CUDArray: CUDA-based NumPy

Larsen, Anders Boesen Lindbo

Publication date: 2014

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

Link back to DTU Orbit

Citation (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
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 GPU-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 adher-
ing 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 CUDAArray features

CUDAArray\(^1\) 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. CUDAArray supports both Python 2 and 3.

2.1 Simplicity vs. feature completeness

Compared to libraries like Theano or Bohrium, CUDAArray is a lightweight framework that simply maps NumPy operations directly to CUDA kernels. Moreover, CUDAArray 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 CUDAArray makes it easy to extend with custom functionality since little knowledge about the framework is needed. Thus, CUDAArray 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 little knowledge about the framework is needed. Thus, CUDAArray encourages users to rely on NumPy functionality for basic array operations and supplement these with custom CUDA kernels for more exotic operations.

2.2 CUDAArray/NumPy interface

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

```python
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 CUDAArray 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. CUDAArray operation follows this behavior, and therefore the user should be aware that array creation and destruction force 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 CUDAArray programming is rarely worth the effort unless the extra CPU resources can be put to use elsewhere.

```python
import cudarray as ca

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

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

# 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

---

\(^1\)CUDAArray source and installation instructions are available at [http://github.com/andersbll/cudarray](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 $x$:

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

(1)

The NumPy/CUDArray-based implementation takes the form:

```python
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:

```python
import cudarray as ca

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

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

4.3 Profiling

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

```python
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 override this behavior by setting the environment variable CUDARRAY_BACKEND to either ‘numpy’ or ‘cuda’ before importing CUDArray:

```python
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
CUDAArray implements only a subset of the NumPy library, it has already shown a viable approach to building high-performance neural networks. Moreover, because CUDAArray 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, CUDAArray 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 back-end almost trivially. In recognition of the fact that CUDA imitates NumPy, it can offer a CPU JIT specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.


