Bohrium Documentation

Release 0.9.0

eScience Group @ NBI

Apr 16, 2018

Contents

Bohrium provides automatic acceleration of array operations in Python/NumPy, C, and C++ targeting multi-core CPUs and GP-GPUs. Forget handcrafting CUDA/OpenCL to utilize your GPU and forget threading, mutexes and locks to utilize your multi-core CPU just use Bohrium!

CHAPTER 1

Features

	Architecture Support		Frontends			
	Multi-Core CPU	Many-Core GPU	Python2/NumPy	Python3/NumPy	C	C++
Linux	✓	✓	✓	✓	✓	✓
Mac OS	✓	✓	✓		✓	✓

- **Lazy Evaluation**, Bohrium will lazy evaluate all Python/NumPy operations until it encounters a "Python Read" such a printing an array or having a if-statement testing the value of an array.
- **Views** Bohrium supports NumPy views fully thus operating on array slices does not involve data copying.
- **Loop Fusion**, Bohrium uses a fusion algorithm that fuses (or merges) array operations into the same computation kernel that are then JIT-compiled and executed. However, Bohrium can only fuse operations that have some common sized dimension and no horizontal data conflicts.
- **Lazy CPU/GPU Communication**, Bohrium only moves data between the host and the GPU when the data is accessed directly by Python or a Python C-extension.
- **python -m bohrium**, automatically makes `import numpy` use Bohrium.
- Jupyter Support, you can use the magic command `%%bohrium` to automatically use Bohrium as NumPy.
- **Zero-copy *Interoperability* with:**
  - NumPy
  - Cython
  - PyOpenCL
  - PyCUDA

Please note:

- Bohrium is a 64-bit project exclusively.
- Source code is available here: https://github.com/bh107/bohrium

CHAPTER 2

Get Started!

2.1 Installation

Bohrium supports Linux and Mac OS.

2.1.1 Linux

PyPI Package

If you use Bohrium through Python, we strongly recommend to install Bohrium through pypi, which will include BLAS, LAPACK, OpenCV, and OpenCL support:

```
pip install --user bohrium
```

Note: On linux, Bohrium requires `gcc` in `$PATH`. E.g. on Ubuntu install the build-essential package: `sudo apt install build-essential`.

Ubuntu

On Ubuntu you can use apt-get:

```
sudo add-apt-repository ppa:bohrium/nightly
sudo apt-get update
sudo apt-get install bohrium
# Optionals
sudo apt-get install bohrium-opencl # GPU support
sudo apt-get install bohrium-visualizer # data visualizing
sudo apt-get install bohrium3 # Python3 support
```

Anaconda

To use Anaconda, simply install the Bohrium PyPI package in an environment:

```
# Activate the environment where you want to install Bohrium:
source activate my_env
# Install Bohrium using pip
pip install bohrium
```

Note: Bohrium requires `gcc` in `$PATH`. E.g. on Ubuntu install the build-essential package: `sudo apt install build-essential`.

Install From Source Package

Visit Bohrium on github.com and download the latest release: https://github.com/bh107/bohrium/releases/latest. Then build and install Bohrium as described in the following subsections.

You need to install all packages required to build NumPy:

```
sudo apt-get build-dep python-numpy
```

And some additional packages:

```
sudo apt-get install python-numpy python-dev swig cmake unzip cython libhwloc-dev␣
↪libboost-filesystem-dev libboost-serialization-dev libboost-regex-dev zlib1g-dev␣
↪libsigsegv-dev
```

And for python v3 support:

```
sudo apt-get install python3-dev python3-numpy python3-dev cython3
```

Packages for visualization:

```
sudo apt-get install freeglut3 freeglut3-dev libxmu-dev libxi-dev
```

Build and install:

```
wget https://github.com/bh107/bohrium/archive/master.zip
unzip master.zip
cd bohrium-master
mkdir build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX=<path to install directory>
make
make install
```

Note: The default install directory is `~/.local`

Note: To compile to a custom Python (with valgrind debug support for example), set `-DPYTHON_EXECUTABLE=<custom python binary>`.

Finally, you need to set the `LD_LIBRARY_PATH` environment variables and if you didn't install Bohrium in `$HOME/.local/lib` your need to set `PYTHONPATH` as well.

The `LD_LIBRARY_PATH` should include the path to the installation directory:

```
export LD_LIBRARY_PATH="<install dir>:$LD_LIBRARY_PATH"
```

The `PYTHONPATH` should include the path to the newly installed Bohrium Python module:

```
export PYTHONPATH="<install dir>/lib/python<python version>/site-packages:$PYTHONPATH"
```

Check Your Installation

Check installation by printing the current runtime stack:

```
python -m bohrium --info
```

2.1.2 Mac OS

The following explains how to get going on Mac OS.

You need to install the Xcode Developer Tools package, which is found in the App Store.

PyPI Package

If you use Bohrium through Python, we strongly recommend to install Bohrium through pypi, which will include BLAS, LAPACK, OpenCV, and OpenCL support:

```
python -m pip install --user bohrium
```

Homebrew

Start by installing Homebrew as explained on their website

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

Then install Bohrium:

```
pip install cython # This dependency cannot be installed via brew.
brew tap bh107/bohrium
brew tap homebrew/science # for clblas and the likes
brew install bohrium # you can add additional options, see `brew info bohrium`
```

Install From Source Package

Start by installing Homebrew as explained on their website

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

Install dependencies:

```
brew install python
brew install cmake
brew install boost --with-icu4c
brew install libsigsegv
python3 -m pip install --user numpy cython twine
```

Visit Bohrium on github.com, download the latest release: https://github.com/bh107/bohrium/releases/latest or download *master*, and then build it:

```
wget https://github.com/bh107/bohrium/archive/master.zip
unzip master.zip
cd bohrium-master
mkdir build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX=<path to install directory>
make
make install
```

Note: The default install directory is `~/.local`

Note: To compile to a custom Python (with valgrind debug support for example), set `-DPYTHON_EXECUTABLE=<custom python binary>`.

Finally, you need to set the `LD_LIBRARY_PATH` environment variables and if you didn't install Bohrium in `$HOME/.local/lib` your need to set `PYTHONPATH` as well.

The `LD_LIBRARY_PATH` should include the path to the installation directory:

```
export LD_LIBRARY_PATH="<install dir>:$LD_LIBRARY_PATH"
```

The `PYTHONPATH` should include the path to the newly installed Bohrium Python module:

```
export PYTHONPATH="<install dir>/lib/python<python version>/site-packages:$PYTHONPATH"
```

Check Your Installation

Check installation by printing the current runtime stack:

```
python -m bohrium --info
```

2.1.3 Installation using Spack

This guide will install Bohrium using the Spack package manager.

Why use Spack?

Spack is a package management tool tailored specifically for supercomputers with a rather dated software stack. It allows to install and maintain packages, starting only from very few dependencies: Pretty much just python2.6, git, curl and some c++ compiler are all that's needed for the bootstrap.

Needless to say that the request for installing a particular package automatically yields the installation of all dependencies with exactly the right version and configurations. If this causes multiple versions/configurations of the same package to be required, this is no problem and gets resolved automatically, too. As a bonus on top, using an installed package later is super easy as well due to an automatic generation of module files, which set the required environment up.

Installation overview

First step is to clone and setup Spack:

```
export SPACK_ROOT="$PWD/spack"
git clone https://github.com/llnl/spack.git
. $SPACK_ROOT/share/spack/setup-env.sh
```

Afterwards the installation of Bohrium is instructed:

```
spack install bohrium
```

This step will take a while, since Spack will download the sources of all dependencies, unpack, configure and compile them. But since everything happens in the right order automatically, you could easily do this over night.

That's it. If you want to use Bohrium, setup up Spack as above, then load the required modules:

```
spack module loads -r bohrium > /tmp/bohrium.modules
. /tmp/bohrium.modules
```

and you are ready to go as the shell environment now contains all required variables (*LD_LIBRARY_PATH*, *PATH*, *CPATH*, *PYTHONPATH*, ...) to get going.

If you get some errors about the command *module* not being found, you need to install the Spack package *environment-modules* beforehand. Again, just a plain:

```
spack install environment-modules
```

is enough to achieve this.

Tuning the installation procedure

Spack offers countless ways to influence how things are installed and what is installed. See the Documentation and especially the Getting Started section for a good overview.

Most importantly the so-called *spec* allows to specify features or requirements with respect to versions and dependencies, that should be enabled or disabled when building the package. For example:

```
spec install bohrium~cuda~opencl
```

Will install Bohrium *without* CUDA or OpenCL support, which has a dramatic impact on the install time due to the reduced amount of dependencies to be installed. On the other hand:

```
spec install bohrium@develop
```

will install specifically the development version of Bohrium. This the current *HEAD* of the *master* branch in the github repository. One may also influence the versions of the dependencies by themselves. For example:

```
spec install bohrium+python^python@3:
```

will specifically compile Bohrium with a python version larger than 3.

The current list of features the Bohrium package has to offer can be listed by the command:

```
spack info bohrium
```

and the list of dependencies which will be installed by a particlar *spec* can be easily reviewed by something like:

```
spack spec bohrium@develop~cuda~opencl
```

2.2 User Guide

2.2.1 Python/NumPy

Runtime Info

Print the current Bohrium runtime stack:

```
python -m bohrium --info
```

Automatic Parallelization

Bohrium implements a new python module `bohrium` that introduces a new array class `bohrium.ndarray` which inherits from `numpy.ndarray`. The two array classes are fully compatible thus one only has to replace `numpy.ndarray` with `bohrium.ndarray` in order to utilize the Bohrium runtime system.

The following example is a heat-equation solver that uses Bohrium. Note that the only difference between Bohrium code and NumPy code is the first line where we import bohrium as np instead of numpy as np:

```
import bohrium as np
def heat2d(height, width, epsilon=42):
  G = np.zeros((height+2,width+2),dtype=np.float64)
  G[:,0]  = -273.15
  G[:,-1] = -273.15
  G[-1,:] = -273.15
  G[0,:]  = 40.0
  center = G[1:-1,1:-1]
  north  = G[:-2,1:-1]
  south  = G[2:,1:-1]
  east   = G[1:-1,:-2]
  west   = G[1:-1,2:]
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(center+north+south+east+west)
    delta = np.sum(np.abs(tmp-center))
    center[:] = tmp
  return center
heat2d(100, 100)
```

Alternatively, you can import Bohrium as NumPy through the command line argument `-m bohrium`:

```
python -m bohrium heat2d.py
```

In this case, all instances of `import numpy` is converted to `import bohrium` seamlessly. If you need to access the real numpy module use `import numpy_force`.

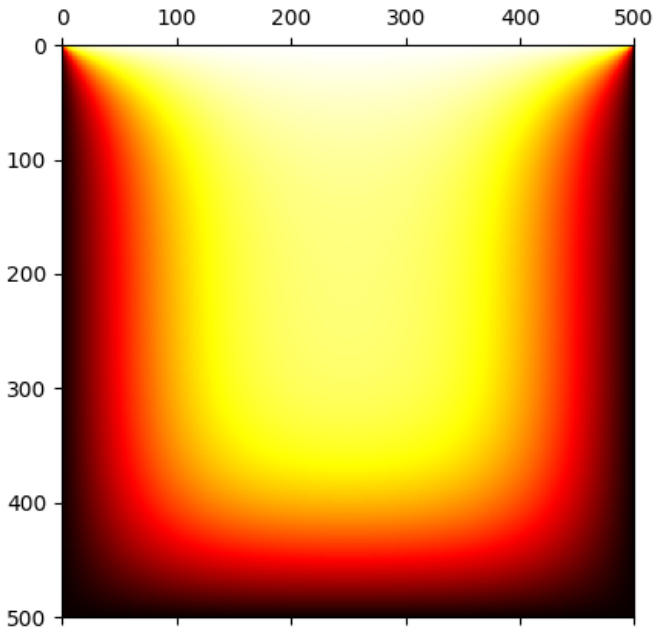
Acceleration

The approach of Bohrium is to accelerate all element-wise functions in NumPy (aka universal functions) as well as the reductions and accumulations of element-wise functions. This approach makes it possible to accelerate the heat-equation solver on both multi-core CPUs and GPUs.

Beside element-wise functions, Bohrium also accelerates a selection of common NumPy functions such as `dot()` and `solve()`. But the number of functions in NumPy and related projects such as SciPy is enormous thus we cannot hope to accelerate every single function in Bohrium. Instead, Bohrium will automatically convert `bohrium.ndarray` to `numpy.ndarray` when encountering a function that Bohrium cannot accelerate. When running on the CPU, this conversion is very cheap but when running on the GPU, this conversion requires the array data to be copied from the GPU to the CPU.

Matplotlib's `matshow()` function is example of a function Bohrium cannot accelerate. Say we want to visualize the result of the heat-equation solver, we could use `matshow()`:

```
from matplotlib import pyplot as plt

res = heat2d(100, 100)
plt.matshow(res, cmap='hot')
plt.show()
```



Beside producing the image (after approx. 1 min), the execution will raise a Python warning informing you that matplotlib function is handled like a regular NumPy:

```
/usr/lib/python2.7/site-packages/matplotlib/cbook.py:1506: RuntimeWarning:
Encountering an operation not supported by Bohrium. It will be handled by the␣
↪original NumPy.
x = np.array(x, subok=True, copy=copy)
```

Note: Increasing the problem size will improve the performance of Bohrium significantly!

Convert between Bohrium and NumPy

It is possible to convert between Bohrium and NumPy explicitly and thus avoid Python warnings. Let's walk through an example:

Create a new NumPy array with ones:

```
np_ary = numpy.ones(42)
```

Convert any type of array to Bohrium:

```
bh_ary = bohrium.array(np_ary)
```

Copy a bohrium array into a new NumPy array:

```
npy2 = bh_ary.copy2numpy()
```

Accelerate Loops

As we all know, having for and while loops in Python is bad for performance but is sometimes necessary. E.g. in the case of the `heat2d()` code, we have to evaluate `delta > epsilon` in order to know when to stop iterating. To address this issue, Bohrium introduces the function `do_while()`, which takes a function and calls it repeatedly until either a maximum number of calls has been reached or until the function return False.

The function signature:

```
def do_while(func, niters, *args, **kwargs):
    """Repeatedly calls the `func` with the `*args` and `**kwargs` as argument.

    The `func` is called while `func` returns True or None and the maximum number
    of iterations, `niters`, hasn't been reached.

    Parameters
    ----------
    func : function
        The function to run in each iterations. `func` can take any argument and may␣
↪return
        a boolean `bharray` with one element.
    niters: int or None
        Maximum number of iterations in the loop (number of times `func` is called).␣
↪If None, there is no maximum.
    *args, **kwargs : list and dict
        The arguments to `func`

    Notes
    -----
    `func` can only use operations supported natively in Bohrium.
    """
```

An example where the function doesn't return anything:

```
>>> def loop_body(a):
...     a += 1
>>> a = bh.zeros(4)
>>> bh.do_while(loop_body, 5, a)
>>> a
array([5, 5, 5, 5])
```

An example where the function returns a `bharray` with one element and of type `bh.bool`:

```
>>> def loop_body(a):
...     a += 1
...     return bh.sum(a) < 10
>>> a = bh.zeros(4)
>>> bh.do_while(loop_body, None, a)
>>> a
array([3, 3, 3, 3])
```

Sliding Views Between Iterations

It can be useful to increase/decrease the beginning of certain array views between iterations of a loop. This can be achieved using `get_iterator()`, which returns a special bohrium iterator. The iterator can be given an optional start value (0 by default). The iterator is increased by one for each iteration, but can be changed increase or decrease by multiplying any constant (see example 2).

Iterators only supports addition, subtraction and multiplication. `get_iterator()` can only be used within Bohrium loops. Views using iterators cannot change shape between iterations. Therefore, views such as `a[i:2*i]` are not supported.

Example 1. Using iterators to create a loop-based function for calculating the triangular numbers (from 1 to 10). The loop in numpy looks the following:

```
>>> a = np.arange(1,11)
>>> for i in range(0,9):
...     a[i+1] += a[i]
>>> a
array([1 3 6 10 15 21 28 36 45 55])
```

The same can be written in Bohrium as:

```
>>> def loop_body(a):
...     i = get_iterator()
...     a[i+1] += a[i]
>>> a = bh.arange(1,11)
>>> bh.do_while(loop_body, 9, a)
>>> a
array([1 3 6 10 15 21 28 36 45 55])
```

Example 2. Increasing every second element by one, starting at both ends, in the same loop. As it can be seen: *i* is increased by 2, while *j* is descreased by 2 for each iteration:

```
>>> def loop_body(a):
...   i = get_iterator(1)
...   a[2*i] += a[2*(i-1)]
...   j = i+1
...   a[1-2*j] += a[1-2*(j-1)]
>>> a = bh.ones(10)
>>> bh.for_loop(loop_body, 4, a)
>>> a
array([1 5 2 4 3 3 4 2 5 1])
```

Interoperability

Bohrium is interoperable with other popular Python projects such as Cython and PyOpenCL. The idea is that if you encounter a problem that you cannot implement using array programming and Bohrium cannot accelerate, you can manually accelerate that problem using Cython or PyOpenCL.

NumPy

One example of such a problem is *bincount()* from NumPy. *bincount()* computes a histogram of an array, which isn't possible to implement efficiently through array programming. One approach is simply to use the implementation of NumPy:

```
import numpy
import bohrium

def bincount_numpy(ary):
    # Make a NumPy copy of the Bohrium array
    np_ary = ary.copy2numpy()
    # Let NumPy handle the calculation
```

(continues on next page)

(continued from previous page)

```
    result = numpy.bincount(np_ary)
    # Copy the result back into a new Bohrium array
    return bohrium.array(result)
```

In this case, we use *bohrium.copy2numpy()* and *bohrium.array()* to copy the Bohrium to NumPy and back again.

Cython

In order to parallelize *bincount()* for a multi-core CPU, one can use Cython:

```
import numpy as np
import bohrium
import cython
from cython.parallel import prange, parallel
from libc.stdlib cimport abort, malloc, free
cimport numpy as cnp
cimport openmp
ctypedef cnp.uint64_t uint64

@cython.boundscheck(False) # turn off bounds-checking
@cython.cdivision(True) # turn off division-by-zero checking
cdef _count(uint64[:] x, uint64[:] out):
    cdef int num_threads, thds_id
    cdef uint64 i, start, end
    cdef uint64* local_histo

    with nogil, parallel():
        num_threads = openmp.omp_get_num_threads()
        thds_id = openmp.omp_get_thread_num()
        start = (x.shape[0] / num_threads) * thds_id
        if thds_id == num_threads-1:
            end = x.shape[0]
        else:
            end = start + (x.shape[0] / num_threads)

        if not(thds_id < num_threads-1 and x.shape[0] < num_threads):
            local_histo = <uint64 *> malloc(sizeof(uint64) * out.shape[0])
            if local_histo == NULL:
                abort()
            for i in range(out.shape[0]):
                local_histo[i] = 0

            for i in range(start, end):
                local_histo[x[i]] += 1

            with gil:
                for i in range(out.shape[0]):
                    out[i] += local_histo[i]
            free(local_histo)


def bincount_cython(x, minlength=None):
    # The output `ret` has the size of the max element plus one
    ret = bohrium.zeros(x.max()+1, dtype=x.dtype)
```

(continues on next page)

(continued from previous page)

```
    # To reduce overhead, we use `interop_numpy.get_array()` instead of `copy2numpy()`
    # This approach means that `x_buf` and `ret_buf` points to the same memory as `x`␣
↪and `ret`.
    # Therefore, only change or deallocate `x` and `ret` when you are finished using␣
↪`x_buf` and `ret_buf`.
    x_buf = bohrium.interop_numpy.get_array(x)
    ret_buf = bohrium.interop_numpy.get_array(ret))

    # Now, we can run the Cython function
    _count(x_buf, ret_buf))

    # Since `ret_buf` points to the memory of `ret`, we can simply return `ret`.
    return ret
```

The function *_count()* is a regular Cython function that performs the histogram calculation. The function *bincount_cython()* uses *bohrium.interop_numpy.get_array()* to retrieve data pointers from the Bohrium arrays without any data copying.

PyOpenCL

In order to parallelize *bincount()* for a GPGPU, one can use PyOpenCL:

```
import bohrium
import pyopencl as cl

def bincount_pyopencl(x):
    # Check that PyOpenCL is installed and that the Bohrium runtime uses the OpenCL␣
↪backend
    if not interop_pyopencl.available():
        raise NotImplementedError("OpenCL not available")

    # Get the OpenCL context from Bohrium
    ctx = bohrium.interop_pyopencl.get_context()
    queue = cl.CommandQueue(ctx)

    x_max = int(x.max())

    # Check that the size of histogram doesn't exceeds the memory capacity of the GPU
    if x_max >= interop_pyopencl.max_local_memory(queue.device) // x.itemsize:
        raise NotImplementedError("OpenCL: max element is too large for the GPU")

    # Let's create the output array and retrieve the in-/output OpenCL buffers
    # NB: we always return uint32 array
    ret = bohrium.empty((x_max+1, ), dtype=np.uint32)
    x_buf = bohrium.interop_pyopencl.get_buffer(x)
    ret_buf = bohrium.interop_pyopencl.get_buffer(ret)

    # The OpenCL kernel is based on the book "OpenCL Programming Guide" by Aaftab␣
↪Munshi at al.
    source = """
    kernel void histogram_partial(
        global DTYPE *input,
        global uint *partial_histo,
        uint input_size
    ){
```

(continues on next page)

(continued from previous page)

```
        int local_size = (int)get_local_size(0);
        int group_indx = get_group_id(0) * HISTO_SIZE;
        int gid = get_global_id(0);
        int tid = get_local_id(0);

        local uint tmp_histogram[HISTO_SIZE];

        int j = HISTO_SIZE;
        int indx = 0;

        // clear the local buffer that will generate the partial histogram
        do {
            if (tid < j)
                tmp_histogram[indx+tid] = 0;
            j -= local_size;
            indx += local_size;
        } while (j > 0);

        barrier(CLK_LOCAL_MEM_FENCE);

        if (gid < input_size) {
            atomic_inc(&tmp_histogram[input[gid]]);
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        // copy the partial histogram to appropriate location in
        // histogram given by group_indx
        if (local_size >= HISTO_SIZE){
            if (tid < HISTO_SIZE)
                partial_histo[group_indx + tid] = tmp_histogram[tid];
        }else{
            j = HISTO_SIZE;
            indx = 0;
            do {
                if (tid < j)
                    partial_histo[group_indx + indx + tid] = tmp_histogram[indx +
↪tid];

                j -= local_size;
                indx += local_size;
            } while (j > 0);
        }
    }

    kernel void histogram_sum_partial_results(
        global uint *partial_histogram,
        int num_groups,
        global uint *histogram
    ){
        int gid = (int)get_global_id(0);
        int group_indx;
        int n = num_groups;
        local uint tmp_histogram[HISTO_SIZE];

        tmp_histogram[gid] = partial_histogram[gid];
        group_indx = HISTO_SIZE;
```

(continues on next page)

(continued from previous page)

```
        while (--n > 0) {
            tmp_histogram[gid] += partial_histogram[group_indx + gid];
            group_indx += HISTO_SIZE;
        }
        histogram[gid] = tmp_histogram[gid];
    }
    """
    source = source.replace("HISTO_SIZE", "%d" % ret.shape[0])
    source = source.replace("DTYPE", interop_pyopencl.type_np2opencl_str(x.dtype))
    prg = cl.Program(ctx, source).build()

    # Calculate sizes for the kernel execution
    local_size = interop_pyopencl.kernel_info(prg.histogram_partial, queue)[0]  # Max␣
↪work-group size
    num_groups = int(math.ceil(x.shape[0] / float(local_size)))
    global_size = local_size * num_groups

    # First we compute the partial histograms
    partial_res_g = cl.Buffer(ctx, cl.mem_flags.WRITE_ONLY, num_groups * ret.nbytes)
    prg.histogram_partial(queue, (global_size,), (local_size,), x_buf, partial_res_g,␣
↪np.uint32(x.shape[0]))

    # Then we sum the partial histograms into the final histogram
    prg.histogram_sum_partial_results(queue, ret.shape, None, partial_res_g, np.
↪uint32(num_groups), ret_buf)
    return ret
```

The implementation is regular PyOpenCL and the OpenCL kernel is based on the book "OpenCL Programming Guide" by Aaftab Munshi et al. However, notice that we use *bohrium.interop_pyopencl.get_context()* to get the PyOpenCL context rather than pyopencl.create_some_context(). In order to avoid copying data between host and device memory, we use *bohrium.interop_pyopencl.get_buffer()* to create a OpenCL buffer that points to the device memory of the Bohrium arrays.

PyCUDA

The PyCUDA implementation is very similar to the PyOpenCL. Besides some minor difference in the kernel source code, we use *interop_pycuda.init()* to initiate PyCUDA and use *interop_pycuda.get_gpuarray()* to get the CUDA buffers from the Bohrium arrays:

```
def bincount_pycuda(x, minlength=None):
    """PyCUDA implementation of `bincount()`"""

    if not interop_pycuda.available():
        raise NotImplementedError("CUDA not available")

    import pycuda
    from pycuda.compiler import SourceModule

    interop_pycuda.init()

    x_max = int(x.max())
    if x_max < 0:
        raise RuntimeError("bincount(): first argument must be a 1 dimensional, non-
↪negative int array")
    if x_max > np.iinfo(np.uint32).max:
```

(continues on next page)

(continued from previous page)

```
        raise NotImplementedError("CUDA: the elements in the first argument must fit␣
↪in a 32bit integer")
    if minlength is not None:
        x_max = max(x_max, minlength)

    # TODO: handle large max element by running multiple bincount() on a range
    if x_max >= interop_pycuda.max_local_memory() // x.itemsize:
        raise NotImplementedError("CUDA: max element is too large for the GPU")

    # Let's create the output array and retrieve the in-/output CUDA buffers
    # NB: we always return uint32 array
    ret = array_create.ones((x_max+1, ), dtype=np.uint32)
    x_buf = interop_pycuda.get_gpuarray(x)
    ret_buf = interop_pycuda.get_gpuarray(ret)

    # CUDA kernel is based on the book "OpenCL Programming Guide" by Aaftab Munshi et␣
↪al.
    source = """
    __global__ void histogram_partial(
        DTYPE *input,
        uint *partial_histo,
        uint input_size
    ){
        int local_size = blockDim.x;
        int group_indx = blockIdx.x * HISTO_SIZE;
        int gid = (blockIdx.x * blockDim.x + threadIdx.x);
        int tid = threadIdx.x;

        __shared__ uint tmp_histogram[HISTO_SIZE];

        int j = HISTO_SIZE;
        int indx = 0;

        // clear the local buffer that will generate the partial histogram
        do {
            if (tid < j)
                tmp_histogram[indx+tid] = 0;
            j -= local_size;
            indx += local_size;
        } while (j > 0);

        __syncthreads();

        if (gid < input_size) {
            atomicAdd(&tmp_histogram[input[gid]], 1);
        }

        __syncthreads();

        // copy the partial histogram to appropriate location in
        // histogram given by group_indx
        if (local_size >= HISTO_SIZE){
            if (tid < HISTO_SIZE)
                partial_histo[group_indx + tid] = tmp_histogram[tid];
        }else{
            j = HISTO_SIZE;
            indx = 0;
```

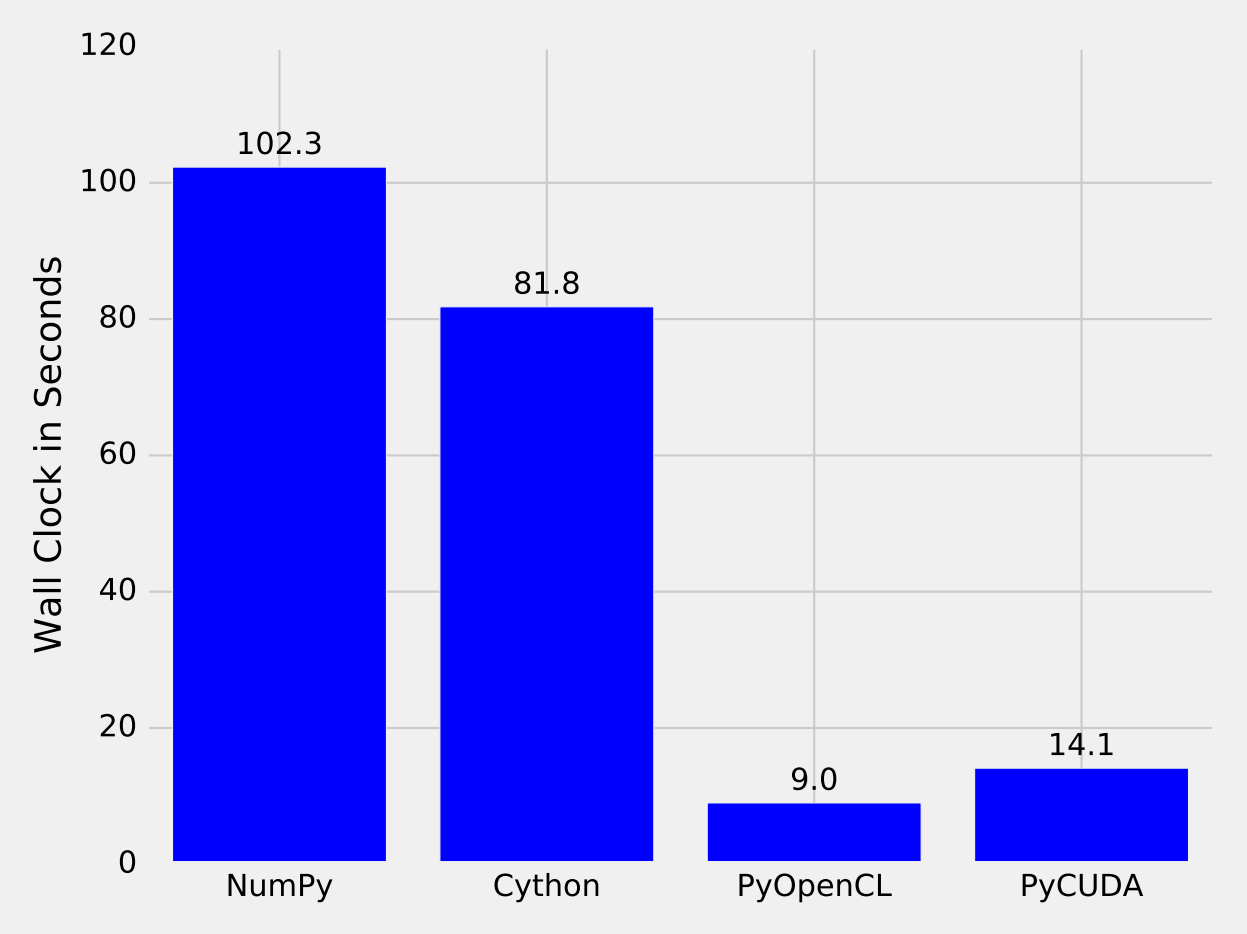
(continues on next page)

(continued from previous page)

```
            do {
                if (tid < j)
                    partial_histo[group_indx + indx + tid] = tmp_histogram[indx +
↪tid];

                j -= local_size;
                indx += local_size;
            } while (j > 0);
        }
    }

    __global__ void histogram_sum_partial_results(
        uint *partial_histogram,
        int num_groups,
        uint *histogram
    ){
        int gid = (blockIdx.x * blockDim.x + threadIdx.x);
        int group_indx;
        int n = num_groups;
        __shared__ uint tmp_histogram[HISTO_SIZE];

        tmp_histogram[gid] = partial_histogram[gid];
        group_indx = HISTO_SIZE;
        while (--n > 0) {
            tmp_histogram[gid] += partial_histogram[group_indx + gid];
            group_indx += HISTO_SIZE;
        }
        histogram[gid] = tmp_histogram[gid];
    }
    """
    source = source.replace("HISTO_SIZE", "%d" % ret.shape[0])
    source = source.replace("DTYPE", interop_pycuda.type_np2cuda_str(x.dtype))
    prg = SourceModule(source)

    # Calculate sizes for the kernel execution
    kernel = prg.get_function("histogram_partial")
    local_size = kernel.get_attribute(pycuda.driver.function_attribute.MAX_THREADS_
↪PER_BLOCK)  # Max work-group size
    num_groups = int(math.ceil(x.shape[0] / float(local_size)))
    global_size = local_size * num_groups

    # First we compute the partial histograms
    partial_res_g = pycuda.driver.mem_alloc(num_groups * ret.nbytes)
    kernel(x_buf, partial_res_g, np.uint32(x.shape[0]), block=(local_size, 1, 1),
↪grid=(num_groups, 1))

    # Then we sum the partial histograms into the final histogram
    kernel = prg.get_function("histogram_sum_partial_results")
    kernel(partial_res_g, np.uint32(num_groups), ret_buf, block=(1, 1, 1), grid=(ret.
↪shape[0], 1))
    return ret
```

Performance Comparison

Finally, let's compare the performance of the difference approaches. We run on a *Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz* with 4 CPU-cores and a *GeForce GTX Titan X (maxwell)*. The timing is wall-clock time including everything, in particular the host/device communication overhead.



The timing code:

```
import numpy as np
import time

SIZE = 500000000
ITER = 100

t1 = time.time()
a = np.minimum(np.arange(SIZE, dtype=np.int64), 64)
for _ in range(ITER):
    b = np.bincount(a)
t2 = time.time()
s = b.sum()
print ("Sum: %d, time: %f sec" % (s, t2 - t1))
```

Conclusion

Interoperability makes it possible to accelerate code that Bohrium doesn't accelerate automatically. The Bohrium team constantly works on improving the performance and increase the number of NumPy operations automatically accelerated but in some cases we simply have to give the user full control.

2.2.2 C++ library

Todo: Write a C++ user guide

For now, we refer to the header files and code examples in the source tree:

- https://github.com/bh107/bohrium/tree/master/bridge/cxx/include/bhxx
- https://github.com/bh107/bohrium/tree/master/bridge/cxx/examples

2.2.3 C library

The C interface introduces two array concepts:

- A base array that has a *rank* (number of dimensions) and *shape* (array of dimension sizes). The memory of the base array is always a single contiguous block of memory.
- A view array that, beside a *rank* and a *shape*, has a *start* (start offset in number of elements) and a *stride* (array of dimension strides in number of elements). The view array refers to a (sub)set of a underlying base array where *start* is the offset into the base array and *stride* is number of elements to skip in order to iterate one step in a given dimension.

API

The C interface consists of a broad range of functions – in the following, we describe some of the important ones.

Create a new empty array with *rank* number of dimensions and with the shape *shape* and returns a handler/pointer to a *complete* view of this new array:

```
bh_multi_array_{TYPE}_p bh_multi_array_{TYPE}_new_empty(uint64_t rank, const int64_t*␣
↪shape);
```

Get pointer/handle to the base of a view:

```
bh_base_p bh_multi_array_{TYPE}_get_base(const bh_multi_array_{TYPE}_p self);
```

Destroy the base array and the associated memory:

```
void bh_multi_array_{TYPE}_destroy_base(bh_base_p base);
```

Destroy the view and base array (but not the associated memory):

```
void bh_multi_array_{TYPE}_free(const bh_multi_array_{TYPE}_p self);
```

Some meta-data access functions:

```
// Gets the number of elements in the array
uint64_t bh_multi_array_{TYPE}_get_length(bh_multi_array_{TYPE}_p self);

// Gets the number of dimensions in the array
uint64_t bh_multi_array_{TYPE}_get_rank(bh_multi_array_{TYPE}_p self);

// Gets the number of elements in the dimension
uint64_t bh_multi_array_{TYPE}_get_dimension_size(bh_multi_array_{TYPE}_p self, const␣
↪int64_t dimension);
```

Before accessing the memory of an array, one has to synchronize the array:

```
void bh_multi_array_{TYPE}_sync(const bh_multi_array_{TYPE}_p self);
```

Access the memory of an array (remember to synchronize):

```
bh_{TYPE}* bh_multi_array_{TYPE}_get_base_data(bh_base_p base);
```

Some of the element-wise operations:

```
//Addition
void bh_multi_array_{TYPE}_add(bh_multi_array_{TYPE}_p out, const bh_multi_array_
↪{TYPE}_p lhs, const bh_multi_array_{TYPE}_p rhs);

//Multiply
void bh_multi_array_{TYPE}_multiply(bh_multi_array_{TYPE}_p out, const bh_multi_array_
↪{TYPE}_p lhs, const bh_multi_array_{TYPE}_p rhs);

//Addition: scalar + array
void bh_multi_array_{TYPE}_add_scalar_lhs(bh_multi_array_{TYPE}_p out, bh_{TYPE} lhs,␣
↪const bh_multi_array_{TYPE}_p rhs);
```

Some of the reduction and accumulate (aka scan) functions where *axis* is the dimension to reduce/accumulate over:

```
//Sum
void bh_multi_array_{TYPE}_add_reduce(bh_multi_array_{TYPE}_p out, const bh_multi_
↪array_{TYPE}_p in, bh_int64 axis);

//Prefix sum
void bh_multi_array_{TYPE}_add_accumulate(bh_multi_array_{TYPE}_p out, const bh_multi_
↪array_{TYPE}_p in, bh_int64 axis);
```

2.2.4 Runtime Configuration

Bohrium supports a broad range of front and back-ends. The default backend is OpenMP. You can change which backend to use by defining the `BH_STACK` environment variable:

- The CPU backend that make use of OpenMP: `BH_STACK=openmp`
- The GPU backend that make use of OpenCL: `BH_STACK=opencl`
- The GPU backend that make use of CUDA: `BH_STACK=cude`

For debug information when running Bohrium, use the following environment variables:

```
BH_<backend>_PROF=true      -- Prints a performance profile at the end of execution.
BH_<backend>_VERBOSE=true   -- Prints a lot of information including the source of the␣
↪JIT compiled kernels. Enables per-kernel profiling when used together with BH_
↪OPENMP_PROF=true.
BH_SYNC_WARN=true           -- Show Python warnings in all instances when copying data␣
↪to Python.
BH_MEM_WARN=true            -- Show warnings when memory accesses are problematic.
BH_<backend>_GRAPH=true     -- Dump a dependency graph of the instructions send to the␣
↪back-ends (.dot file).
BH_<backend>_VOLATILE=true -- Declare temporary variables using `volatile`, which␣
↪avoid precision differences because of Intel's use of 80-bit floats internally.
```

Particularly, `BH_<backend>_PROF=true` is very useful to explore why Bohrium might not perform as expected:

```
BH_OPENMP_PROF=1 python -m bohrium heat_equation.py --size=4000*4000*100
heat_equation.py - target: bhc, bohrium: True, size: 4000*4000*100, elapsed-time: 6.
↪446084

[OpenMP] Profiling:
Fuse cache hits:                199/203 (98.0296%)
Codegen cache hits              299/304 (98.3553%)
Kernel cache hits               300/304 (98.6842%)
Array contractions:             700/1403 (49.8931%)
Outer-fusion ratio:             13/23 (56.5217%)

Max memory usage:               0 MB
Syncs to NumPy:                 99
Total Work:                     12800400099 operations
Throughput:                     1.9235e+09ops
Work below par-threshold (1000): 0%

Wall clock:                     6.65473s
Total Execution:                6.04354s
  Pre-fusion:                   0.000761211s
  Fusion:                       0.00411354s
  Codegen:                      0.00192224s
  Compile:                      0.285544s
  Exec:                         4.91214s
  Copy2dev:                     0s
  Copy2host:                    0s
  Ext-method:                   0s
  Offload:                      0s
  Other:                        0.839052s

Unaccounted for (wall - total):  0.611198s
```

Which tells us, among other things, that the execution of the compiled JIT kernels (`Exec`) takes 4.91 seconds, the JIT compilation (`Compile`) takes 0.29 seconds, and the time spend outside of Bohrium (`Unaccounted for`) takes 0.61.

OpenCL Configuration

In order to choose which OpenCL platform and device to use, set the following environment variables:

```
# OpenCL platform. -1 means automatic. Other numbers will index into list of␣
↪platforms.
BH_OPENCL_PLATFORM_NO = -1

# Device type can be one of 'auto', 'gpu', 'cpu', 'accelerator', or 'default'
BH_OPENCL_DEVICE_TYPE = auto
```

You can also set the options in the configure file under the `[opencl]` section.

Also under the `[opencl]` section, you can set the OpenCL work group sizes:

```
# OpenCL work group sizes
work_group_size_1dx = 128
work_group_size_2dx = 32
work_group_size_2dy = 4
work_group_size_3dx = 32
work_group_size_3dy = 2
work_group_size_3dz = 2
```

Advanced Configuration

In order to configure the runtime setup of Bohrium you must provide a configuration file to Bohrium. The installation of Bohrium installs a default configuration file in `/etc/bohrium/config.ini` when doing a system-wide installation, `~/.bohrium/config.ini` when doing a local installation, and `<python library>/bohrium/config.ini` when doing a pip installation.

At runtime Bohrium will search through the following prioritized list in order to find the configuration file:

- The environment variable `BH_CONFIG`
- The config within the Python package `bohrium/config.ini` (in the same directory as `__init__.py`)
- The home directory config `~/.bohrium/config.ini`
- The system-wide config `/usr/local/etc/bohrium/config.ini`
- The system-wide config `/etc/bohrium/config.ini`

The default configuration file looks similar to the config below:

```
#
# Stack configurations, which are a comma separated lists of components.
# NB: 'stacks' is a reserved section name and 'default'
#     is used when 'BH_STACK' is unset.
#     The bridge is never part of the list
#
[stacks]
default    = bcexp, bccon, node, openmp
openmp     = bcexp, bccon, node, openmp
opencl     = bcexp, bccon, node, opencl, openmp

#
# Managers
#

[node]
impl = /usr/lib/libbh_vem_node.so
timing = false
```

(continues on next page)

(continued from previous page)

```
[proxy]
address = localhost
port = 4200
impl = /usr/lib/libbh_vem_proxy.so


#
# Filters - Helpers / Tools
#
[pprint]
impl = /usr/lib/libbh_filter_pprint.so

#
# Filters - Bytecode transformers
#
[bccon]
impl = /usr/lib/libbh_filter_bccon.so
collect = true
stupidmath = true
muladd = true
reduction = false
find_repeats = false
timing = false
verbose = false

[bcexp]
impl = /usr/lib/libbh_filter_bcexp.so
powk = true
sign = false
repeat = false
reduce1d = 32000
timing = false
verbose = false

[noneremover]
impl = /usr/lib/libbh_filter_noneremover.so
timing = false
verbose = false

#
# Engines
#
[openmp]
impl = /usr/lib/libbh_ve_openmp.so
tmp_bin_dir = /usr/var/bohrium/object
tmp_src_dir = /usr/var/bohrium/source
dump_src = true
verbose = false
prof = false #Profiling statistics
compiler_cmd = "/usr/bin/x86_64-linux-gnu-gcc"
compiler_inc = "-I/usr/share/bohrium/include"
compiler_lib = "-lm -L/usr/lib -lbh"
compiler_flg = "-x c -fPIC -shared  -std=gnu99  -O3 -march=native -Werror -fopenmp"
compiler_openmp = true
compiler_openmp_simd = false
```

(continues on next page)

(continued from previous page)

```
[opencl]
impl = /usr/lib/libbh_ve_opencl.so
verbose = false
prof = false #Profiling statistics
# Additional options given to the opencl compiler. See documentation for␣
↪clBuildProgram
compiler_flg = "-I/usr/share/bohrium/include"
serial_fusion = false # Topological fusion is default
```

The configuration file consists of two things: `components` and orchestration of components in `stacks`.

Components marked with square brackets. For example `[node]`, `[openmp]`, `[opencl]` are all components available for the runtime system.

The `stacks` define different default configurations of the runtime environment and one can switch between them using the environment var `BH_STACK`.

The configuration of a component can be overwritten with environment variables using the naming convention `BH_[COMPONENT]_[OPTION]`, below are a couple of examples controlling the behavior of the CPU vector engine:

```
BH_OPENMP_PROF=true    -- Prints a performance profile at the end of execution.
BH_OPENMP_VERBOSE=true -- Prints a lot of information including the source of the JIT␣
↪compiled kernels. Enables per-kernel profiling when used together with BH_OPENMP_
↪PROF=true.
```

Useful environment variables:

```
BH_SYNC_WARN=true       -- Show Python warnings in all instances when copying data to␣
↪Python.
BH_MEM_WARN=true        -- Show warnings when memory accesses are problematic.
BH_<backend>_GRAPH=true -- Dump a dependency graph of the instructions send to the␣
↪back-ends (.dot file).
BH_<backend>_VOLATILE=true -- Declare temporary variables using `volatile`, which␣
↪avoid precision differences because of Intel's use of 80-bit floats internally.
```

2.3 Developer Guide

Bohrium is hosted and made publicly available via a git-repository on github under the *LGPLv3 License*.

If you want to join / contribute then fork the repository on Github and get in touch with us.

If you just want read-access then simply clone the repository:

```
git clone git@github.com/bh107/bohrium.git
cd bohrium
```

Continue by taking a look at *Installation* on how to build / install Bohrium.

2.3.1 Further information

Tools

Valgrind, GDB, and Python

Valgrind is a great tool for memory debugging, memory leak detection, and profiling. However, both Python and NumPy floods the valgrind output with memory errors - it is therefore necessary to use a debug and valgrind friendly version of Python and NumPy:

```
sudo apt-get build-dep python
sudo apt-get install zlib1g-dev valgrind

mkdir python_debug_env
cd python_debug_env
export INSTALL_DIR=$PWD

# Build and install Python:
export VERSION=2.7.11
wget http://www.python.org/ftp/python/$VERSION/Python-$VERSION.tgz
tar -xzf Python-$VERSION.tgz
cd Python-$VERSION
./configure --with-pydebug --without-pymalloc --with-valgrind --prefix=$INSTALL_DIR
make install
sudo ln -s $PWD/python-gdb.py /usr/bin/python-gdb.py
sudo ln -s $INSTALL_DIR/bin/python /usr/bin/dython
cd ..
rm Python-$VERSION.tgz

# Build and install Cython
export VERSION=0.24
wget http://cython.org/release/Cython-$VERSION.tar.gz
tar -xzf Cython-$VERSION.tar.gz
cd Cython-$VERSION
dython setup.py install
cd ..
rm Cython-$VERSION.tar.gz

export VERSION=21.1.0
wget https://pypi.python.org/packages/f0/32/
↪99ead2d74cba43bd59aa213e9c6e8212a9d3ed07805bb66b8bf9affbb541/setuptools-$VERSION.
↪tar.gz#md5=8fd8bdbf05c286063e1052be20a5bd98
tar -xzf setuptools-$VERSION.tar.gz
cd setuptools-$VERSION
dython setup.py install
cd ..
rm setuptools-$VERSION.tar.gz

# Build and install NumPy
export VERSION=1.11.0
wget  https://github.com/numpy/numpy/archive/v$VERSION.tar.gz
tar -xzf v$VERSION.tar.gz
cd numpy-$VERSION
dython setup.py install
cd ..
rm v$VERSION.tar.gz
```

Build Bohrium with custom Python

Build and install Bohrium (with some components deactivated):

```
unzip master.zip
cd bohrium-master
mkdir build
cd build
cmake .. -DPYTHON_EXECUTABLE=/usr/bin/dython -DEXT_FFTW=OFF -DEXT_VISUALIZER=OFF -
↪DVEM_PROXY=OFF -DVE_GPU=OFF  -DBRIDGE_NUMCIL=OFF -DTEST_CIL=OFF
make
make install
cd ..
rm master.zip
```

Most Used Commands

GDB

GDB supports some helpful Python commands (https://docs.python.org/devguide/gdb.html). To activate, `source` the `python-gdb.py` file within GDB:

```
source /usr/bin/python-gdb.py
```

Then you can use Python specific GDB commands such as `py-list` or `py-bt`.

Valgrind

Valgrind can be used to detect memory errors by invoking it with:

```
valgrind --suppressions=<path to bohrium>/misc/valgrind.supp dython <SCRIPT_NAME>
```

Narrowing the valgrind analysis, add the following to your source code:

```
#include <valgrind/callgrind.h>
... your code ...
CALLGRIND_START_INSTRUMENTATION;
... your code ...
CALLGRIND_STOP_INSTRUMENTATION;
CALLGRIND_DUMP_STATS;
```

Then run valgrind with the flag:

```
--instr-atstart=no
```

Invoking valgrind to determine cache-utilization:

```
--tool=callgrind --simulate-cache=yes <PROG> <PROG_PARAM>
```

Cluster VEM (MPI)

In order to use MPI with valgrind, the MPI implementation needs to be compiled with PIC and no-dlopen flag. E.g, OpenMPI could be installed as follows:

```
wget http://www.open-mpi.org/software/ompi/v1.6/downloads/openmpi-1.6.5.tar.gz
cd tar -xzf openmpi-1.6.5.tar.gz
cd openmpi-1.6.5
./configure --with-pic --disable-dlopen --prefix=/opt/openmpi
make
sudo make install
```

And then executed using valgrind:

```
export LD_LIBRARY_PATH=/opt/openmpi/lib/:$LD_LIBRARY_PATH
export PATH=/opt/openmpi/bin:$PATH
mpiexec -np 1 valgrind dython test/numpy/numpytest.py : -np 1 valgrind ~/.local/bh_
↪vem_cluster_slave
```

Writing Documentation

The documentation is written in Sphinx.

You will need the following to write/build the documentation:

```
sudo apt-get install doxygen python-sphinx python-docutils python-setuptools
```

As well as a python-packages **breathe** and **numpydoc** for integrating doxygen-docs with Sphinx:

```
sudo easy_install breathe numpydoc
```

Overview of the documentation files:

```
bohrium/doc                 # Root folder of the documentation.
bohrium/doc/source          # Write / Edit the documentation here.
bohrium/doc/build           # Documentation is "rendered" and stored here.
bohrium/doc/Makefile        # This file instructs Sphinx on how to "render" the␣
↪documentation.
bohrium/doc/make.bat        # ---- || ----, on Windows
bohrium/doc/deploy_doc.sh   # This script pushes the rendered docs to http://bohrium.
↪bitbucket.org.
```

Most used commands

These commands assume that your current working dir is **bohrium/doc**.

Initiate doxygen:

```
make doxy
```

Render a html version of the docs:

```
make html
```

Push the html-rendered docs to http://bohrium.bitbucket.org, this command assumes that you have write-access to the doc-repos on Bitbucket:

```
make deploy
```

The docs still needs a neat way to integrate a full API-documentation of the Bohrium core, managers and engines.

Continuous Integration

Currently we use both a privately hosted Jenkins server as well as Travis for our CI.

Setup jenkins:

```
wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > /etc/apt/sources.list.
↪d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins
```

Then configure it via the web interface.

- Open Student Projects
- Benchmark Suite

2.4 Frequently Asked Questions (FAQ)

Does it automatically support lazy evaluation (also called: late evaluation, expression templates)?

Yes, Bohrium will lazy evaluate all Python/NumPy operations until it encounters a "Python Read", such a printing an array or having an if-statement testing the value of an array.

Does it support "views" in the sense that a sub-slice is simply a view on the same array?

Yes, Bohrium supports NumPy views fully thus operating on array slices does not involve data copying.

Does it support generator functions (which only start calculating once the evaluation is forced)? Which ones are supported? Which conditions force evaluations? Presumably reduce operations?

Yes, Bohrium uses a fusion algorithm that fuses (or merges) array operations into the same computation kernel that are then JIT-compiled and executed. However, Bohrium can only fuse operations that have some common sized dimension and no horizontal data conflicts. Typically, reducing a vector to a scalar will force evaluate (but reducing a matrix to a vector will not force an evaluate on it own).

On GPUs, will Bohrium automatically keep all data (i.e. all Bohrium arrays) on the card?

Yes, we only move data back to the host when the data is accessed directly by Python or a Python C-extension.

Does it fully support operations on the complex datatype in Bohrium arrays?

Yes.

Will it lazily operate even over for-loops effectively unrolling them?

Yes, a for-loop in Python does not force evaluation. However, loops in Python with many iterations will hurt performance, just like it does in regular NumPy or Matlab

Is Bohrium using CUDA on Nvidia Cards or generic OpenCL for any GPU?

At the moment, Bohrium uses OpenCL for both Nvidia, AMD, and Intel graphic cards.

What is the disadvantage of Bohrium? I wonder why it exists as a separate project. From my point of view it looks like Bohrium is "just reimplementing" NumPy. That's probably extremely oversimplified, but is there a plan to feed the results of Bohrium into the NumPy project?

The only disadvantage of Bohrium is the extra dependencies e.g. Bohrium need a C99 compiler for JIT-complication. Thus, the idea of incorporating Bohrium into NumPy as an alternative "backend" is very appealing and we hope it could be realized some day.

2.5 Reporting Bugs

Please help us make Bohrium even better by submitting bugs and/or feature requests to us via the issue tracker on https://github.com/bh107/bohrium/issues

When reporting problems please include the output from:

```
python -m bohrium --info
```

2.6 Publications

1. Mads R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In Python for High Performance and Scientific Computing (PyHPC 2013), 2013.
2. Simon A. F. Lund, Kenneth Skovhede, Mads R. B. Kristensen, and Brian Vinter. Doubling the Performance of Python/NumPy with less than 100 SLOC. In Python for High Performance and Scientific Computing (PyHPC 2013), 2013.
3. Troels Blum, Mads R. B. Kristensen, and Brian Vinter. Transparent gpu execution of numpy applications.. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International. IEEE, 2014.
4. Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: a virtual machine approach to portable parallelism. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International. IEEE, 2014.
5. Simon A.F. Lund, Mads R.B. Kristensen, Brian Vinter, Dimitrios Katsaros. Bypassing the Conventional Software Stack Using Adaptable Runtime Systems. In Proceedings of the Euro-Par Workshops, 2014.
6. Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, James Avery, and Brian Vinter. Separating NumPy API from Implementation. In Proceedings of the Python for High Performance and Scientific Computing (PyHPC 2014), 2014.
7. Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, and James Avery. Fusion of Parallel Array Operations. In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT'16), 2016.
8. Mads R.B. Kristensen, Simon A.F. Lund, Troels Blum, James Avery, and Brian Vinter. Battling Memory Requirements of Array Programming through Streaming. In Proceedings of the International Conference on High Performance Computing, 2016.

2.7 History and License

Bohrium is an active research project started by Mads R. B. Kristensen, Troels Blum, and Brian Vinter at the Niels Bohr Institute - University of Copenhagen. Contributors include those listed below in no particular order:

- Troels Blum <blum@nbi.dk>
- Brian Vinter <vinter@nbi.dk>
- Kenneth Skovhede <skovhede@nbi.dk>
- Simon Andreas Frimann Lund <safl@nbi.dk>
- Mads Ruben Burgdorff Kristensen <madsbk@nbi.dk>

- Mads Ohm Larsen <ohm@nbi.dk>

Contributors are welcome, do not hesitate to contact us!

Bohrium is distributed under the LGPLv3 license:

```
                   GNU LESSER GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.


This version of the GNU Lesser General Public License incorporates
the terms and conditions of version 3 of the GNU General Public
License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser
General Public License, and the "GNU GPL" refers to version 3 of the GNU
General Public License.

"The Library" refers to a covered work governed by this License,
other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided
by the Library, but which is not otherwise based on the Library.
Defining a subclass of a class defined by the Library is deemed a mode
of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an
Application with the Library.  The particular version of the Library
with which the Combined Work was made is also called the "Linked
Version".

The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the
object code and/or source code for the Application, including any data
and utility programs needed for reproducing the Combined Work from the
Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License
without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
facility is invoked), then you may convey a copy of the modified
version:
```

(continues on next page)

(continued from previous page)

```
 a) under this License, provided that you make a good faith effort to
 ensure that, in the event an Application does not supply the
 function or data, the facility still operates, and performs
 whatever part of its purpose remains meaningful, or

 b) under the GNU GPL, with none of the additional permissions of
 this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from
a header file that is part of the Library.  You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

 a) Give prominent notice with each copy of the object code that the
 Library is used in it and that the Library and its use are
 covered by this License.

 b) Accompany the object code with a copy of the GNU GPL and this license
 document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
the following:

 a) Give prominent notice with each copy of the Combined Work that
 the Library is used in it and that the Library and its use are
 covered by this License.

 b) Accompany the Combined Work with a copy of the GNU GPL and this license
 document.

 c) For a Combined Work that displays copyright notices during
 execution, include the copyright notice for the Library among
 these notices, as well as a reference directing the user to the
 copies of the GNU GPL and this license document.

 d) Do one of the following:

     0) Convey the Minimal Corresponding Source under the terms of this
     License, and the Corresponding Application Code in a form
     suitable for, and under terms that permit, the user to
     recombine or relink the Application with a modified version of
     the Linked Version to produce a modified Combined Work, in the
     manner specified by section 6 of the GNU GPL for conveying
     Corresponding Source.

     1) Use a suitable shared library mechanism for linking with the
     Library.  A suitable mechanism is one that (a) uses at run time
```

(continues on next page)

(continued from previous page)

```
     a copy of the Library already present on the user's computer
     system, and (b) will operate properly with a modified version
     of the Library that is interface-compatible with the Linked
     Version.

  e) Provide Installation Information, but only if you would otherwise
  be required to provide such information under section 6 of the
  GNU GPL, and only to the extent that such information is
  necessary to install and execute a modified version of the
  Combined Work produced by recombining or relinking the
  Application with a modified version of the Linked Version. (If
  you use option 4d0, the Installation Information must accompany
  the Minimal Corresponding Source and Corresponding Application
  Code. If you use option 4d1, you must provide the Installation
  Information in the manner specified by section 6 of the GNU GPL
  for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the
Library side by side in a single library together with other library
facilities that are not Applications and are not covered by this
License, and convey such a combined library under terms of your
choice, if you do both of the following:

  a) Accompany the combined library with a copy of the same work based
  on the Library, uncombined with any other library facilities,
  conveyed under the terms of this License.

  b) Give prominent notice with the combined library that part of it
  is a work based on the Library, and explaining where to find the
  accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions
of the GNU Lesser General Public License from time to time. Such new
versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the
Library as you received it specifies that a certain numbered version
of the GNU Lesser General Public License "or any later version"
applies to it, you have the option of following the terms and
conditions either of that published version or of any later version
published by the Free Software Foundation. If the Library as you
received it does not specify a version number of the GNU Lesser
General Public License, you may choose any version of the GNU Lesser
General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide
whether future versions of the GNU Lesser General Public License shall
apply, that proxy's public statement of acceptance of any version is
permanent authorization for you to choose that version for the
Library.
```