numba-dppy

Release 0.13.1

Intel

Apr 09, 2021

CORE FEATURES

numba-dppy is a standalone extension to the Numba JIT compiler that adds SYCL programming capabilities to Numba. numba-dppy uses dpCtl to support SYCL features and currently Intel's DPC++ is the only SYCL runtime supported by numba-dppy.

There are two ways to program SYCL devices using numba-dppy:

- An explicit kernel programming mode.

```
import numpy as np
import numba_dppy, numba_dppy as dppy
import dpctl

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

a = np.array(np.random.random(20), dtype=np.float32)
b = np.array(np.random.random(20), dtype=np.float32)
c = np.ones_like(a)

with dpctl.device_context("opencl:gpu"):
    sum[20, dppy.DEFAULT_LOCAL_SIZE](a, b, c)
```

- An automatic offload mode for NumPy data-parallel expressions and Numba parallel loops.

```
from numba import njit
import numpy as np
import dpctl

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

global_size = 64
local_size = 32
N = global_size * local_size
a = np.ones(N, dtype=np.float32)
b = np.ones(N, dtype=np.float32)
with dpctl.device_context("opencl:gpu:0"):
    c = f1(a, b)
```

CHAPTER

ONE

CODE-GENERATION BASED ON A DEVICE

In `numba-dppy`, kernels are written in a device-agnostic fashion making it easy to write portable code. A kernel is compiled for the device on which the kernel is enqueued to be executed. The device is specified using a `dpctl.device_context` context manager. In the following example, two versions of the `sum` kernel are compiled, one for a GPU and another for a CPU based on which context the function was invoked. Currently, `numba-dppy` supports OpenCL CPU and GPU devices and Level Zero GPU devices. In future, compilation support may be extended to other type of SYCL devices that are supported by DPC++'s runtime.

```
import numpy as np
import numba_dppy, numba_dppy as dppy
import dpctl

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

a = np.array(np.random.random(20), dtype=np.float32)
b = np.array(np.random.random(20), dtype=np.float32)
c = np.ones_like(a)

with dpctl.device_context("level_zero:gpu"):
    sum[20, dppy.DEFAULT_LOCAL_SIZE](a, b, c)

with dpctl.device_context("opencl:cpu"):
    sum[20, dppy.DEFAULT_LOCAL_SIZE](a, b, c)
```

CHAPTER

TWO

AUTOMATIC OFFLOAD OF NUMPY EXPRESSIONS

A key distinction between `numba-dppy` and other the GPU backends in Numba is the ability to automatically offload specific data-parallel sections of a Numba `jit` function.

Todo: Details and examples to be added.

2.1 Controllable Fallback

By default, if a section of code cannot be offloaded to the GPU, it is automatically executed on the CPU and warning is printed. This behavior is only applicable to `jit` functions, auto-offloading of NumPy calls, array expressions and `prange` loops. To disable this functionality and force code running on GPU set the environment variable `NUMBA_DPPY_FALLBACK_OPTION` to false (e.g. `export NUMBA_DPPY_FALLBACK_OPTION=0`). In this case the code is not automatically offloaded to the CPU and errors occur if any.

2.2 Offload Diagnostics

Setting the debug environment variable `NUMBA_DPPY_OFFLOAD_DIAGNOSTICS` (e.g. `export NUMBA_DPPY_OFFLOAD_DIAGNOSTICS=1`) provides emission of the parallel and offload diagnostics information based on produced parallel transforms. The level of detail depends on the integer value between 1 and 4 that is set to the environment variable (higher is more detailed). In the "Auto-offloading" section there is the information on which device (device name) this parfor or kernel was offloaded.

CHAPTER

THREE

GETTING STARTED

3.1 Installation

`numba-dppy` depends on following components:

- numba 0.52.* (Intel Python Numba)
- dpctl 0.6.* (Intel Python dpCtl)
- dpnp >=0.5.1 (optional, Intel Python DPNP)
- llvm-spirv (SPIRV generation from LLVM IR)
- llvmdev (LLVM IR generation)
- spirv-tools
- cython (for building)
- scipy (for testing)
- pytest (for testing)

It is recommended to use conda packages from Intel Distribution for Python channel or anaconda.org/intel channel. Intel Distribution for Python is available from Intel oneAPI.

Create conda environment:

```
export ONEAPI_ROOT=/opt/intel/oneapi
conda create -n numba-dppy-env numba-dppy dpnp -c ${ONEAPI_ROOT}/conda_channel
```

3.2 Build and Install Conda Package

Create and activate conda build environment:

```
conda create -n build-env conda-build
conda activate build-env
```

Set environment variable `ONEAPI_ROOT` and build conda package:

```
export ONEAPI_ROOT=/opt/intel/oneapi
conda build conda-recipe -c ${ONEAPI_ROOT}/conda_channel
```

Install conda package:

```
conda install numba-dppy
```

3.3 Build and Install with setuptools

`setup.py` requires environment variable `ONEAPI_ROOT` and following packages installed in conda environment:

```
export ONEAPI_ROOT=/opt/intel/oneapi
conda create -n numba-dppy-env -c ${ONEAPI_ROOT}/conda_channel python=3.7 dpctl dpnp␣
↪numba spirv-tools llvm-spirv llvmdev cython pytest
conda activate numba-dppy-env
```

Activate DPC++ compiler:

```
source ${ONEAPI_ROOT}/compiler/latest/env/vars.sh
```

For installing:

```
python setup.py install
```

For development:

```
python setup.py develop
```

3.4 Testing

See folder `numba_dppy/tests`.

To run the tests:

```
python -m pytest --pyargs numba_dppy.tests
```

3.5 Examples

See folder `numba_dppy/examples`.

To run the examples:

```
python numba_dppy/examples/sum.py
```

3.6 Limitations

Using `numba-dppy` requires Intel Python Numba as that version of Numba has patches needed to recognize a `dpctl.device_context` scope and trigger code-generation for a SYCL device. Work in underway to upstream all patches, so that in future `numba-dppy` can work with upstream Numba.

CHAPTER

FOUR

PROGRAMMING SYCL KERNELS USING `NUMBA_DPPY.KERNEL`

4.1 Writing SYCL Kernels

4.1.1 Introduction

`numba-dppy` offers a way of programming SYCL supporting devices using Python. Similar to SYCL's C++ programming model for heterogeneous computing, `numba-dppy` offers Python abstractions for expressing data-parallelism using a hierarchical syntax. Note that not all SYCL concepts are currently supported in `numba-dppy`, and some of the concepts may not be a good fit for Python.

The explicit kernel programming mode of `numba-dppy` bears lots of similarities with Numba's other GPU backends:`numba.cuda` and `numba.roc`. Readers who are familiar with either of the existing backends of Numba, or in general with OpenCL, CUDA, or SYCL programming should find writing kernels in `numba-dppy` extremely intuitive. Irrespective of the reader's level of familiarity with GPU programming frameworks, this documentation should serves as a guide for using the current features available in `numba-dppy`.

4.1.2 Kernel declaration

A kernel function is a device function that is meant to be called from host code, where a device can be any SYCL supported device such as a GPU, CPU, or an FPGA. The present focus of development of `numba-dppy` is mainly on Intel's GPU hardware. The main characteristics of a kernel function are:

- kernels cannot explicitly return a value; all result data must be written to an array passed to the function (if computing a scalar, you will probably pass a one-element array)
- kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block (note that while a kernel is compiled once, it can be called multiple times with different block sizes or grid sizes).

Example

```
#! /usr/bin/env python
# Copyright 2021 Intel Corporation
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
```

(continues on next page)

(continued from previous page)

```
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from __future__ import print_function
from timeit import default_timer as time

import sys
import numpy as np
import numpy.testing as testing
import numba_dppy, numba_dppy as dppy
import dpctl


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


def driver(a, b, c, global_size):
    print("A : ", a)
    print("B : ", b)
    data_parallel_sum[global_size, dppy.DEFAULT_LOCAL_SIZE](a, b, c)
    print("A + B = ")
    print("C ", c)
    testing.assert_equal(c, a + b)


def main():
    global_size = 10
    N = global_size
    print("N", N)

    a = np.array(np.random.random(N), dtype=np.float32)
    b = np.array(np.random.random(N), dtype=np.float32)
    c = np.ones_like(a)

    if dpctl.has_gpu_queues():
        print("\nScheduling on OpenCL GPU\n")
        with dpctl.device_context("opencl:gpu") as gpu_queue:
            driver(a, b, c, global_size)
    else:
        print("\nSkip scheduling on OpenCL GPU\n")
    if dpctl.has_gpu_queues(dpctl.backend_type.level_zero):
        print("\nScheduling on Level Zero GPU\n")
        with dpctl.device_context("level0:gpu") as gpu_queue:
            driver(a, b, c, global_size)
    else:
        print("\nSkip scheduling on Level Zero GPU\n")
    if dpctl.has_cpu_queues():
        print("\nScheduling on OpenCL CPU\n")
        with dpctl.device_context("opencl:cpu") as cpu_queue:
            driver(a, b, c, global_size)
    else:
```

(continues on next page)

(continued from previous page)

```
        print("\nSkip scheduling on OpenCL CPU\n")
    print("Done...")


if __name__ == "__main__":
    main()
```

4.1.3 Kernel invocation

A kernel is typically launched in the following way:

```
def driver(a, b, c, global_size):
    print("A : ", a)
    print("B : ", b)
    data_parallel_sum[global_size, dppy.DEFAULT_LOCAL_SIZE](a, b, c)
    print("A + B = ")
    print("C ", c)
    testing.assert_equal(c, a + b)
```

4.1.4 Indexing functions

Currently, `numba-dppy` supports the following indexing functions that have the same semantics as OpenCL.

- `numba_dppy.get_local_id`
- `numba_dppy.get_local_size`
- `numba_dppy.get_group_id`
- `numba_dppy.get_num_groups`

4.2 Memory Management

`numba-dppy` uses DPC++'s USM shared memory allocator (`memory_alloc`) to enable host to device and *vice versa* data transfer. By using USM shared memory allocator, `numba-dppy` allows seamless interoperability between `numba-dppy` and other SYCL-based Python extensions and across multiple kernels written using `numba_dppy.kernel` decorator.

`numba-dppy` uses the USM memory manager provided by `dpctl` and supports the **SYCL USM Array Interface** protocol to enable zero-copy data exchange across USM memory-backed Python objects.

Note: USM pointers make sense within a SYCL context and can be of four allocation types: `host`, `device`, `shared`, or `unknown`. Host applications, including CPython interpreter, can work with USM pointers of type `host` and `shared` as if they were ordinary host pointers. Accessing `device` USM pointers by host applications is not allowed.

4.2.1 SYCL USM Array Interface

A SYCL library may allocate USM memory for the result that needs to be passed to Python. A native Python extension that makes use of such a library may expose this memory as an instance of Python class that will implement memory management logic (ensures that memory is freed when the instance is no longer needed). The need to manage memory arises whenever a library uses a custom allocator. For example, `daal4py` uses Python capsule to ensure that a native library-allocated memory is freed using the appropriate deallocator.

To enable native extensions to pass the memory allocated by a native SYCL library to Numba, or another SYCL-aware Python extension without making a copy, the class must provide `__sycl_usm_array_interface__` attribute which returns a Python dictionary with the following fields:

- `shape`: tuple of `int`
- `typestr`: `string`
- `typedescr`: a list of tuples
- `data`: (`int`, `bool`)
- `strides`: tuple of `int`
- `offset`: `int`
- `version`: `int`
- `syclobj`: `dpctl.SyclQueue` or `dpctl.SyclContext` object

The dictionary keys align with those of `numpy.ndarray.__array_interface__` and `__cuda_array_interface__`. For host accessible USM pointers, the object may also implement CPython PEP-3118 compliant buffer interface which will be used if a `data` key is not present in the dictionary. Use of a buffer interface extends the interoperability to other Python objects, such as `bytes`, `bytearray`, `array.array`, or `memoryview`. The type of the USM pointer stored in the object can be queried using methods of the `dpctl`.

4.2.2 Device-only memory and explicit data transfer

At the moment, there is no mechanism for the explicit transfer of arrays to the device and back. Please use usm arrays.

4.2.3 Local memory

In SYCL's memory model, local memory is a contiguous region of memory allocated per work group and is visible to all the work items in that group. Local memory is device-only and cannot be accessed from the host. From the perspective offers the device, the local memory is exposed as a contiguous array of a specific types. The maximum available local memory is hardware-specific. The SYCL local memory concept is analogous to CUDA's shared memory concept.

`numba-dppy` provides a special function `dppy.local.array` to allocate local memory for a kernel.

```
def local_memory():
    blocksize = 10

    # @dppy.kernel("void(float32[::1])")
    @dppy.kernel
    def reverse_array(A):
        lm = dppy.local.array(shape=10, dtype=float32)
        i = dppy.get_global_id(0)

        # preload
```

(continues on next page)

(continued from previous page)

```
        lm[i] = A[i]
        # barrier local or global will both work as we only have one work group
        dppy.barrier(dppy.CLK_LOCAL_MEM_FENCE)  # local mem fence
        # write
        A[i] += lm[blocksize - 1 - i]

    arr = np.arange(blocksize).astype(np.float32)
    print(arr)

    with dpctl.device_context("opencl:gpu") as gpu_queue:
        reverse_array[blocksize, dppy.DEFAULT_LOCAL_SIZE](arr)

    # there arr should be orig[::-1] + orig, i.e. [9, 9, 9, ...]
    print(arr)
```

Note: To go convert from `numba.cuda` to `numba-dppy`, replace `numba.cuda.shared.array` with `numba_dppy.local.array(shape=blocksize, dtype=float32)`.

Todo: Add details about current limitations for local memory allocation in `numba-dppy`.

4.2.4 Private and Constant memory

`numba-dppy` does not yet support SYCL private and constant memory.

4.3 Synchronization Functions

Currently, `numba-dppy` only supports some of the SYCL synchronization operations. For synchronization of all threads in the same thread block, `numba-dppy` provides a helper function called `numba_dppy.barrier()`. This function implements the same pattern as barriers in traditional multi-threaded programming: invoking the function forces a thread to wait until all threads in the block reach the barrier, at which point it returns control to all its callers.

`numba_dppy.barrier()` supports two memory fence options:

- `numba_dppy.CLK_GLOBAL_MEM_FENCE`: The barrier function will queue a memory fence to ensure correct ordering of memory operations to global memory. Using the option can be useful when work-items, for example, write to buffer or image objects and then want to read the updated data. Passing no arguments to `numba_dppy.barrier()` is equivalent to setting the global memory fence option. For example,

```
def no_arg_barrier_support():
    # @dppy.kernel("void(float32[::1])")
    @dppy.kernel
    def twice(A):
        i = dppy.get_global_id(0)
        d = A[i]
        # no argument defaults to global mem fence
        dppy.barrier()
        A[i] = d * 2

    N = 10
```

(continues on next page)

(continued from previous page)

```
    arr = np.arange(N).astype(np.float32)
    print(arr)

    with dpctl.device_context("opencl:gpu") as gpu_queue:
        twice[N, dppy.DEFAULT_LOCAL_SIZE](arr)

    # there arr should be original arr * 2, i.e. [0, 2, 4, 6, ...]
    print(arr)
```

- `numba_dppy.CLK_LOCAL_MEM_FENCE`: The barrier function will either flush any variables stored in local memory or queue a memory fence to ensure correct ordering of memory operations to local memory. For example,

```
def local_memory():
    blocksize = 10

    # @dppy.kernel("void(float32[::1])")
    @dppy.kernel
    def reverse_array(A):
        lm = dppy.local.array(shape=10, dtype=float32)
        i = dppy.get_global_id(0)

        # preload
        lm[i] = A[i]
        # barrier local or global will both work as we only have one work group
        dppy.barrier(dppy.CLK_LOCAL_MEM_FENCE)  # local mem fence
        # write
        A[i] += lm[blocksize - 1 - i]

    arr = np.arange(blocksize).astype(np.float32)
    print(arr)

    with dpctl.device_context("opencl:gpu") as gpu_queue:
        reverse_array[blocksize, dppy.DEFAULT_LOCAL_SIZE](arr)

    # there arr should be orig[::-1] + orig, i.e. [9, 9, 9, ...]
    print(arr)
```

Note: The `numba_dppy.barrier()` function is semantically equivalent to `numba.cuda.syncthreads`.

4.4 Writing Device Functions

OpenCL and SYCL do not directly have a notion for device-only functions, *i.e.* functions that can be only invoked from a kernel and not from a host function. However, `numba-dppy` provides a special decorator `numba_dppy.func` specifically to implement device functions.

```
@dppy.func
def a_device_function(a):
    return a + 1
```

To use a device function from an another device function:

```
@dppy.func
def an_another_device_function(a):
    return a_device_function(a)
```

To use a device function from a kernel function `numba_dppy.kernel`:

```
@dppy.kernel
def a_kernel_function(a, b):
    i = dppy.get_global_id(0)
    b[i] = an_another_device_function(a[i])
```

Unlike a kernel function, a device function can return a value like normal functions.

Todo: Specific capabilities and limitations for device functions need to be added.

4.5 Supported Atomic Operations

`numba-dppy` supports some of the atomic operations supported in DPC++. Those that are presently implemented are as follows:

4.5.1 Example

Here's an example of how to use atomics add in DPPY:

```
def main():
    @dppy.kernel
    def atomic_add(a):
        dppy.atomic.add(a, 0, 1)

    global_size = 100
    a = np.array([0])

    with dpctl.device_context("opencl:gpu") as gpu_queue:
        atomic_add[global_size, dppy.DEFAULT_LOCAL_SIZE](a)
        # Expected 100, because global_size = 100
        print(a)
```

Note: The `numba_dppy.atomic.add` function is analogous to The `numba.cuda.atomic.add` provided by the `numba.cuda` backend.

4.5.2 Full examples

- numba_dppy/examples/atomic_op.py

4.6 Reduction on SYCL-supported Devices

numba-dppy does not yet provide any specific decorator to implement reduction kernels. However, a kernel reduction can be written explicitly. This section provides two approaches for writing a kernel reduction using numba_dppy.kernel.

4.6.1 Examples

Example 1

This example demonstrates a summation reduction on a one-dimensional array.

Full example can be found at numba_dppy/examples/sum_reduction.py.

In this example, to reduce the array we invoke the kernel multiple times.

```
@dppy.kernel
def sum_reduction_kernel(A, R, stride):
    i = dppy.get_global_id(0)
    # sum two element
    R[i] = A[i] + A[i + stride]
    # store the sum to be used in nex iteration
    A[i] = R[i]
```

```
def sum_reduce(A):
    """Size of A should be power of two."""
    total = len(A)
    # max size will require half the size of A to store sum
    R = np.array(np.random.random(math.ceil(total / 2)), dtype=A.dtype)

    context = get_context()
    with dpctl.device_context(context):
        while total > 1:
            global_size = total // 2
            sum_reduction_kernel[global_size, dppy.DEFAULT_LOCAL_SIZE](
                A, R, global_size
            )
            total = total // 2

    return R[0]
```

Example 2

Full example can be found at `numba_dppy/examples/sum_reduction_recursive_ocl.py`.

```
@dppy.kernel
def sum_reduction_kernel(A, input_size, partial_sums):
    local_id = dppy.get_local_id(0)
    global_id = dppy.get_global_id(0)
    group_size = dppy.get_local_size(0)
    group_id = dppy.get_group_id(0)

    local_sums = dppy.local.array(64, int32)

    local_sums[local_id] = 0

    if global_id < input_size:
        local_sums[local_id] = A[global_id]

    # Loop for computing local_sums : divide workgroup into 2 parts
    stride = group_size // 2
    while stride > 0:
        # Waiting for each 2x2 addition into given workgroup
        dppy.barrier(dppy.CLK_LOCAL_MEM_FENCE)

        # Add elements 2 by 2 between local_id and local_id + stride
        if local_id < stride:
            local_sums[local_id] += local_sums[local_id + stride]

        stride >>= 1

    if local_id == 0:
        partial_sums[group_id] = local_sums[0]
```

```
def sum_recursive_reduction(size, group_size, Dinp, Dpartial_sums):
    result = 0
    nb_work_groups = 0
    passed_size = size

    if size <= group_size:
        nb_work_groups = 1
    else:
        nb_work_groups = size // group_size
        if size % group_size != 0:
            nb_work_groups += 1
            passed_size = nb_work_groups * group_size

    sum_reduction_kernel[passed_size, group_size](Dinp, size, Dpartial_sums)

    if nb_work_groups <= group_size:
        sum_reduction_kernel[group_size, group_size](
            Dpartial_sums, nb_work_groups, Dinp
        )
        result = Dinp[0]
    else:
        result = sum_recursive_reduction(
            nb_work_groups, group_size, Dpartial_sums, Dinp
        )
```

(continues on next page)

(continued from previous page)

```
    return result
```

```
def sum_reduce(A):
    global_size = len(A)
    work_group_size = 64
    nb_work_groups = global_size // work_group_size
    if (global_size % work_group_size) != 0:
        nb_work_groups += 1

    partial_sums = np.zeros(nb_work_groups).astype(A.dtype)

    context = get_context()
    with dpctl.device_context(context):
        inp_buf = dpctl_mem.MemoryUSMShared(A.size * A.dtype.itemsize)
        inp_ndarray = np.ndarray(A.shape, buffer=inp_buf, dtype=A.dtype)
        np.copyto(inp_ndarray, A)

        partial_sums_buf = dpctl_mem.MemoryUSMShared(
            partial_sums.size * partial_sums.dtype.itemsize
        )
        partial_sums_ndarray = np.ndarray(
            partial_sums.shape, buffer=partial_sums_buf, dtype=partial_sums.dtype
        )
        np.copyto(partial_sums_ndarray, partial_sums)

        result = sum_recursive_reduction(
            global_size, work_group_size, inp_ndarray, partial_sums_ndarray
        )

    return result
```

Note: `numba-dppy` does not yet provide any analogue to the `numba.cuda.reduce` decorator for writing reductions kernel. Such decorator will be added in future releases.

4.6.2 Full examples

- `numba_dppy/examples/sum_reduction_recursive_ocl.py`
- `numba_dppy/examples/sum_reduction_ocl.py`
- `numba_dppy/examples/sum_reduction.py`

4.7 Universal Functions

Numba provides a set of decorators to create NumPy universal functions-like routines that are JIT compiled. Although, a close analog to NumPy universal functions Numba's `@vectorize` are not fully compatible with a regular NumPy ufunc.. Refer Creating NumPy universal functions for details.

`numba-dppy` only supports `numba.vectorize` decorator and not yet the `numba.guvectorize` decorator. Another present limitation is that `numba-dppy` ufunc kernels cannot invoke `numba_dppy.kernel` functions. Ongoing work is in progress to address these limitations.

4.7.1 Example 1: Basic Example

Full example can be found at `numba_dppy/examples/vectorize.py`.

```
@vectorize(nopython=True)
def ufunc_kernel(x, y):
    return x + y
```

```
def test_ufunc():
    N = 10
    dtype = np.float64

    A = np.arange(N, dtype=dtype)
    B = np.arange(N, dtype=dtype) * 10

    context = get_context()
    with dpctl.device_context(context):
        C = ufunc_kernel(A, B)

    print(C)
```

4.7.2 Example 2: Calling `numba.vectorize` inside a `numba_dppy.kernel`

Full example can be found at `numba_dppy/examples/blacksholes_njit.py`.

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

Note: `numba.cuda` requires `target='cuda'` parameter for `numba.vectorize` and `numba.guvectorize` functions. `numba-dppy` eschews the `target` parameter for `@vectorize` and infers the target from the `dpctl.device_context` in which the `numba.vectorize` function is called.

4.7.3 Full Examples

- numba_dppy/examples/vectorize.py
- numba_dppy/examples/blacksholes_njit.py

4.8 Supported Python Features in a numba-dppy Kernel

This page lists the Python features supported inside a numba_dppy.kernel function.

4.8.1 Built-in types

Supported Types

- int
- float

Unsupported Types

- complex
- bool
- None
- tuple

4.8.2 Built-in functions

The following built-in functions are supported:

- abs()
- float
- int
- len()
- range()
- round()

4.8.3 Standard library modules

The following functions from the math module are supported:

- math.acos()
- math.asin()
- math.atan()
- math.acosh()
- math.asinh()
- math.atanh()

- `math.cos()`
- `math.sin()`
- `math.tan()`
- `math.cosh()`
- `math.sinh()`
- `math.tanh()`
- `math.erf()`
- `math.erfc()`
- `math.exp()`
- `math.expm1()`
- `math.fabs()`
- `math.gamma()`
- `math.lgamma()`
- `math.log()`
- `math.log10()`
- `math.log1p()`
- `math.sqrt()`
- `math.ceil()`
- `math.floor()`

The following functions from the operator module are supported:

- `operator.add()`
- `operator.eq()`
- `operator.floordiv()`
- `operator.ge()`
- `operator.gt()`
- `operator.iadd()`
- `operator.ifloordiv()`
- `operator.imod()`
- `operator.imul()`
- `operator.ipow()`
- `operator.isub()`
- `operator.itruediv()`
- `operator.le()`
- `operator.lshift()`
- `operator.lt()`
- `operator.mod()`

- `operator.mul()`
- `operator.ne()`
- `operator.neg()`
- `operator.not_()`
- `operator.or_()`
- `operator.pos()`
- `operator.pow()`
- `operator.sub()`
- `operator.truediv()`

4.8.4 Unsupported Constructs

The following Python constructs are **not supported**:

- Exception handling (`try .. except`, `try .. finally`)
- Context management (the `with` statement)
- Comprehensions (either list, dict, set or generator comprehensions)
- Generator (any `yield` statements)
- The `raise` statement
- The `assert` statement

4.8.5 NumPy support

NumPy functions are whole array operations and are not supported within a `numba_dppy.kernel`.

CHAPTER

FIVE

DEBUGGING WITH GDB

`numba-dppy` allows SYCL kernels to be debugged with the GDB debugger. Setting the debug environment variable `NUMBA_DPPY_DEBUG` (e.g. `export NUMBA_DPPY_DEBUG=True`) enables the emission of debug information. To disable debugging, unset the variable, *i.e*, `export NUMBA_DPPY_DEBUG=`.

Not all GDB features supported by Numba on CPUs are yet supported in `numba-dppy`. Currently, the following debugging features are available:

- Source location (filename and line number).
- Setting break points by the line number.
- Stepping over break points.

5.1 Requirements

Intel® Distribution for GDB is needed for `numba-dppy`'s debugging features to work. Intel® Distribution for GDB is part of Intel oneApi and the relevant documentation can be found at Intel® Distribution for GDB documentation.

5.2 Example of GDB usage

```
$ export NUMBA_DPPY_DEBUG=True
$ gdb-oneapi -q python
(gdb) break numba_dppy/examples/sum.py:14  # Assumes the kernel is in file sum.py, at␣
↪line 14
(gdb) run sum.py
```

For example, given the following `numba-dppy` kernel code:

```
import numpy as np
import numba_dppy as dppy
import dpctl

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

global_size = 10
N = global_size
```

(continues on next page)

(continued from previous page)

```
a = np.array(np.random.random(N), dtype=np.float32)
b = np.array(np.random.random(N), dtype=np.float32)
c = np.ones_like(a)

with dpctl.device_context("opencl:gpu") as gpu_queue:
    data_parallel_sum[global_size, dppy.DEFAULT_LOCAL_SIZE](a, b, c)
```

GDB output:

```
Thread 2.2 hit Breakpoint 1,  with SIMD lanes [0-7], dppl_py_devfn__5F__5F_main_5F__
↪5F__2E_data_5F_parallel_5F_sum_24_1_2E_array_28_float32_2C__20_1d_2C__20_C_29__2E_
↪array_28_float32_2C__20_1d_2C__20_C_29__2E_array_28_float32_2C__20_1d_2C__20_C_29_␣
↪() at sum.py:14
14          i = dppy.get_global_id(0)
(gdb)
(gdb) n  # Making step
15          c[i] = a[i] + b[i]
```

5.3 Limitations

Currently, `numba-dppy` provides only initial support of debugging SYCL kernels. The following functionalities are **not supported** :

- Printing kernel local variables (e.g. `info locals`).
- Stepping over several offloaded functions.

CHAPTER

SIX

NUMBA-DPPY FOR NUMBA.CUDA DEVELOPERS

Todo: Coming soon.

CHAPTER

SEVEN

ABOUT

`numba-dppy` is developed by Intel and is part of the Intel Distribution for Python.

CHAPTER

EIGHT

CONTRIBUTING

Refer the contributing guide for information on coding style and standards used in `numba-dppy`.

CHAPTER

NINE

LICENSE

numba-dppy is Licensed under Apache License 2.0 that can be found in LICENSE. All usage and contributions to the project are subject to the terms and conditions of this license.