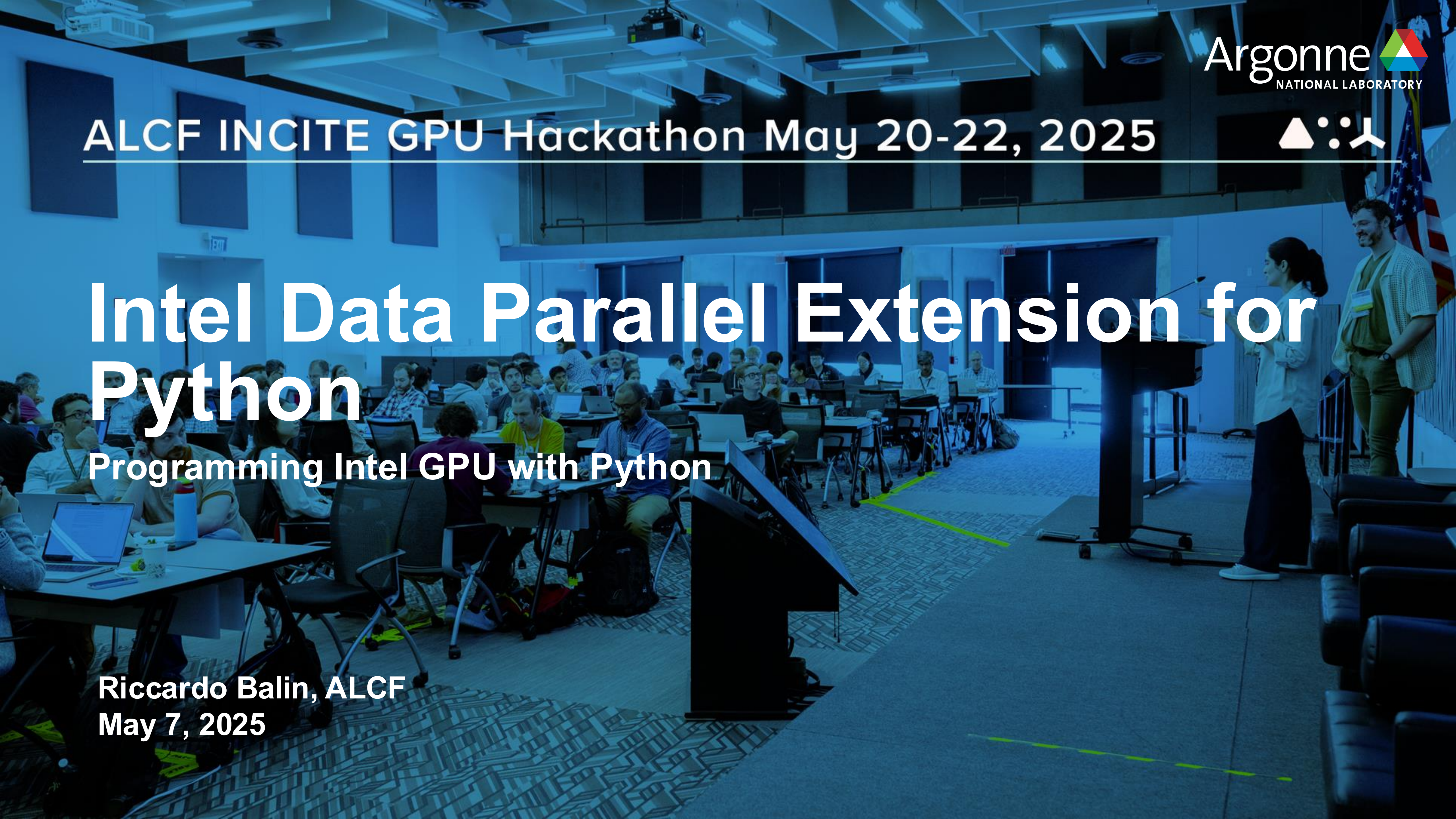
Argonne NATIONAL LABORATORY

ALCF INCITE GPU Hackathon May 20-22, 2025



Intel Data Parallel Extension for Python

Programming Intel GPU with Python

Riccardo Balin, ALCF
May 7, 2025

Data Parallel Extension for Python (DPEP)

- Intel's Python stack for programming on heterogeneous devices, including Aurora's CPUs and GPUs
- Composed of three packages:
  - dpnp – Data Parallel Extension for NumPy
  - dpctl – Data Parallel Control
  - numba-dpex – Data Parallel Extension for Numba
- Compute-follows-data programming model
  - Offload target is inferred from the input arguments (data) to the library call or kernel (compute)
  - No need to specify offload target directly for each function/kernel call

compute_follows_data.py

```
import dpctl.tensor as dpt

x_gpu = dpt.arange(100, device="gpu")
sqx_gpu = dpt.square(x_gpu)
print(sqx_gpu.device)
```

Output log

```
>>> python compute_follows_data.py
Device(level_zero:gpu:0)
```

Accessing DPEP Packages on Aurora

- dpnp and dpctl are included in base AI/ML frameworks conda environment

```
>>> module load frameworks
>>> conda list | grep -e dpnp -e dpctl
dpctl      0.18.3     py310ha998128_0    https://software.repos.intel.com/python/conda
dpnp       0.16.3     py310ha998128_0    https://software.repos.intel.com/python/conda
```

- Currently, accessing numba-dpex requires additional install steps

```
>>> module load frameworks
# clone base conda environment or create a new one
>>> conda install -y scikit-build numba==0.59* -c conda-forge
>>> pip install versioneer
>>> git clone https://github.com/IntelPython/numba-dpex.git
>>> cd numba-dpex
>>> CXX=$(which dpcpp) python setup.py develop
```

- For more details on cloning the base conda environment or creating a new environment, see our documentation page

dpnp: Data Parallel Extension for NumPy

- Implements a subset of NumPy API using DPC++
- Intended as drop-in replacement for NumPy, similarly to CuPy for CUDA devices

```
- import numpy as np # or import cupy as np
+ import dpnp as np
```

- Should be used to port NumPy and CuPy code to Aurora GPU
- Check comparison table for current API coverage relative to NumPy and CuPy
- Supports reduced precision for floating, complex, and integer numbers (e.g., fp16, int8, complex64)
- Leverages oneMKL

dpnp: Very Simple Example

dpnp_sum.py

```
import dpnp as np

x = np.asarray([1, 2, 3])
print("Array x allocated on the device:", x.device)

y = np.sum(x)
print("Result y is located on the device:", y.device)
```

Output log

```
>>> python dpnp_sum.py
Array x allocated on the device: Device(level_zero:gpu:0)
Result y is located on the device: Device(level_zero:gpu:0)
```

dpnp: Very Simple Example (cont.)

dpnp_sum.py

```
import dpnp as np

x = np.asarray([1, 2, 3])
print("Array x allocated on the device:", x.device)

y = np.sum(x)
print("Result
```

- The array created on the default SYCL device, i.e. PVC on Aurora
- SYCL queue and device objects are created and carried with the array
  - SYCL queue: `x.sycl_queue`
  - SYCL device: `x.device`

- The pre-compiled kernel for `np.sum()` is submitted to the queue of the input array `x`
- The output array is allocated on the same SYCL device as `x` and associated with the same queue

Output log

```
>>> python dpnp_sum.py
Array x allocated on the device: Device(level_zero:gpu:0)
Result y is located on the device: Device(level_zero:gpu:0)
```

dpnp: Important Details

- Array creation API take as arguments device, USM memory type and SYCL queue
  - —Allocate array on any GPU: `x = np.asarray([1, 2, 3], device=“gpu:1”)`
  - —Allocate array on USM: `x = np.asarray([1, 2, 3], usm_type=“shared”)`
- By default, only GPU are visible as SYCL devices
  - —On Aurora, `ONEAPI_DEVICE_SELECTOR=level_zero:gpu` is set by default
  - —To access CPU, set `ONEAPI_DEVICE_SELECTOR=“level_zero:gpu;opencl:cpu”`
- From version >0.15.0, all kernels run asynchronously with linear ordering (similar to CuPy)
  - —To time kernels, need to use `.sycl_queue.wait()`

dpnp_matmul.py

```
import dpnp as np
from time import perf_counter

x = np.random.randn(1000,1000)
tic = perf_counter()
y = np.matmul(x,x)
y.sycl_queue.wait()
print(f"Execution time: {perf_counter() - tic} sec")
```

dpctl: Data Parallel Control

- Library used to access devices supported by the DPC++ SYCL runtime
- Expose to Python features such as:
  - Device introspection
  - Execution queue creation
  - Memory allocation
  - Kernel creation and submission
- Also provides a tensor library implemented in C++ and SYCL called dpctl.tensor
  - Follows Python Array API standard (interoperability through DLPack)
  - Provides API for array creation, manipulation and linear algebra functions
  - Supports reduced precision for floating, complex, and integer numbers (e.g., fp16, int8, complex64)
- dpctl is a dependency of dpnp

dpctl: Device Introspection

dpctl_device_introspection.py

```
import dpctl

num_devices = dpctl.get_num_devices(device_type="gpu")
print(f"Found {num_devices} GPU devices")

device_list = dpctl.get_devices(device_type="gpu")
for device in device_list:
    print(f"\t{device}")

print("\nFound CPU devices: ", dpctl.has_cpu_devices())
```

dpctl: Device Introspection (cont.)

Output log

```
>>> python dpctl_device_introspection.py
Found 12 GPU devices
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8da03430>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8da034f0>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8da035b0>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8da03530>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8da03f70>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8d005770>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8b5a3ab0>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8d2467b0>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8da24ef0>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8da03fb0>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8da240b0>
    <dpctl.SyclDevice [backend_type.level_zero, device_type.gpu, Intel(R) Data Center GPU Max 1550] at 0x154e8da24130>

Found CPU devices: False
```

- Found 12 GPU on Aurora node
    - ML frameworks module sets `ZE_FLAT_DEVICE_HIERARCHY=FLAT`
    - Each of 12 tiles on 6 PVC are visible as a distinct device
- CPU not visible due to default `ONEAPI_DEVICE_SELECTOR=level_zero:gpu`

Argonne NATIONAL LABORATORY

dpctl: Device Selection and Queue Creation

dpctl_device_selection.py

```
import dpnp as dp
import dpctl

devices = dpctl.get_devices()
num_devices = len(devices)
print(f'Found {num_devices} GPU devices’)

_queues = [None] * num_devices
for device_id in range(num_devices):
    _queues[device_id] = dpctl.SyclQueue(devices[device_id])

def func(device_id=0):
    arr = dp.ndarray([0,1,2],sycl_queue=_queues[device_id])
    return arr

for device_id in range(num_devices):
    print(func(device_id).device)
```

dpctl: Device Selection and Queue Creation (cont.)

dpctl_device_selection.py

```
import dpnp as dp
import dpctl

devices = dpctl.get_devices()
num_devices = len(devices)
print(f'Found {num_devices} GPU devices’)

_queues = [None] * num_devices
for device_id in range(num_devices):
    _queues[device_id] = dpctl.SyclQueue(devices[device_id])

def func(device_id=0):
    arr = dp.ndarray([0,1,2],sycl_queue=_queues[device_id])
    return arr

for device_id in range(num_devices):
    print(func(device_id).device)
```

Create a separate SYCL queue to access each GPU

Allocate array on each GPU with specific SYCL queue

dpctl: Device Selection and Queue Creation (cont.)

Output log

```
Found 12 GPU devices
Device(level_zero:gpu:0)
Device(level_zero:gpu:1)
Device(level_zero:gpu:2)
Device(level_zero:gpu:3)
Device(level_zero:gpu:4)
Device(level_zero:gpu:5)
Device(level_zero:gpu:6)
Device(level_zero:gpu:7)
Device(level_zero:gpu:8)
Device(level_zero:gpu:9)
Device(level_zero:gpu:10)
Device(level_zero:gpu:11)
```

Created an array on each of 12 GPU devices

numba-dpex: Data Parallel Extension for Numba

- Generate performant, parallel code on Aurora’s CPU and GPU with Numba JIT compilation
- Provides a kernel programming API (kapi) in Python integrated with LLVM-based code generator
- Integrates with numpy, dpnp and dpctl and is based on compute-follows-data programming model
  - Kernels execute on CPU or GPU depending on where input data is allocated
- Offers range and nd-range kernels for increased complexity of kernels
- Offers device callable functions to be invoked from within a kernel or other device function
- Functions from the math and dpnp packages can be called within kernels
  - See documentation for mode details on supported functions and data types

numba-dpex: Range Kernels

- Implements basic parallel-for calculation
- Ideally suited for embarrassingly parallel kernels, e.g., elementwise computations

dpex_sum.py

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

# Data parallel kernel implementation of vector sum
@dpex.kernel
def vecadd(item: kapi.Item, a, b, c):
    i = item.get_id(0)
    c[i] = a[i] + b[i]

N = 1024
a = dpnp.ones(N)
b = dpnp.ones_like(a)
c = dpnp.zeros_like(a)
dpex.call_kernel(vecadd, dpex.Range(N), a, b, c)
```

numba-dpex: Range Kernels (cont.)

dpex_sum.py

```
import dpnp
import numba_dpex as dpex
from numba_dpe

@dpex.kernel
def vecadd(item: kapi.Item, a, b, c):
    i = item.get_id(0)
    c[i] = a[i] + b[i]

N = 1024
a = dpnp.ones(N)
b = dpnp.ones_like(a)
c = dpnp.zeros_like(a)
dpex.call_kernel(vecadd, dpex.Range(N), a, b, c)
```

Decorate function as a dpex kernel

Get the work item
- numba-dpex follows SPMD programming model
- Each work item runs the function for a subset of the elements of the input arrays

Define the set of work items

Interoperability through DLPack

- dpnp and dpctl provide interoperability with other Python libraries following Array API standards
- Zero-copy data access for CPU/GPU allocated arrays with NumPy (CPU only) and PyTorch, **not** TensorFlow
- For interoperability between ML frameworks and numba-dpex, first create dpnp or dpctl view of arrays

dlpack_example.py

```
import dpnp as dp
import torch
import intel_extension_for_pytorch as ipex

t_ary = torch.arange(4).to('xpu') # array [0, 1, 2, 3] on GPU
dp_ary = dp.from_dlpack(t_ary)
t_ary[0] = -2.0 # modify the PyTorch array
print(f'Original PyTorch array: {t_ary}’)
print(f'dpnp view of PyTorch array: {dp_ary} on device {dp_ary.device}\n')
```

Output log

```
>>> python dlpack_example.py
Original PyTorch array: tensor([-2, 1, 2, 3], device='xpu:0’)
dpnp view of PyTorch array: [-2 1 2 3] on device Device(level_zero:gpu:0)
```

Thank you!

Please send any questions to support@alcf.anl.gov or contact me on Slack