Accelerating AI Training and Inference for Science on Aurora

Frameworks, Tools, and Best Practices

Riccardo Balin, Filippo Simini, Khalid Hossain, Sam Foreman, Sid Raskar

Argonne Leadership Computing Facility

May 28th, 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 to the library call or kernel
  - No need to specify offload target directly for each function/kernel call
- Interoperability with AI frameworks supported through DLPack
  - Zero-copy data access for CPU/GPU allocated arrays with NumPy (CPU only) and PyTorch, **not** TensorFlow
- Please refer to our documentation and GettingStarted repo for additional details and examples
  - Documentation: https://docs.alcf.anl.gov/aurora/data-science/python/
  - GettingStarted examples: https://github.com/argonne-lcf/GettingStarted/tree/developer_session_2025_05/DataScience/developer_session_2025_05

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 (see our documentation for details)

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
- Should be used to port NumPy and CuPy code to Aurora GPU

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

- 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 for performant linear algebra on PVC

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.)

- 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`

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 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)
```

dpctl: Data Parallel Control

- Library used to access devices supported by the DPC++ 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 Selection and Queue Creation

dpctl_device_selection.py

```
import dpnp as dp
import dpctl

devices = dpctl.get_devices(device_type="gpu")
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(device_type="gpu")
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)
```

Get the available GPU devices on the node

Create a separate SYCL queue to access each GPU

Allocate array on each GPU with specific SYCL queue

dpnp and dpctl: Further Details

- Array creation API take as arguments the device, USM memory type and SYCL queue
  - —Allocate array on GPU 1: `x = dpnp.asarray([1, 2, 3], device="gpu:1")`
  - —Allocate array on USM: `x = dpnp.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"`
- ML frameworks module sets `ZE_FLAT_DEVICE_HIERARCHY=FLAT`
  - Each tile on 6 PVC is visible as a distinct device for a total of 12 devices
- From dpnp >0.15.0 and dpctl >0.17.0, all kernels run asynchronously with linear ordering (similar to CuPy)
  - —To time kernels, need to use `.sycl_queue.wait()` on output array or function call

numba-dpex: Data Parallel Extension for Numba

- Generate performant, parallel code on Aurora's CPU and GPU with Numba JIT compilation
- Provides a Python kernel programming API (kapi) 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 functions
- Functions from the math and dpnp packages can be called within kernels
  - See numba-dpex 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

- 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 paralle                                    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)
```

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