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 glueing 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
backend writers (Fig. 1). The source code of both interfaces and all backend implementations is available at 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 \texttt{npbackend} that implements an array object that inherit from NumPy’s \texttt{ndarray}. The idea is that this new \texttt{npbackend-array} can be a drop-in replacement of the \texttt{numpy-array} such that only the array object in NumPy applications needs to be changed. Similarly, the \texttt{npbackend} module is a drop-in replacement of the NumPy module.

The user can make use of \texttt{npbackend} through an explicit and an implicit approach. The user can explicitly import \texttt{npbackend} instead of NumPy in the source code e.g. “\texttt{import npbackend as np}” or the user can alias NumPy imports with \texttt{npbackend} imports globally through the \texttt{-m} interpreter argument e.g. “\texttt{python -m npbackend user_app.py}”.

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

---

\(^1\)\url{http://bh107.org}

---

```python
import npbackend as np
import matplotlib.pyplot as plt

def solve(height, width, epsilon=0.005):
    grid = np.zeros((height+2, width+2), dtype=np.float64)
    grid[:,0] = -273.15
    grid[:,1] = -273.15
    grid[:,2] = -273.15
    grid[:,1] = 40.0
    center = grid[1:-1,1:-1]
    north = grid[1:-1,1:-2]
    south = grid[1:-1,2:-1]
    west = grid[1:-2,1:-1]
    delta = epsilon+1
    while delta > epsilon:
        tmp = 0.2 * (center + north + south + east + west)
        delta = np.sum(np.abs(tmp-center))
        center[:] = tmp
    plt.matshow(center, cmap='hot')
    plt.show()
```

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

Now, in order to explain what we mean by \texttt{directly}, we have to describe some implementation details of NumPy. A NumPy \texttt{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 \texttt{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 \texttt{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 \texttt{directly} we mean that Matplotlib accesses the memory segment of \texttt{center} through the C-pointer. In which case, the only option for \texttt{npbackend} is to make sure that the computed values of \texttt{center} are located at the correct memory segment. \texttt{Npbackend} is oblivious to the actual operations Matplotlib performs on \texttt{center}.

Consequently, the result of the Matplotlib call is a Python warning explaining that \texttt{npbackend} will not accelerate the operation on \texttt{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.
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 of the npbackend backend since many of them use 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 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, libgzuarray, 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 memmovers 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() function simply returns the data from the ndarray ary.

---

2Using a victim cache
ufunc_cmds = { 'add' : "il+12", 'multiply' : "il*12", 'sqrt' : "sqrt(i1)", ... }  

def ufunc(op, *args):
    args = [a.ndarray for a in args]
    il=args[1];
    if len(args) > 2:
        i2=args[2];
    out=ufunc_cmds[op].
    ...
import backend
import backendumpy
import numpy

def dtype_name(obj):
    return numpy.dtype(obj).name

class backendumpy:
    def __init__(self, size, dtype, bhc_obj=None):
        super().__init__(size, dtype)
        f = eval("bhc.bh_multi_array_%s_new_empty" % dtype_name(dtype))
        bhc_obj = f().__init__(size,)
        self.bhc_obj = bhc_obj

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

class backendumpy:
    def __init__(self, size, dtype, bhc_obj=None):
        super().__init__(size, dtype)
        f = eval("bhc.bh_multi_array_%s_new_empty" % dtype_name(dtype))
        bhc_obj = f().__init__(size,)
        self.bhc_obj = bhc_obj

    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 = self.dtype(ary)
        ary = ary.bhc_obj
        if data is None:
            if not allocate:
                return 0
        return exec("data = bhc.bh_multi_array_%s_destroy(self.bhc_obj)" % dtype_name(self.dtype))

    def set_bhc_data_from_ary(self, ary):
        return backendumpy.set_bhc_data_from_ary(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_destroy" % dtype_name(in_dtype))
        exec(f(*args)

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

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

class backendumpy:
    def __init__(self, size, dtype, bhc_obj=None):
        super().__init__(size, dtype)
        f = eval("bhc.bh_multi_array_%s_new_empty" % dtype_name(dtype))
        bhc_obj = f().__init__(size,

    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 = self.dtype(ary)
        ary = ary.bhc_obj
        if data is None:
            if not allocate:
                return 0
        return exec("data = bhc.bh_multi_array_%s_destroy(self.bhc_obj)" % dtype_name(self.dtype))

    def set_bhc_data_from_ary(self, ary):
        return backendumpy.set_bhc_data_from_ary(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_destroy" % dtype_name(in_dtype))
        exec(f(*args)

    def extmethod(name, out, in1, in2):
        f = eval("bhc.bh_multi_array_%s_destroy" % dtype_name(out))
        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 backendumpy module is figure 5.

### Table II: The benchmark execution setup

<table>
<thead>
<tr>
<th>Hardware Utilization</th>
<th>Matrix Multiplication Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>1 CPU-core ATLAS v3.10</td>
</tr>
<tr>
<td>NumPy</td>
<td>1 CPU-core ATLAS v3.10</td>
</tr>
<tr>
<td>Numexpr</td>
<td>8 CPU-cores ATLAS v3.10</td>
</tr>
<tr>
<td>libgpuarray</td>
<td>1 GPU cIBLAS v2.2</td>
</tr>
<tr>
<td>BohriumCPU</td>
<td>8 CPU-cores O(n^3)</td>
</tr>
<tr>
<td>BohriumGPU</td>
<td>1 GPU O(n^3)</td>
</tr>
</tbody>
</table>

*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 nbackend module. The source codes of the benchmarks are available at the Bohrium project’s website:

**Heat Equation** simulates the heat transfer on a surface represented by a two-dimensional grid, implemented using jacobiti-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.

**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.

**Heat Equation**

Figure 9 shows the result of the Heat Equation benchmark where the Native NumPy execution provides the baseline. Even though the nbackend 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.

---

4http://www.bh107.org
5http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/
6https://gist.github.com/natalinobusa/4635275
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 translates 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.

**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, achieve 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 cBLAS library but we conclude that the Bohrium-GPU with its loop fusion and array contraction matches cBLAS 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 claim, 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**


\(^7\)[http://buildbot.pypy.org/numpy-status/latest.html](http://buildbot.pypy.org/numpy-status/latest.html)


