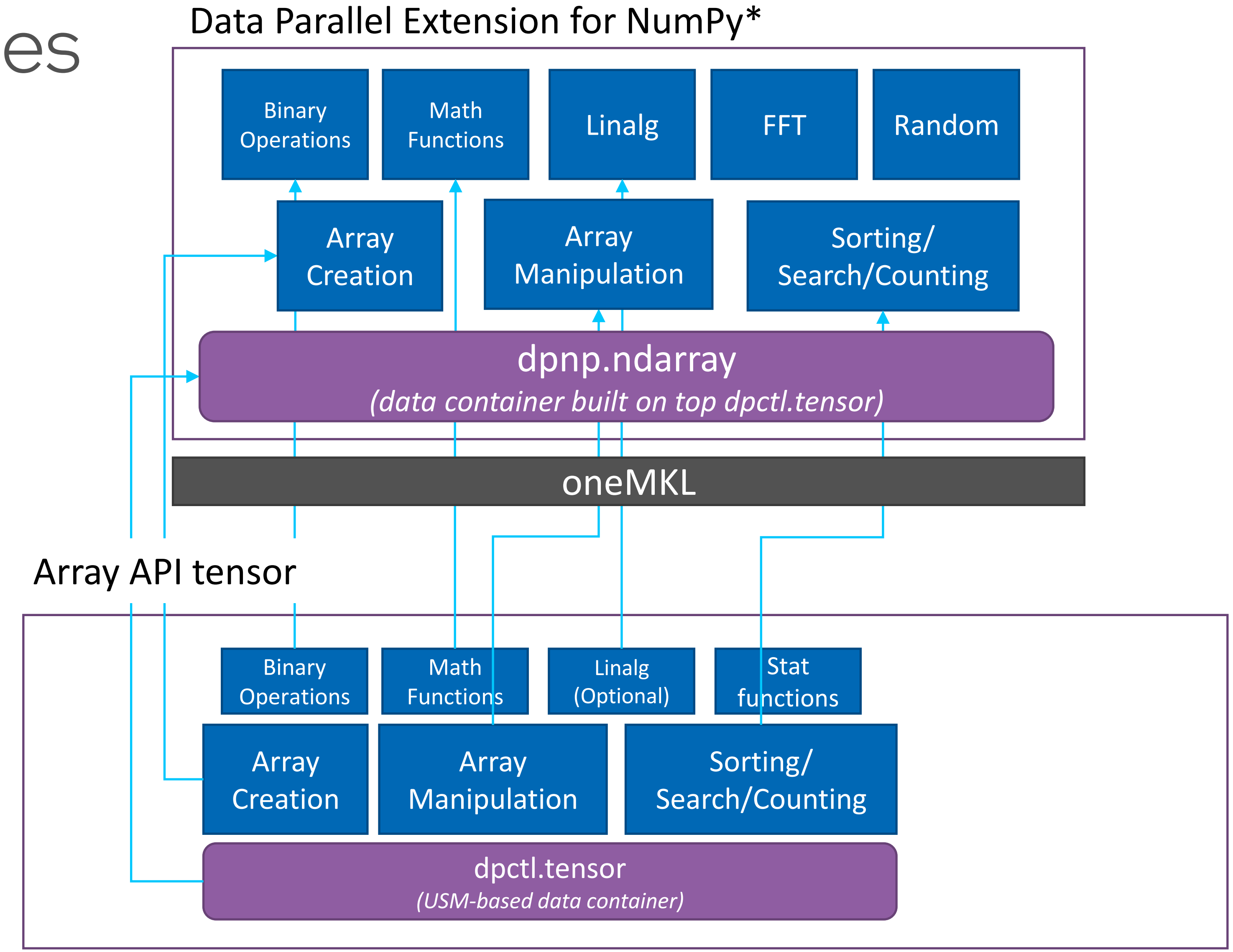
- Portable array data type for data on devices from heterogeneous compute platforms
- Set of standard operations to manipulate the array type

Benefits

Make any PyData package that supports Array API support heterogeneous computing by passing in a tensor array container

Building block for higher-level packages

Enables Array API-based Python packages to be portable across different heterogeneous platforms



dpnp : Data Parallel Extension for NumPy*

dpnp : Data Parallel Extension for NumPy*

What?

Drop-in replacement for NumPy to allow heterogeneous computation on supported SYCL devices

Why?

- NumPy is CPU-only and not parallel
- Large corpus of legacy code base, so the same API needs to be reimplemented

Benefits

- Minimal change to existing NumPy-based code base
- Support for CPU, Intel iGPU, and Intel dGPU

A quick demo

Original NumPy script

```
import numpy as np

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

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

Same, but running on the default SYCL device

```
import dpnp as np

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

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

Modified script – specify a device to run operations there!

```
import dpnp as np

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

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

API Coverage and Status as of 01/31/2024

Name	NumPy	DPNP	CuPy
Module-Level	397	233	299
Multi-Dimensional Array	56	36	47
Linear Algebra	20	15	16
Discrete Fourier Transform	18	18	18
Random Sampling	51	48	49
Total	542	350	429

Source: https://intelpython.github.io/dpnp/reference/comparison.html#summary