Separating NumPy API from Implementation

Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and Kenneth Skovhede
Niels Bohr Institute, University of Copenhagen, Denmark
{madsbk/safl/blum/skovhede}@nbi.dk

Abstract—**In this paper, we introduce a unified backend framework for NumPy that combine a broad range of Python code accelerators with no modifications to the user Python/NumPy application. Thus, a Python/NumPy application can utilize hardware architecture such as multi-core CPUs and GPUs and optimization techniques such as Just-In-Time compilation and loop fusion without any modifications. The backend framework defines a number of primitive functions, including all existing ufuncs in NumPy, that a specific backend must implement in order to accelerate a Python/NumPy application. The framework then seamlessly translates the Python/NumPy application into a stream of calls to these primitive functions.**

In order to demonstrate the usability of our unified backend framework, we implement and benchmark four different backend implementations that use four different Python libraries: NumPy, Numexpr, libgpuarray, and Bohrium. The results are very promising with a speedup of up to 18 compared to a pure NumPy execution.

I. Introduction

Python is a high-level, general-purpose, interpreted language. Python advocates high-level abstractions and convenient language constructs for readability and productivity rather than high-performance. However, Python is easily extensible with libraries implemented in high-performance languages such as C and FORTRAN, which makes Python a great tool for gluing high-performance libraries together[1]. NumPy is the de-facto standard for scientific applications written in Python[2] and contributes to the popularity of Python in the HPC community. NumPy provides a rich set of high-level numerical operations and introduces a powerful array object. The array object is essential for scientific libraries, such as SciPy[3] and matplotlib[4], and a broad range of Python wrappers of external scientific libraries[5], [6], [7]. NumPy supports a declarative vector programming style where numerical operations applies to full arrays rather than scalars. This programming style is often referred to as vector or array programming and is commonly used in programming languages and libraries that target the scientific community, e.g. HPF[8], ZPL[9], MATLAB[10], Armadillo[11], and Blitz++[12].

NumPy does not make Python a high-performance language but through array programming it is possible to achieve performance within one order of magnitude of C. In contrast to pure Python, which typically is more than hundred if not thousand times slower than C. However, NumPy does not utilize parallel computer architectures when implementing basic array operations; thus only through external libraries, such as BLAS or FFTW, is it possible to utilize data or task parallelism.

In this paper, we introduce a unified NumPy backend that enables seamless utilization of parallel computer architecture such as multi-core CPUs, GPUs, and Clusters. The framework exposes NumPy applications as a stream of abstract array operations that architecture-specific computation backends can execute in parallel without the need for modifying the original NumPy application.

The aim of this new unified NumPy backend is to provide support for a broad range of computation architectures with minimal or no changes to existing NumPy applications. Furthermore, we insist on legacy support (at least back to version 1.6 of NumPy), thus we will not require any changes to the NumPy source code itself.

II. Related Work

Numerous projects strive to accelerate Python/NumPy applications through very different approaches. In order to utilize the performance of existing programming languages, projects such as Cython[13], IronPython[14], and Jython[15], introduce static source-to-source compilation to C, .NET, and Java, respectively. However, none of the projects are seamlessly compatible with Python – Cython extends Python with static type declarations whereas IronPython and Jython do not support third-party libraries such as NumPy.

PyPy[16] is a Python interpreter that makes use of Just-in-Time (JIT) compilation in order to improve performance. PyPy is also almost Python compliant, but again PyPy does not support libraries such as NumPy fully and, similar to IronPython and Jython, it is not possible to fall back to the original Python interpreter CPython when encountering unsupported Python code.

Alternatively, projects such as Weave[17], Numexpr[18], and Numba[19] make use of JIT compilation to accelerate parts of the Python application. Common for all of them is the introduction of functions or decorators that allow the user to specify acceleratable code regions.

In order to utilize GPGPUs the PyOpenCL and PyCUDA projects enable the user to write GPU kernels directly in Python[20]. The user writes OpenCL[21] or CUDA[22] specific kernels as text strings in Python, which simplifies the utilization of OpenCL or CUDA compatible GPUs but still requires OpenCL or CUDA programming knowledge. Less intrusively, libgpuarray, which is part of the Theano[23] project, introduces GPU arrays on which all operations execute on the GPU. The GPU arrays are similar to NumPy arrays but are not a drop-in replacement.

III. The Interface

The interface of our unified NumPy backend (npbackend) consists of two parts: a user interface that facilitates the end NumPy user and a backend interface that facilitates the

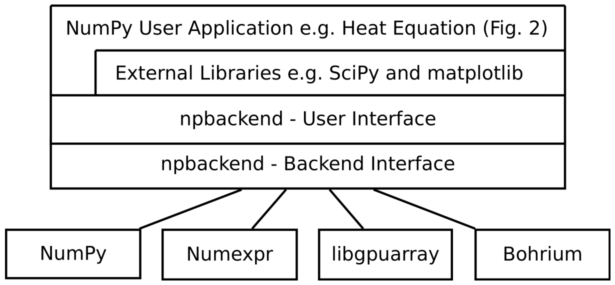
NumPy User Application e.g. Heat Equation (Fig. 2)
External Libraries e.g. SciPy and matplotlib
npbackend - User Interface
npbackend - Backend Interface
NumPy
Numexpr
libgpuarray
Bohrium

Fig. 1: The Software Stack

backend writers (Fig. 1). The source code of both interfaces and all backend implementations is an available at the Bohrium project's website[1] for further inspection. In the following two subsections, we present the two interfaces.

A. *The User Interface*

The main design objective of the user interface is easy transition from regular NumPy code to code that utilizes a unified NumPy backend. Ideally, there should be no difference between NumPy code with or without a unified NumPy backend. Through modifications of the NumPy source code, the DistNumPy[24] and Bohrium[25] projects demonstrate that it is possible to implement an alternative computation backend that does not require any changes to the user's NumPy code. However, it is problematic to maintain a parallel version of NumPy that contains complex modifications to numerous parts of the project, particularly when we have to fit each modification to a specific version of NumPy (version 1.6 through 1.9).

As a consequence, instead of modifying NumPy, we introduce a new Python module *npbackend* that implements an array object that inherit from NumPy's ndarray. The idea is that this new npbackend-array can be a drop-in replacement of the numpy-array such that only the array object in NumPy applications needs to be changed. Similarly, the npbackend module is a drop-in replacement of the NumPy module.

The user can make use of npbackend through an explicit and an implicit approach. The user can explicitly import npbackend instead of NumPy in the source code e.g. "`import npbackend as numpy`" or the user can alias NumPy imports with npbackend imports globally through the `-m` interpreter argument e.g. "`python -m npbackend user_app.py`".

Even though the npbackend is a drop-in replacement, the backend might not implement all of the NumPy API, in which case npbackend will gracefully use the original NumPy implementation. Since npbackend-array inherits from numpy-array, the original NumPy implementation can access and apply operations on the npbackend-array seamlessly. The result is that a NumPy application can utilize an architecture-specific backend with minimal or no modification. However, npbackend does not guarantee that all operations in the application will utilize the backend — only the ones that the backend support.

[1]http://bh107.org

```
 1 import npbackend as np
 2 import matplotlib.pyplot as plt
 3
 4 def solve(height, width, epsilon=0.005):
 5   grid = np.zeros((height+2,width+2),dtype=np.float64)
 6   grid[:,0]  = -273.15
 7   grid[:,-1] = -273.15
 8   grid[-1,:] = -273.15
 9   grid[0,:]  = 40.0
10   center = grid[1:-1,1:-1]
11   north  = grid[:-2,1:-1]
12   south  = grid[2:,1:-1]
13   east   = grid[1:-1,:-2]
14   west   = grid[1:-1,2:]
15   delta  = epsilon+1
16   while delta > epsilon:
17     tmp = 0.2*(center+north+south+east+west)
18     delta = np.sum(np.abs(tmp-center))
19     center[:] = tmp
20   plt.matshow(center, cmap='hot')
21   plt.show()
```

Fig. 2: Python implementation of a heat equation solve that uses the finite-difference method to calculate the heat diffusion. Note that we could replace the first line of code with "`import numpy as np`" and still utilize npbackend through the command line argument "`-m`", e.g. "`python -m npbackend heat2d.py`"

Figure 2, is an implementation of a heat equation solver that imports the npbackend module explicitly at the first line and a popular visualization module, Matplotlib, at the second line. At line 5, the function `zeros()` creates a new npbackend-array that overloads the arithmetic operators, such as `*` and `+`. Thus, at line 17 the operators use npbackend rather than NumPy. However, in order to visualize (Fig. 3) the `center` array at line 20, Matplotlib accesses the memory of `center` directly.

Now, in order to explain what we mean by *directly*, we have to describe some implementation details of NumPy. A NumPy ndarray is a C implementation of a Python class that exposes a segment of main memory through both a C and a Python interface. The ndarray contains metadata that describes how the memory segment is to be interpreted as a multi-dimensional array. However, only the Python interface seamlessly interprets the ndarray as a multi-dimensional array. The C interface provides a C-pointer to the memory segment and lets the user handle the interpretation. Thus, with the word *directly* we mean that Matplotlib accesses the memory segment of `center` through the C-pointer. In which case, the only option for npbackend is to make sure that the computed values of `center` are located at the correct memory segment. Npbackend is oblivious to the actual operations Matplotlib performs on `center`.

Consequently, the result of the Matplotlib call is a Python warning explaining that npbackend will not accelerate the operation on `center` at line 20; instead the Matplotlib implementation will handle the operation exclusively.

B. *The Backend Interface*

The main design objective of the backend interface is to isolate the calculation-specific from the implementation-specific. In order to accomplish this, we translate a NumPy execution into a sequence of primitive function calls, which the backend must implement.

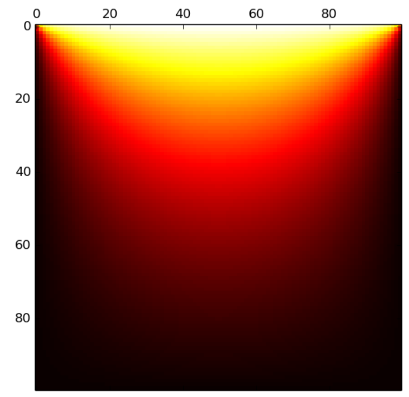

Fig. 3: The *matplotlib* result of executing the heat equation solver from figure 2: `solve(100,100)`

.

Figure 4 is the abstract Python module that a npbackend must implement. It consists of two Python classes, `base` and `view`, that represent a memory sequence and a multi-dimensional array-view thereof. Since this is the abstract Python module, the base class does not refer to any physical memory but only a size and a data type. In order to implement a backend, the base class could, for example, refer to the main memory or GPU memory. Besides the two classes, the backend must implement eight primitive functions. Seven of the functions are self-explanatory (Fig. 4), however the `extmethod()` function requires some explanation. In order to support arbitrary NumPy operations, npbackend introduces an Extension Method that passes any operations through to the backend. For example, it is not convenient to implement operations such as matrix multiplication or FFT only using ufuncs; thus we define an Extension Method called *matmul* that corresponds to a matrix multiplication. Now, if a backend knows the *matmul* operation it should perform a matrix multiplication. On the other hand, if the backend does not know *matmul* it must raise a `NotImplementedError` exception.

IV. The Implementation

The implementation of npbackend consists primarily of the new npbackend-array that inherits from NumPy's numpy-array. The npbackend-array is implemented in C and uses the Python-C interface to inherit from numpy-array. Thus, it is possible to replace npbackend-array with numpy-array both in C and in Python — a feature npbackend must support in order to support code such as the heat equation solver in figure 2.

As is typical in object-oriented programming, the npbackend-array exploits the functionality of numpy-array as much as possible. The original numpy-array implementation handles metadata manipulation, such as slicing and transposing; only the actual array calculations will be handled by the npbackend. The npbackend-array overloads arithmetic operators thus an operator on npbackend-arrays will call the backend function `ufunc` (Fig. 4 Line 26). Furthermore, since npbackend-arrays inherit from numpy-array, an operator on a mix of the two array classes will also use the backend function.

However, NumPy functions in general will not make use

```
1  """Abstract module for computation backends"""
2
3  class base(object):
4    """Abstract base array handle (an array has only one ↩
          base array)"""
5    def __init__(self, size, dtype):
6      self.size = size #Total number of elements
7      self.dtype = dtype #Data type
8
9  class view(object):
10   """Abstract array view handle"""
11   def __init__(self, ndim, start, shape, stride, base):
12     self.ndim = ndim #Number of dimensions
13     self.shape = shape #Tuple of dimension sizes
14     self.base = base #The base array this view refers to
15     self.start = start*base.dtype.itemsize #Offset from ↩
            base (in bytes)
16     self.stride = [x*base.dtype.itemsize for x in stride] ↩
            #Tuple of strides (in bytes)
17
18 def get_data_pointer(ary, allocate=False, nullify=False):
19   """Return a C-pointer to the array data (as a Python ↩
          integer)"""
20   raise NotImplementedError()
21
22 def set_data_from_ary(self, ary):
23   """Copy data from 'ary' into the array 'self'"""
24   raise NotImplementedError()
25
26 def ufunc(op, *args):
27   """Perform the ufunc 'op' on the 'args' arrays"""
28   raise NotImplementedError()
29
30 def reduce(op, out, a, axis):
31   """Reduce 'axis' dimension of 'a' and write the result ↩
          to out"""
32   raise NotImplementedError()
33
34 def accumulate(op, out, a, axis):
35   """Accumulate 'axis' dimension of 'a' and write the ↩
          result to out"""
36   raise NotImplementedError()
37
38 def extmethod(name, out, in1, in2):
39   """Apply the extended method 'name'"""
40   raise NotImplementedError()
41
42 def range(size, dtype):
43   """Create a new array containing the values [0:size["""
44   raise NotImplementedError()
45
46 def random(size, seed):
47   """Create a new random array"""
48   raise NotImplementedError()
```

Fig. 4: The backend interface of npbackend.

of the npbackend backend since many of them uses the C-interface to access the array memory directly. In order to address this problem, npbackend has to re-implement much of the NumPy API, which is a lot of work and is prone to error. However, we can leverage the work by the PyPy project; PyPy does not support the NumPy C-interface either but they have re-implemented much of the NumPy API already. Still, the problem goes beyond NumPy; any library that makes use of the NumPy C-interface will have to be rewritten.

The result is that the npbackend implements all array creation functions, matrix multiplication, random, FFT, and all ufuncs for now. All other functions that access array memory directly will simply get unrestricted access to the memory.

A. *Unrestricted Direct Memory Access*

In order to detect and handle direct memory access to arrays, npbackend uses two address spaces for each array memory: a user address space visible to the user interface and a backend address space visible to the backend interface. Initially, the user address space of a new array is memory protected with `mprotect` such that subsequent accesses to the memory will trigger a segmentation fault. In order to detect and handle direct memory access, npbackend can then handle this kernel signal by transferring array memory from the backend address space to the user address space. In order to get access to the backend address space memory, npbackend calls the `get_data_pointer()` function (Fig. 4, Line 18). Similarly, npbackend calls the `set_data_from_ary()` function (Fig. 4, Line 22) when the npbackend should handle the array again.

In order to make the transfer between the two address spaces, we use `mremap` rather than the more expensive `memcpy`. However, `mremap` requires that the source and destination are memory page aligned. That is not a problem at the backend since the backend implementer can simply use `mmap` when allocating memory; on the other hand, we cannot change how NumPy allocates its memory at the user address space. The solution is to re-allocate the array memory when the constructor of npbackend-array is called using `mmap`. This introduces extra overhead but will work in all cases with no modifications to the NumPy source code.

V. Backend Examples

In order to demonstrate the usability of npbackend, we implement four backends that use four different Python libraries: NumPy, Numexpr, libgpuarray, and Bohrium, all of whom are standalone Python libraries in their own right. In this section, we will describe how the four backends implement the eight functions that make up the backend interface (Fig. 4).

A. *NumPy Backend*

In order to explore the overhead of npbackend, we implement a backend that uses NumPy i.e. NumPy uses NumPy through npbackend. Figure 5 is a code snippet of the implementation that includes the `base` and `view` classes, which inherit from the abstract classes in figure 4, the three essential functions `get_data_pointer()`, `set_data_from_ary()`, and `ufunc()`, and the Extension Method function `extmethod()`.

The NumPy backend associates a NumPy view (`.ndarray`) with each instance of the `view` class and an `mmap` object for each `base` instance, which enables memory allocation reuse and guarantees memory-page-aligned allocations. In [26] the authors demonstrate performance improvement through memory allocation reuse in NumPy. The NumPy backend uses a similar technique[2] where it preserves a pool of memory allocations for recycling. The constructor of `base` will check this memory pool and, if the size matches, reuse the memory allocation (line 11-15).

The `get_data_pointer()` function simply returns a C-pointer to the ndarray data. The `set_data_from_ary()` function `memmoves` the data from the ndarray *ary* to the `view` *self*. The `ufunc()` function simply calls the NumPy library with the corresponding ufunc. Finally, the `extmethod()`

```
1  import numpy
2  import backend
3  import os
4
5  VCACHE_SIZE = int(os.environ.get("VCACHE_SIZE", 10))
6  vcache = []
7  class base(backend.base):
8    def __init__(self, size, dtype):
9      super(base, self).__init__(size, dtype)
10     size *= dtype.itemsize
11     for i, (s,m) in enumerate(vcache):
12       if s == size:
13         self.mmap = m
14         vcache.pop(i)
15         return
16     self.mmap = mmap.mmap(-1, size)
17   def __str__(self):
18     return "<base memory at %s>"%self.mmap
19   def __del__(self):
20     if len(vcache) < VCACHE_SIZE:
21       vcache.append((self.size*self.dtype.itemsize, ↩
            self.mmap))
22
23 class view(backend.view):
24   def __init__(self, ndim, start, shape, stride, base):
25     super(view, self).__init__(ndim, start, shape, stride,↩
           base)
26     buf = np.frombuffer(self.base.mmap, dtype=self.dtype, ↩
           offset=self.start)
27     self.ndarray = np.lib.stride_tricks.as_strided(buf, ↩
           shape, self.stride)
28
29 def get_data_pointer(ary, allocate=False, nullify=False):
30   return ary.ndarray.ctypes.data
31
32 def set_data_from_ary(self, ary):
33   d = get_data_pointer(self, allocate=True, nullify=False)
34   ctypes.memmove(d, ary.ctypes.data, ary.dtype.itemsize * ↩
         ary.size)
35
36 def ufunc(op, *args):
37   args = [a.ndarray for a in args]
38   f = eval("numpy.%s"%op)
39   f(*args[1:], out=args[0])
40
41 def extmethod(name, out, in1, in2):
42   (out, in1, in2) = (out.ndarray, in1.ndarray, in2.ndarray↩
       )
43   if name == "matmul":
44     out[:] = np.dot(in1, in2)
45   else:
46     raise NotImplementedError()
```

Fig. 5: A code snippet of the NumPy backend. Note that the `backend` module refers to the implementation in figure 4.

recognizes the *matmul* method and calls NumPy's `dot()` function.

B. *Numexpr Backend*

In order to utilize multi-core CPUs, we implement a backend that uses the Numexpr library, which in turn utilize Just-In-Time (JIT) compilation and shared-memory parallelization through OpenMP.

Since Numexpr is compatible with NumPy `ndarrays`, the Numexpr backend can inherit most functionality from the NumPy backend; only the `ufunc()` implementation differs. Figure 6 is a code snippet that includes the `ufunc()` implementation where it uses `numexpr.evaluate()` to evaluate a ufunc operation. Now, this is a very naïve implementation since we only evaluate one operation at a time. In order to maximize performance of Numexpr, we could collect as many ufunc operations as possible into one `evaluate()`

[2]Using a victim cache

```
ufunc_cmds = {'add'      : "i1+i2",
              'multiply' : "i1*i2",
              'sqrt'     : "sqrt(i1)",
              #...
              }

def ufunc(op, *args):
  args = [a.ndarray for a in args]
  i1=args[1];
  if len(args) > 2:
    i2=args[2]
  numexpr.evaluate(ufunc_cmds[op], \
                   out=args[0], casting='unsafe')
```

Fig. 6: A code snippet of the Numexpr backend.

```
 1 import pygpu
 2 import backend_numpy
 3 class base(backend_numpy.base):
 4   def __init__(self, size, dtype):
 5     self.clary = pygpu.empty((size,), dtype=dtype, cls=↩
          elemary)
 6     super(base, self).__init__(size, dtype)
 7
 8 class view(backend_numpy.view):
 9   def __init__(self, ndim, start, shape, stride, base):
10     super(view, self).__init__(ndim, start, shape, stride,↩
           base)
11     self.clary = pygpu.gpuarray.from_gpudata(base.clary.↩
          gpudata, offset=self.start, dtype=base.dtype, ↩
          shape=shape, strides=self.stride, writable=True, ↩
          base=base.clary, cls=elemary)
12
13 def get_data_pointer(ary, allocate=False, nullify=False):
14   ary.ndarray[:] = np.asarray(ary.clary)
15   return ary.ndarray.ctypes.data
16
17 def set_bhc_data_from_ary(self, ary):
18   self.clary[:] = pygpu.asarray(ary)
19
20 def ufunc(op, *args):
21   args = [a.ndarray for a in args]
22   out=args[0]
23   i1=args[1];
24   if len(args) > 2:
25     i2=args[2]
26   cmd = "out[:] = %s"%ufunc_cmds[op]
27   exec cmd
28
29 def extmethod(name, out, in1, in2):
30   (out, in1, in2) = (out.clary, in1.clary, in2.clary)
31   if name == "matmul":
32     pygpu.blas.gemm(1, in1, in2, 1, out, overwrite_c=True)
33   else:
34     raise NotImplementedError()
```

Fig. 7: A code snippet of the ligpuarray backend (the Python binding module is called `pygpu`). Note that the `backend_numpy` module refers to the implementation in figure 5 and note that `ufunc_cmds` is from figure 6.

call, which would enable Numexpr to fuse multiple ufunc operations together into one JIT compiled computation kernel. However, such work is beyond the focus of this paper – in this paper we map the libraries directly.

C. Libgpuarray Backend

In order to utilize GPUs, we implement a backend that makes use of libgpuarray, which introduces a GPU-array that is compatible with NumPy's ndarray. For the two classes, `base` and `view`, we associate a GPU-array that points to memory on the GPU; thus the user address space lies in main memory and the backend address space lies in GPU-memory. Consequently, the implementation of the two functions `get_data_pointer()` and `set_data_from_ary()` uses `asarray()` to copy between main memory and GPU-memory (Fig. 7 Line 14 and 15). The implementation of `ufunc()` is very similar to the Numexpr backend implementation since GPU-arrays supports ufunc directly. However, note that libgpuarray does not support the output argument, which means we have to copy the result of an ufunc operation into the output argument.

The `extmethod()` recognizes the *matmul* method and calls Libgpuarray's `blas.gemm()` function.

Processor:	Intel Xeon E5640
Clock:	2.66 GHz
L3 Cache:	12MB
Memory:	96GB DDR3
GPU:	Nvidia GeForce GTX 460
GPU-Memory:	1GB DDR5
Compiler:	GCC 4.8.2 & OpenCL 1.2
Software:	Linux 3.13, Python 2.7, & NumPy 1.8.1

TABLE I: The Machine Specification

D. Bohrium Backend

Our last backend implementation uses the Bohrium runtime system to utilize both CPU and GPU architectures. Bohrium supports a range of frontend languages including C, C++, and CIL[3], and a range of backend architectures including multi-core CPUs through OpenMP and GPUs through OpenCL. The Bohrium runtime system utilizes the underlying architectures seamlessly. Thus, as a user we use the same interface whether we utilize a CPU or a GPU. The interface of Bohrium is very similar to NumPy – it consists of a multidimensional array and the same ufuncs as in NumPy.

The Bohrium backend implementation uses the C interface of Bohrium, which it calls directly from Python through SWIG[27]. The two `base` and `view` classes points to a Bohrium multidimensional array called `.bhc_obj` (Fig. 8). In order to use the Bohrium C interface through SWIG, we dynamically construct a Python string that matches a specific C function in the Bohrium C interface.

The `set_bhc_data_from_ary()` function is identical to the one in the NumPy backend. However, `get_data_pointer()` needs to synchronize the array data before returning a Python pointer to the data. This is because the Bohrium runtime system uses lazy evaluation in order to fuse multiple operations into single kernels. The synchronize function (Fig. 8 Line 34) makes sure that all pending operations on the array have been executed and that the array data is in main memory, e.g. copied from GPU-memory.

The implementations of `ufunc()` and `extmethod()` simply call the Bohrium C interface with the Bohrium arrays (`.bhc_obj`).

VI. Benchmarks

In order to evaluate the performance of npbackend, we perform a number of performance comparisons between a

[3]Common Intermediate Language

```
import backend
import backend_numpy
import numpy

def dtype_name(obj):
  """Return name of the dtype"""
  return numpy.dtype(obj).name

class base(backend.base):
  def __init__(self, size, dtype, bhc_obj=None):
    super(base, self).__init__(size, dtype)
    if bhc_obj is None:
      f = eval("bhc.bh_multi_array_%s_new_empty"%dtype_name(dtype))
      bhc_obj = f(1, (size,))
    self.bhc_obj = bhc_obj

  def __del__(self):
    exec "bhc.bh_multi_array_%s_destroy(self.bhc_obj)"%dtype_name(self.dtype)

class view(backend.view):
  def __init__(self, ndim, start, shape, stride, base):
    super(view, self).__init__(ndim, start, shape, stride, base)
    dtype = dtype_name(self.dtype)
    exec "base = bhc.bh_multi_array_%s_get_base(base.bhc_obj)"%dtype
    f = eval("bhc.bh_multi_array_%s_new_from_view"%dtype)
    self.bhc_obj = f(base, ndim, start, shape, stride)

  def __del__(self):
    exec "bhc.bh_multi_array_%s_destroy(self.bhc_obj)"%dtype_name(self.dtype)

def get_data_pointer(ary, allocate=False, nullify=False):
  dtype = dtype_name(ary)
  ary = ary.bhc_obj
  exec "bhc.bh_multi_array_%s_sync(ary)"%dtype
  exec "bhc.bh_multi_array_%s_discard(ary)"%dtype
  exec "bhc.bh_runtime_flush()"
  exec "base = bhc.bh_multi_array_%s_get_base(ary)"%dtype
  exec "data = bhc.bh_multi_array_%s_get_base_data(base)"%dtype
  if data is None:
    if not allocate:
      return 0
    exec "data = bhc.bh_multi_array_%s_get_base_data_and_force_alloc(base)"%dtype
    if data is None:
      raise MemoryError()
  if nullify:
    exec "bhc.bh_multi_array_%s_nullify_base_data(base)"%dtype
  return int(data)

def set_bhc_data_from_ary(self, ary):
  return backend_numpy.set_bhc_data_from_ary(self, ary)

def ufunc(op, *args):
    args = [a.bhc_obj for a in args]
    in_dtype = dtype_name(args[1])
    f = eval("bhc.bh_multi_array_%s_%s"%(dtype_name(in_dtype), op.info['name']))
    exec f(*args)

def extmethod(name, out, in1, in2):
  f = eval("bhc.bh_multi_array_extmethod_%s_%s_%s"%(dtype_name(out), dtype_name(in1), dtype_name(in2)))
  ret = f(name, out, in1, in2)
  if ret != 0:
    raise NotImplementedError()
```

Fig. 8: A code snippet of the Bohrium backend. Note that the `backend` module refers to the implementation in figure 4 and note that the `backend_numpy` module is figure 5.

	Hardware Utilization	Matrix Multiplication Software
Native	1 CPU-core	ATLAS v3.10
NumPy	1 CPU-core	ATLAS v3.10
Numexpr	8 CPU-cores	ATLAS v3.10
libgpuarray	1 GPU	clBLAS v2.2
BohriumCPU	8 CPU-cores	$O(n^3)$
BohriumGPU	1 GPU	$O(n^3)$

TABLE II: The benchmark execution setup. Note that *Native* refers to a regular NumPy execution whereas *NumPy* refers to the backend implementation that makes use of the NumPy library.

regular NumPy execution, referred to as Native, and the four backend implementations: NumPy, Numexpr, libgpuarray, and Bohrium, referred to by their name.

We run all benchmarks, on an Intel Xeon machine with a dedicated Nvidia graphics card (Table I). Not all benchmark executions utilize the whole machine; Table II shows the specific setup of each benchmark execution. For each benchmark, we report the mean of ten execution runs and the error margin of two standard deviations from the mean. We use 64-bit double floating-point precision for all calculations and the size of the memory allocation pool (vcache) is 10 entries when applicable.

We use three Python applications that use either the NumPy module or the npbackend module. The source codes of the benchmarks are available at the Bohrium project's website[4]:

Heat Equation simulates the heat transfer on a surface represented by a two-dimensional grid, implemented using jacobi-iteration with numerical convergence (Fig. 2).

Shallow Water simulates a system governed by the Shallow Water equations. The simulation commences by placing a drop of water in a still container. The simulation then proceeds, in discrete time-steps, simulating the water movement. The implementation is a port of the MATLAB application by Burkardt[5].

Snakes and Ladders is a simple children's board game that is completely determined by dice rolls with no player choices. In this benchmark, we calculate the probability of ending the game after k-th iterations through successive matrix multiplications. The implementation is by Natalino Busa[6].

Heat Equation

Figure 9 shows the result of the Heat Equation benchmark where the Native NumPy execution provides the baseline. Even though the npbackend invertible introduces an overhead, the NumPy backend outperforms the Native NumPy execution, which is the result of the memory allocation reuse (vcache). The Numexpr achieves a 2.2 speedup compared to Native NumPy, which is disappointing since Numexpr utilizes all eight CPU-cores. The problem is twofold: we only provide one ufunc for Numexpr to JIT compile at a time, which hinders loop fusion, and secondly, since the problem is memory bound, the utilization of eight CPU-cores through OpenMP is limited.

[4]http://www.bh107.org
[5]http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/
[6]https://gist.github.com/natalinobusa/4633275

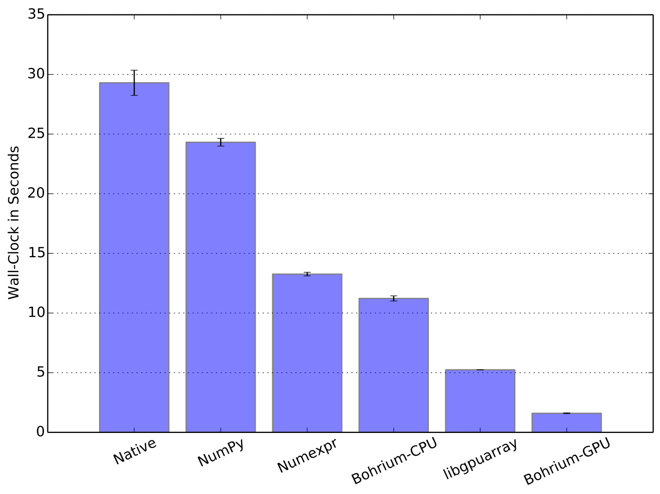

Fig. 9: The Heat Equation Benchmark where the domain size is 3000^2 and the number of iterations is 100.

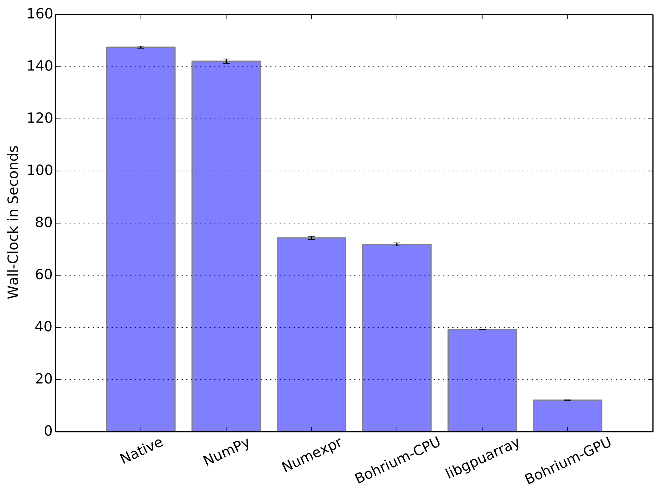

Fig. 10: The Shallow Water Benchmark where domain size is 2000^2 and the number of iterations is 100.

The Bohrium-CPU backend achieves a speedup of 2.6 while utilizing eight CPU-cores as well.

Finally, the two GPU backends, libgpuarray and Bohrium-GPU, achieve a speedup of 5.6 and 18 respectively. Bohrium-GPU performs better than libgpuarray primarily because of loop fusion and array contraction[28], which is possible since Bohrium-GPU uses lazy evaluation to fuse multiple ufunc operations into single kernels.

Shallow Water

Figure 10 shows the result of the Shallow Water benchmark. This time the Native Numpy execution and the NumPy backend perform the same, thus the vcache still hides the npbackend overhead. Again, Numexpr and Bohrium-CPU achieve a disappointing speedup of 2 compared to Native NumPy, which translate into a CPU utilization of 25%.

Finally, the two GPU backends, libgpuarray and Bohrium-GPU, achieve a speedup of 3.7 and 12 respectively. Again, Bohrium-GPU outperforms libgpuarray because of loop fusion and array contraction.

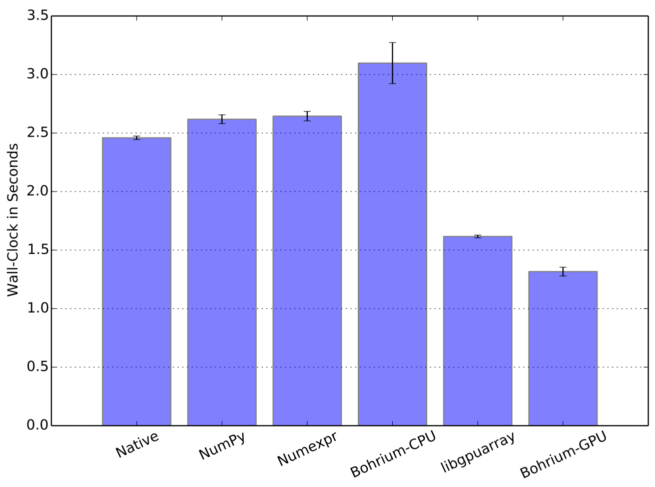

Fig. 11: The Snakes and Ladders Benchmark where the domain size is 1000^2 and the number of iterations is 10.

Snakes and Ladders

Figure 11 shows the result of the Snakes and Ladders benchmark where the performance of matrix multiplication dominates the overall performance. This is apparent when examining the result of the three first executions, Native, NumPy, and Numexpr, that all make use of the matrix multiplication library ATLAS (Table II). The Native execution outperforms the NumPy and Numexpr executions with a speedup of 1.1, because of reduced overhead.

The performance of the Bohrium-CPU execution is significantly slower than the other CPU execution, which is due to the naïve $O(n^3)$ matrix multiplication algorithm and no clever cache optimizations.

Finally, the two GPU backends, libgpuarray and Bohrium-GPU, achieves a speedup of 1.5 and 1.9 respectively. It is a bit surprising that libgpuarray does not outperform Bohrium-GPU since it uses the clBLAS library but we conclude that the Bohrium-GPU with its loop fusion and array contraction matches clBLAS in this case.

Fallback Overhead: In order to explore the overhead of falling back to the native NumPy implementation, we execute the Snakes and Ladders benchmark where the backends do not support matrix multiplication. In order for the native NumPy to perform the matrix multiplication each time the application code uses matrix multiplication, npbackend will transfer the array data from the backend address space to the user address space and vice versa. However, since npbackend uses the `mremap()` function to transfer array data, the overhead is only around 14% (Fig. 12) for the CPU backends. The overhead of libgpuarray is 60% because of multiple memory copies when transferring to and from the GPU (Fig. 7 Line 13-18). Contrarily, the Bohrium-GPU backend only performs one copy when transferring to and from the GPU, which results in an overhead of 23%.

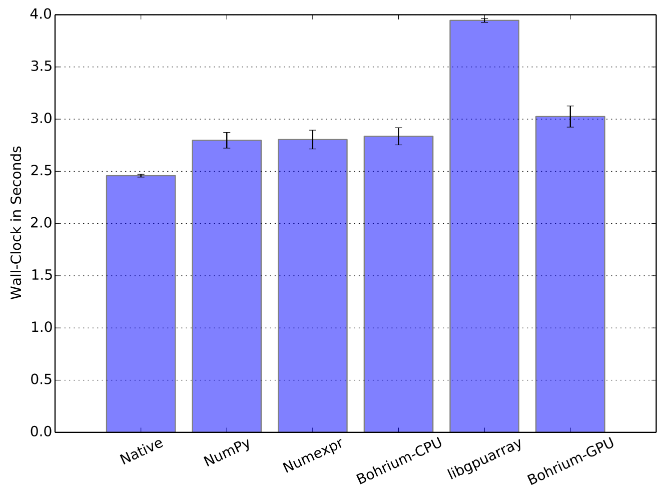

Fig. 12: The Snakes and Ladders Benchmark where the backends does not have matrix multiplication support. The domain size is 1000^2 and the number of iterations is 10.

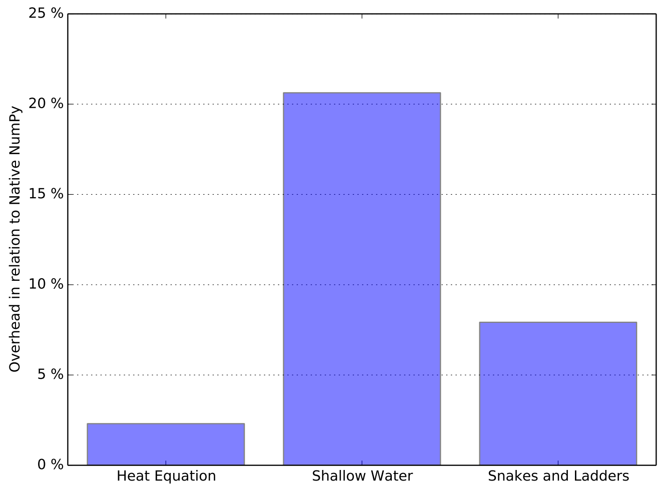

Fig. 13: Overhead of npbackend where we compare the NumPy backend with the native NumPy execution from the previous benchmarks.

Overhead

In the benchmarks above, the overhead of the npbackend is very modest and in the case of the Heat Equation and Shallow Water benchmarks, the overhead is completely hidden by the memory allocation pool (vcache). Thus, in order to measure the precise overhead, we deactivate the vcache and re-run the three benchmarks with the NumPy backend (Fig. 13). The ratio between the number of NumPy operations and the quantity of the operations dictates the npbackend overhead. Thus, the Heat Equation benchmark, which has a domain size of 3000^2, has a lower overhead than the Shallow Water benchmark, which has a domain size of 2000^2. The Snakes and Ladders benchmark has an even smaller domain size but since the matrix multiplication operation has a $O(n^3)$ time complexity, the overhead lies between the two other benchmarks.

VII. Future Work

An important improvement of the npbackend framework is to broaden the support of the NumPy API. Currently, npbackend supports array creation functions, matrix multiplication, random, FFT, and all ufuncs, thus many more functions remain unsupported. Even though we can leverage the work by the PyPy project, which re-implements a broad range of the NumPy API in Python[7], we still have to implement Extension Methods for the part of the API that is not expressed well using ufuncs.

Currently, npbackend supports CPython version 2.6 to 2.7; however there is no technical reason not to support version 3 and beyond thus we plan to support version 3 in the near future.

The implementation of the backend examples we present in this paper has a lot of optimization potential. The Numexpr and libgpuarray backends could use lazy evaluation in order to compile many ufunc operations into single execution kernels and gain similar performance results as the Bohrium CPU and GPU backends.

Current ongoing work explores the use of Chapel[29] as a backend for NumPy, providing transparent mapping (facilitated by npbackend), of NumPy array operations to Chapel array operations. Thereby, facilitating the parallel and distributed features of the Chapel language.

Finally, we want to explore other hardware accelerators, such as the Intel Xeon Phi Coprocessor, or distribute the calculations through MPI on a computation cluster.

VIII. Conclusion

In this paper, we have introduced a unified NumPy backend, npbackend, that unifies a broad range of Python code accelerators. Without any modifications to the original Python application, npbackend enables backend implementations to improve the Python execution performance. In order to assess this clam, we use three benchmarks and four different backend implementations along with a regular NumPy execution. The results show that the overhead of npbackend is between 2% and 21% but with a simple memory allocation reuse scheme it is possible to achieve overall performance improvements.

Further improvements are possible when using JIT compilation and utilizing multi-core CPUs, a Numexpr backend achieves 2.2 speedup and a Bohrium-CPU backend achieves 2.6 speedup. Even further improvement is possible when utilizing a dedicated GPU, a libgpuarray backend achieves 5.6 speedup and a Bohrium-GPU backend achieves 18 speedup. Thus, we conclude that it is possible to accelerate Python/NumPy application seamlessly using a range of different backend libraries.

References

[1] G. van Rossum, "Glue it all together with python," in *Workshop on Compositional Software Architectures, Workshop Report, Monterey, California*, 1998.

[2] T. E. Oliphant, *A Guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[7]http://buildbot.pypy.org/numpy-status/latest.html

[3] E. Jones, T. Oliphant, and P. Peterson, "Scipy: Open source scientific tools for python," *http://www. scipy. org/*, 2001.

[4] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[5] M. Sala, W. Spotz, and M. Heroux, "PyTrilinos: High-performance distributed-memory solvers for Python," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, March 2008.

[6] D. I. Ketcheson, K. T. Mandli, A. J. Ahmadia, A. Alghamdi, M. Quezada de Luna, M. Parsani, M. G. Knepley, and M. Emmett, "PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C210–C231, Nov. 2012.

[7] J. Enkovaaraa, M. Louhivuoria, P. Jovanovicb, V. Slavnicb, and M. Rännarc, "Optimizing gpaw," *Partnership for Advanced Computing in Europe*, September 2012.

[8] D. Loveman, "High performance fortran," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 1, no. 1, pp. 25–42, 1993.

[9] B. Chamberlain, S.-E. Choi, C. Lewis, C. Lin, L. Snyder, and W. Weathersby, "Zpl: a machine independent programming language for parallel computers," *Software Engineering, IEEE Transactions on*, vol. 26, no. 3, pp. 197–211, Mar 2000.

[10] W. Yang, W. Cao, T. Chung, and J. Morris, *Applied numerical methods using MATLAB*. Wiley-Interscience, 2005.

[11] C. Sanderson *et al.*, "Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments," Technical report, NICTA, Tech. Rep., 2010.

[12] T. Veldhuizen, "Arrays in Blitz++," in *Computing in Object-Oriented Parallel Environments*, ser. Lecture Notes in Computer Science, D. Caromel, R. Oldehoeft, and M. Tholburn, Eds. Springer Berlin Heidelberg, 1998, vol. 1505, pp. 223–230.

[13] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[14] M. Foord and C. Muirhead, *IronPython in Action*. Greenwich, CT, USA: Manning Publications Co., 2009.

[15] S. Pedroni and N. Rappin, *Jython Essentials: Rapid Scripting in Java*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[16] A. Rigo and S. Pedroni, "Pypy's approach to virtual machine construction," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 944–953. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176753

[17] E. Jones and P. J. Miller, "Weaveinlining c/c++ in python." OReilly Open Source Convention, 2002.

[18] D. Cooke and T. Hochberg, "Numexpr. fast evaluation of array expressions by using a vector-based virtual machine."

[19] T. Oliphant, "Numba python bytecode to llvm translator," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2012.

[20] A. Klckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU runtime code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012.

[21] A. Munshi *et al.*, "The OpenCL Specification," *Khronos OpenCL Working Group*, vol. 1, pp. 11–15, 2009.

[22] C. Nvidia, "Programming guide," 2008.

[23] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun. 2010, oral Presentation.

[24] M. R. B. Kristensen and B. Vinter, "Numerical python for scalable architectures," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 15:1–15:9.

[25] M. 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.

[26] S. A. F. Lund, K. Skovhede, M. R. B. Kristensen, and B. Vinter, "Doubling the Performance of Python/NumPy with less than 100 SLOC," in *Python for High Performance and Scientific Computing (PyHPC 2013)*, 2013.

[27] D. M. Beazley *et al.*, "Swig: An easy to use tool for integrating scripting languages with c and c++," in *Proceedings of the 4th USENIX Tcl/Tk workshop*, 1996, pp. 129–139.

[28] V. Sarkar and G. R. Gao, "Optimization of array accesses by collective loop transformations," in *Proceedings of the 5th International Conference on Supercomputing*, ser. ICS '91. New York, NY, USA: ACM, 1991, pp. 194–205.

[29] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The Cascade High Productivity Language," in *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*. IEEE Computer Society, April 2004, pp. 52–60.