# CUDArray: CUDA-based NumPy

**Larsen, Anders Boesen Lindbo**

*Publication date:*
2014

*Document Version*
Publisher's PDF, also known as Version of record

*Citation (APA):*
Larsen, A. B. L. (2014). *CUDArray: CUDA-based NumPy*. Kgs. Lyngby: Technical University of Denmark (DTU). DTU Compute-Technical Report-2014, No. 21

# CUDArray: CUDA-based NumPy

**Anders Boesen Lindbo Larsen**
Department of Applied Mathematics and Computer Science
Technical University of Denmark
abll@dtu.dk

DTU Compute Technical Report-2014-21

November 8, 2014

### Abstract

This technical report introduces CUDArray – a CUDA-accelerated subset of the NumPy library. The goal of CUDArray is to combine the ease of development from NumPy with the computational power of Nvidia GPUs in a lightweight and extensible framework.

Since the motivation behind CUDArray is to facilitate neural network programming, CUDArray extends NumPy with a neural network submodule. This module has both a CPU and a GPU back-end to allow for experiments without requiring a GPU.

## 1 Introduction

Over the last years, Python has grown steadily in popularity for scientific computing. A wealth of libraries build upon the NumPy library [15] and its powerful $N$-dimensional array class offering high-productivity and fast CPU-based numerical operations. With its large user base and time-tested usability, NumPy has become the de-facto standard for numerical computing in Python.

With the recent wave of cheap yet massively parallel processing capabilities of GPUs, it is tempting to combine the popular NumPy interface with a GPU implementation to speed up demanding numerical operations. At the time of writing, however, there exist no well-established and mature code base meeting these criteria. One likely explanation could be the workload of implementing the entire NumPy library. Moreover, NumPy allows for elaborate array operations with its *slicing* and *broadcasting* functionality. Such operations are difficult to implement efficiently on a GPU architecture where e.g. careful memory handling is crucial for obtaining good performance.

While CUDArray aims at being a drop-in replacement for NumPy, it currently imposes many limitations in order to span a manageable subset of the NumPy library. Nonetheless, CUDArray is beyond the proof-of-concept stage as it supports a state-of-the-art neural network pipeline [12].

### 1.1 Related work

There exist several GPU-based numerical Python libraries. Each library offers a different approach to combining high-level Python programming with high-performance GPU code.

*PyCUDA* [10] is a Pythonic wrapper around the *CUDA driver API*. PyCUDA supports run-time code generation for flexible CUDA programming through Python. Moreover, PyCUDA includes an array submodule containing NumPy-like functionality without adhering to the NumPy library interface.

*Bohrium* [11] is a runtime environment for vectorized computations with a NumPy front-end (among others). The front-end uses lazy evaluation to compile NumPy expressions to the runtime bytecode which is then compiled to *OpenCL* for its GPU target.

*ViennaCL* [16] is a OpenCL-based linear algebra library that comes with Python bindings. Like PyCUDA, ViennaCL does not conform exactly to the NumPy interface making it unsuitable as a drop-in replacement.

*Theano* [3] is a compiler from NumPy-like array expressions in Python to either C or CUDA code. Though array operations in Theano closely resembles those in NumPy, Theano works quite differently from the user's perspective since the array expressions must be explicitly compiled before usage.

Finally, *CUDAMat* [13] combined with *Gnumpy* [17] are closely related to the approach taken by CUDArray. CUDAMat implements common matrix operations and exposes them through a Python module without adhering to the NumPy interface. Gnumpy wraps CUDAMat operations in the NumPy interface. A notable limitation of CUDAMat is its focus on 2D arrays of data type `float`.

## 2 CUDArray features

CUDArray[1] is an open source project under the MIT license. It implements a subset of NumPy routines for array creation, array manipulation, mathematical functions, linear algebra and random sampling. Appendix A lists the NumPy operations implemented so far. CUDArray supports both Python 2 and 3.

### 2.1 Simplicity vs. feature completeness

Compared to libraries like Theano or Bohrium, CUDArray is a lightweight framework that simply maps NumPy operations directly to CUDA kernels. Moreover, CUDArray relies on the CUDA SDK-bundled libraries *cuBLAS*, *cuRAND* and *cuDNN* for high-performance implementations of critical operations such as matrix multiplications.

The simplicity of CUDArray makes it easy to extend with custom functionality since little knowledge about the framework is needed. Thus, CUDArray encourages users to rely on NumPy functionality for basic array operations and supplement these with custom CUDA kernels for more exotic operations.

Arguably, custom CUDA kernel programming is hard to avoid even with full NumPy library functionality since complex NumPy may be hard to compile to high-performance GPU code. Only very few Python libraries address this problem [11]. Instead, libraries typically allow the user to compose operations through data-parallel primitives (e.g. *scan*, *stencil* and *reduce*) [2,4–6,8].

### 2.2 CUDArray/NumPy interface

For transferring arrays from CPU to GPU memory, CUDArray implements the `numpy.array` method. Conversely, CUDArray's array class implements the `__array__` method that returns a CPU copy of an array in GPU memory. These operations comprise the CUDArray/NumPy interface without introducing new functions. Thus, developers can conveniently combine CUDArray with NumPy for any CPU-exclusive parts of a program as demonstrated in the following.

```
import numpy as np
import cudarray as ca

# Copy from CPU to GPU
a_np = np.zeros(100)
a_ca = ca.array(a_ca)

# Copy from GPU to CPU
a_ca = ca.zeros(100)
a_np = np.array(a_ca)
```

Because many NumPy functions accept objects implementing the `__array__` method, it is possible to perform memory transfers implicitly by passing CUDArray objects to NumPy. Preferably, though, memory transfers should be made explicit since they may be expensive.

### 2.3 GPU synchronization

CUDA kernels are executed asynchronously and run in parallel with the CPU code. CUDA operations like memory allocation and freeing are synchronous meaning that they block CPU operation until all previous kernels have run. CUDArray operation follows this behavior, and therefore the user should be aware that array creation and destruction forces CPU/GPU synchronization.

To achieve asynchrony, the user should be careful not to create or destroy arrays in the middle of a series of array operations. Unfortunately, dynamic array allocation is a key component in the NumPy programming style as operators consequently returns new arrays. The user must therefore avoid using operators and rather rely on the `out` method parameter as demonstrated below. That said, asynchronous CUDArray programming is rarely worth the effort unless the extra CPU resources can be put to use elsewhere.

```
import cudarray as ca

a = ca.ones(10)
b = ca.ones(10)
c = ca.empty(10)

# Synchronous code
c = a + b

# Asynchronous code
ca.add(a, b, out=c)
```

### 2.4 Speed

Inevitably, the NumPy logic introduces a computational overhead. This overhead diminishes typically when the array operations are computationally demanding. Moreover, when the user defers from using GPU synchronization operations (array creation and destruction), the NumPy logic on the CPU can even run in parallel with the array operations on the GPU.

Note that advanced optimization techniques requiring runtime code generation (e.g. *loop fusion*) is beyond

[1] CUDArray source and installation instructions are available at `http://github.com/andersbll/cudarray`

the scope of CUDArray because the framework simply maps NumPy operations directly to CUDA kernels.

For neural networks, a few expensive operations (e.g. matrix multiplication and convolution) dominate completely. CUDArray is based on similar CUDA kernels as other popular neural network libraries [3, 7, 9, 13] making it very competitive speed-wise.

### 2.5 NumPy/CPU fall-back

While Nvidia GPUs are popular on the PC market, CUDA-enabled hardware cannot be assumed available on all computers. For computers without CUDA support, CUDArray changes its back-end to NumPy by simply importing the contents of the NumPy module into the CUDArray module.

This feature is practical for performing smaller experiments on e.g. a laptop. Moreover, this is beneficial for neural networks since already trained networks can be deployed on PCs without GPUs for the prediction phase. Network prediction is relatively cheap and should run sufficiently fast on CPUs.

### 2.6 Data types

CUDArray currently supports `numpy.bool`, `numpy.int32` and `numpy.float32`. When the CUDA back-end is activated, CUDArray overrides its default data types with the above. Boolean values are represented as `int` (instead of `unsigned char` as in NumPy) to avoid implicit type conversions when operating on the values [14], While double precision support is trivial to implement, single point precision is sufficient and faster for neural networks.

### 2.7 Limitations

The NumPy library is a result of numerous man-years of effort. It is unrealistic for CUDArray to aim for a full NumPy implementation. Rather, CUDArray tries to cover the basics first and expand functionality as it becomes necessary. Apart from the operations not listed in Appendix A, CUDArray is currently limited by the following.

- Binary element-wise operations ('+', '<', etc.) can broadcast only along either contiguous *inner* or *outer* axes.
- Reduction operations ('sum', 'max', etc.) is supported on either *leading* or *trailing* axes only.
- Array indexing supports only views to contiguous memory.

### 2.8 Neural network module

CUDArray extends NumPy with specialized functionality for neural networks. These are found in the submodule `cudarray.nnet`.

**Neuron activations** Sigmoid, hyperbolic tangent and rectified linear functions.

**One-hot encoding** For encoding/decoding class labels (numbers) to the *one-hot* representation, aka. *one-of-k*.

**Multinomial logistic regression** Softmax and categorical cross entropy functions.

**Convnet operations** Convolution, pooling and local response normalization.

These operations are also implemented on the CPU to support NumPy fall-back.

## 3 Library design

Taking a pragmatic approach, CUDArray combines efficient array operation primitives in a lightweight C++ library. Functionality not provided by the libraries *cuBLAS*, *cuRAND* and *cuDNN* are implemented from scratch. The C++ library operations are then glued together in Python to imitate NumPy. The implementation of CUDArray is divided into the three parts:

**C++ library** The C++ library (named `libcudarray`) provides CUDA-based array operations. The library interface is based on pointers to device memory together with array dimensions rather than defining a separate array class. In the library interface, C++ features are used only for 1) template data types and 2) classes in rare situations where state is beneficial for an operation.

**C++ wrapper** In the C++ wrapper, *Cython* [1] is used to expose `libcudarray` functionality to Python. Cython is preferred over *ctypes* as it supports C++ templates and classes which simplifies the wrapper code.

**NumPy logic** The NumPy interface is implemented in Python on top of the C++ wrapper. Alternatively, both the C++ wrapper and the NumPy logic could have been written in Cython in order to save CPU cycles. However, Python is favored over Cython to keep the implementation simple and extensible without requiring considerable Cython knowledge.

### 3.1 Extending CUDArray

In order to add new functionality to CUDArray, the library components above should be extended. Admittedly, updating three components is a bit cumbersome but necessary in order to keep a clear separation of concerns.

In comparison with a larger framework such as Theano [3], CUDArray is easily extensible because it does not impose elaborate abstractions such as Theano’s expression graph. Additionally, dynamic code generation makes debugging hard because compiler errors are wrapped by the library adding extra complexity and slowing down the development process.

Note that when implementing missing NumPy features, one should keep in mind that not all NumPy operations (e.g. advanced indexing) are easily mapped to high-performance CUDA code and that custom CUDA kernels may be more appropriate to solve the task at hand (cf. Section 2.1).

## 4 Usage examples

The following examples demonstrate the basic functionality of CUDArray. The CUDArray-based neural network in [12] serves as a more complete example.

### 4.1 Softmax

The softmax function is used in neural networks for classification tasks. It operates on a batch of vectors stored as a 2D array $x$:

$$\mathrm{softmax}(x)_{ij} = \frac{\exp(x_{ij})}{\sum_k \exp(x_{ik})} \quad . \tag{1}$$

The NumPy/CUDArray-based implementation takes the form:

```
import cudarray as ca

def softmax(x):
    e = ca.exp(x)
    return e/ca.sum(e, axis=1, keepdims=True)
```

### 4.2 Data type handling

Data types in CUDArray works similar to NumPy:

```
import cudarray as ca

# Returns array of ca.float_
ca.ones(10)

# Returns array of ca.int_
ca.ones(10, dtype=ca.int_)
```

The user should be careful about data types when combining NumPy and CUDArray. CUDArray automatically converts NumPy’s default datatypes to CUDArray’s default data types. To avoid this conversion, the user should tell NumPy to use CUDArray’s data types:

```
import numpy as np
import cudarray as ca

# Returns array of ca.float_ converted from np.float_
ca.array(np.ones(10))

# Returns array of ca.float_ without conversion
ca.array(np.ones(10, dtype=ca.float_))
```

### 4.3 Profiling

This example demonstrates simple profiling of a CUDArray program that calls the previously defined `softmax` method:

```
import cudarray as ca

a = ca.random.uniform(size=(100, 10))
for _ in range(1000):
    softmax(a)
```

Use `nvprof` to profile the performance of the CUDA code:

```
nvprof python <filename>
```

Use Python to profile the performance of the NumPy logic (including calls to CUDA):

```
python -m cProfile -s cumtime t <filename>
```

### 4.4 Back-end selection

CUDArray checks automatically on module import if the CUDA back-end is available. If not, CUDArray falls back on its NumPy back-end by importing everything from `numpy` into `cudarray`. The user can overrule this behavior by setting the environment variable `CUDARRAY_BACKEND` to either `'numpy'` or `'cuda'` before importing CUDArray:

```
import os
# Force NumPy back-end
os.environ[’CUDARRAY_BACKEND’] = ’numpy’
# Force CUDArray back-end
os.environ[’CUDARRAY_BACKEND’] = ’cuda’
import cudarray as ca
```

## 5 Conclusion

This work introduces CUDArray – a library combining the high-productivity of NumPy with the processing power of CUDA-enabled GPUs. Although

CUDArray implements only a subset of the NumPy library, it has already shown a viable approach to building high-performance neural networks. Moreover, because CUDArray imitates NumPy, it can offer a CPU back-end almost trivially. In recognition of the fact that NumPy is too comprehensive to be implemented efficiently in its entirety, CUDArray seeks to strike a balance by providing basic functionality that can easily be mapped to high-performance code. For more exotic operations, the user is encouraged to provide custom CUDA kernels, which is relatively easy thanks to the lightweight framework.

## A CUDArray interface

At the time of writing, the CUDArray module contains the following functionality. Note that Python's standard operators are mapped to their equivalent binary/unary operations if listed below.

**Core** `array`, `bool_`, `clip`, `copyto`, `empty`, `empty_like`, `float_`, `int_`, `ones`, `ones_like`, `reshape`, `transpose`, `zeros`, `zeros_like`

**Mathematical functions** `abs`, `absolute`, `add`, `amax`, `amin`, `argmax`, `argmin`, `cos`, `divide`, `dot`, `equal`, `exp`, `fabs`, `greater`, `greater_equal`, `inner`, `less`, `less_equal`, `log`, `maximum`, `mean`, `minimum`, `multiply`, `negative`, `not_equal`, `power`, `sin`, `sqrt`, `subtract`, `sum`, `tanh`

**Random module** `random.normal`, `random.seed`, `random.uniform`

**Neural network module** `nnet.ConvBC01`, `nnet.PoolB01`, `nnet.categorical_cross_entropy`, `nnet.one_hot_decode`, `nnet.one_hot_encode`, `nnet.relu`, `nnet.relu_d`, `nnet.sigmoid`, `nnet.sigmoid_d`, `nnet.softmax`, `nnet.tanh_d`

## Bibliography

[1] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[2] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems*, October 2011.

[3] 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)*, June 2010.

[4] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, Sep 2010.

[5] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanovi, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[6] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[7] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[8] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, Aug. 2007.

[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[10] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[11] 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*, November 2013.

[12] A. B. L. Larsen. deeppy: Deep learning in Python. `http://github.com/andersbll/deeppy`, 2014.

[13] V. Mnih. CUDAMat: a CUDA-based matrix class for python. Technical Report UTML TR 2009-004, Department of Computer Science, University of Toronto, November 2009.

[14] NVIDIA Corporation. *CUDA C Programming Guide*, PG-02829-001_v6.5 edition, August 2014.

[15] T. E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[16] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*, pages 51–56, 2010.

[17] T. Tieleman. Gnumpy: an easy way to use GPU boards in Python. Technical Report UTML TR 2010-002, University of Toronto, Department of Computer Science, 2010.