CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations

Ryosuke Okuta   Yuya Unno   Daisuke Nishino   Shohei Hido   Crissman Loomis
Preferred Networks
Tokyo, Japan
`{okuta, unno, nishino, hido, crissman}@preferred.jp`

Abstract

CuPy [1] is an open-source library with NumPy syntax that increases speed by doing matrix operations on NVIDIA GPUs. It is accelerated with the CUDA platform from NVIDIA and also uses CUDA-related libraries, including cuBLAS, cuDNN, cuRAND, cuSOLVER, cuSPARSE, and NCCL, to make full use of the GPU architecture. CuPy's interface is highly compatible with NumPy; in most cases it can be used as a drop-in replacement. CuPy supports various methods, data types, indexing, broadcasting, and more.

1 Introduction

NumPy is a standard tool for machine learning practitioners, researchers, and algorithm developers. Based on Python, one of the most popular programming languages, NumPy provides multi-dimensional arrays, the fundamental data structure for scientific computing, and a variety of operations and functions. Based on NumPy, the Python community has developed libraries such as scikit-learn (sklearn) [1], a machine learning library. On the other hand, parallel computing on GPUs has been increasing for the last five years in machine learning research, thanks to the popularity of deep learning. As the typical size of neural networks grows, more and more computation is required for training and applying neural networks. Deep learning computations principally require linear algebra computations, which is one of NumPy's strengths. However, NumPy does not support calculations on GPUs. This was the motivation to develop CuPy – to fully benefit from fast computations using the latest GPUs with a NumPy-compatible interface.

CuPy was first developed as the back-end of Chainer, a Python-based deep learning framework [2]. The initial version of Chainer was implemented using PyCUDA [3], a widely-used Python library for CUDA GPU calculation. However, one drawback of PyCUDA is that its syntax differs from NumPy. PyCUDA is designed for CUDA developers who choose to use Python and not for machine learning developers who want their NumPy-based code to run on GPUs. Therefore, we replaced PyCUDA and designed CuPy as NumPy-equivalent library so users can benefit from fast GPU computation without learning CUDA syntax.

CuPy has the following advantages:

High performance on NVIDIA GPUs: CuPy uses NVIDIA's CUDA and other CUDA-related libraries including cuBLAS, cuDNN, cuRAND, cuSOLVER, cuSPARSE, and NCCL to make full use of the GPU architecture.

Highly compatible with NumPy: The interface of CuPy is highly compatible with NumPy; in most cases it can be used as a drop-in replacement. Replacing NumPy with CuPy in Python code is usually enough to move computation to the GPU. It supports standard numerical dtypes, array indexing, slice, transpose, reshape, and broadcasting.

[1]Code available at https://github.com/cupy/cupy.

Easy to install: CuPy can be installed using pip and supports various versions of CUDA and cuDNN. The installer automatically detects the installed versions of CUDA and cuDNN and configures CuPy accordingly.

Easy to write custom kernels: Users can make custom CUDA kernels to run code faster, using code snippets of C++. CuPy automatically wraps and compiles the code to make a CUDA binary. Compiled binaries are cached and reused in subsequent runs.

CuPy became independent from Chainer in June 2017, when Chainer v2.0 and CuPy v1.0 were released. Since then, adoption of CuPy has expanded outside of the Chainer community to general GPU-based computations. For example, a Python-based probabilistic modeling software, Pomegranate [4], uses CuPy as its GPU backend. We believe this is thanks to CuPy's NumPy-like design and strong performance based on NVIDIA libraries.

2 Basics of CuPy

Multi-dimensional array: Since CuPy is a Python package like NumPy, it can be imported into a Python program in the same way. In the following code, `cp` is used as an abbreviation of CuPy, as `np` is often done for NumPy.

```
>>> import numpy as np
>>> import cupy as cp
```

The `cupy.ndarray` class is in the core of CuPy as a the GPU alternative of `numpy.ndarray`.

```
>>> x_gpu = cp.array([1, 2, 3])
```

`x_gpu` in the above example is an instance of a `cupy.ndarray`. Its creation is identical to NumPy syntax, except that NumPy is replaced with CuPy. The main difference of `cupy.ndarray` from `numpy.ndarray` is that the content is allocated on the GPU memory.

Most of the CuPy array manipulations are similar to NumPy. Take the Euclidean norm (a.k.a L2 norm) for example – NumPy uses `numpy.linalg.norm` to calculate it on CPU.

```
>>> x_cpu = np.array([1, 2, 3])
>>> l2_cpu = np.linalg.norm(x_cpu)
```

We can calculate it on a GPU with CuPy with:

```
>>> x_gpu = cp.array([1, 2, 3])
>>> l2_gpu = cp.linalg.norm(x_gpu)
```

CuPy implements many functions on `cupy.ndarray` objects. See the reference [2] for the supported subset of NumPy API. Since CuPy covers most NumPy features, reading the NumPy documentation can be helpful for using CuPy.

```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2, 3).astype('f')
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.sum(axis=1)
array([  3.,  12.], dtype=float32)
```

Current Device: CuPy has a current device setting, which is the default device on which the allocation, manipulation, calculation, etc. of arrays take place. The current device can be set by `cupy.cuda.Device.use()` as follows:

```
>>> cp.cuda.Device(0).use()
>>> x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
>>> cp.cuda.Device(1).use()
>>> x_on_gpu1 = cp.array([1, 2, 3, 4, 5])
```

[2]https://docs-cupy.chainer.org

To switch the current GPU temporarily, the `with` statement comes in handy.

```
>>> with cp.cuda.Device(1):
...    x_on_gpu1 = cp.array([1, 2, 3, 4, 5])
>>> x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
```

Move arrays between the CPU and GPU: `cupy.asarray()` can be used to move a `numpy.ndarray`, a list, or any object that can be passed to `numpy.array()` to the current device:

```
>>> x_cpu = np.array([1, 2, 3])
>>> x_gpu = cp.asarray(x_cpu)  # move the data to the current device.
```

`cupy.asarray()` can accept `cupy.ndarray`, which means we can transfer the array between devices with this function.

```
>>> with cp.cuda.Device(0):
...     x_gpu_0 = cp.ndarray([1, 2, 3])  # create an array in GPU 0
>>> with cp.cuda.Device(1):
...     x_gpu_1 = cp.asarray(x_gpu_0)  # move the array to GPU 1
```

`cupy.asarray()` does not copy the input array, if possible. Instead, it returns the input object itself. To force CuPy to copy the array, use `cupy.array()` with `copy=True`.

Moving a device array to the host can be done by `cupy.asnumpy()` as follows:

```
>>> x_gpu = cp.array([1, 2, 3])  # create an array in the current device
>>> x_cpu = cp.asnumpy(x_gpu)  # move the array to the host.
```

We can also use `cupy.ndarray.get()`:

```
>>> x_cpu = x_gpu.get()
```

How to write CPU/GPU agnostic code CuPy/NumPy compatibility allows CPU/GPU generic code. This can be made using the `cupy.get_array_module()` function. This function returns the appropriate NumPy or CuPy module based on whether the argument is a `cupy.ndarray` or `numpy.ndarray`. An example of a CPU/GPU generic function can be defined as follows:

```
>>> # CPU/GPU agnostic implementation of log(1 + exp(x))
>>> def softplus(x):
...     xp = cp.get_array_module(x)
...     return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

3 Supported Functionalities

3.1 Indexing

CuPy arrays support many types of useful indexing in the same syntax as NumPy indexing.

```
>>> x = cp.arange(10)
>>> print(x[3])  # int
3
>>> print(x[2:7:2])  # slice
[2 4 6]
>>> print(x[cp.array([0, 3, 5])])  # int array
[0 3 5]
>>> print(x[x % 3 == 1])  # boolean array
[1 4 7]
```

3.2 Linear Algebra

CuPy supports most linear algebra functions in NumPy using NVIDIA's cuBLAS. cuBLAS is CUDA version of a LAPACK implementation and has many linear algebra operations such as eigen decomposition, Cholesky decomposition, QR decomposition, singular value decomposition, linear equation solver, inverse of matrix and Moore-Penrose pseudo inverse. These functions are defined in `cupy.linalg` and are compatible with `numpy.linalg`.

3.3 Sorting

Unlike other operations, sort cannot be efficiently implemented by using element-wise or reduction approaches. Therefore, CuPy uses Thrust, a parallel algorithms library in C++ for better performance. With such implementation techniques, `cupy.sort` and other sort functions can be used without worrying about the internal mechanism. CuPy currently supports `sort`, `argsort`, and `lexsort`.

3.4 Sparse Matrices

CuPy supports sparse matrices using NVIDIA's cuSPARSE. These matrices have the same interfaces of sparse matrices in SciPy [5], `scipy.sparse`. Depending on their requirements, users can choose between coordinate-format sparse matrix, compressed sparse row matrix, compressed sparse column matrix, or sparse matrix with diagonal storage.

4 User-defined CUDA Kernels

For additional speed improvements, users can define kernels to optimize their programs. CuPy supports two types of commonly used kernels. One is the element-wise kernel, that applies the same operation to all data. For example, the `add` function applies a + operator for each data pair. The other is the reduction kernel, that folds all elements by a binary operator. For example, the `sum` function applies a + operator to fold all the data.

Users can define arbitrary element-wise kernels and reduction kernels with parts of CUDA code. Here is an example of user-defined element-wise kernel.

```
>>> kernel = cupy.ElementwiseKernel(
...     'float32␣x,␣float32␣y,␣float32␣z',  # input type
...     'float32␣w',                         # output type
...     'w␣=␣(x␣*␣y)␣+␣z;',                  # This is CUDA code snippet
...     'my_kernel')                         # kernel name
>>> w = kernel(x, y, z)
```

The first argument is a list of input variables, and the second one is a list of output variables. Definition of each variable consists of a type specifier and a name of an argument. The third argument is a code snippet the user wants to define. It is a code snippet of CUDA and can use arbitrary CUDA code.

CuPy also supports generic types. When a user uses a type parameter such as `T` instead of actual types `float32` for a type specifier, `ElementwiseKernel` generates a template function. It supports arbitrary types of arrays. For example, when the input type is `'T x, T y'`, this function supports all standard types such as integer and float.

Element-wise kernels and reduction kernels are similar to Map and Reduce from the MapReduce [6] framework.

5 Comparison with NumPy

Thanks to GPU processing speed, CuPy is faster than NumPy in many ways.

To benchmark the performance of CuPy compared to NumPy, the following compute environment was used for the benchmark:

- CPU: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz x 2
- Memory: 64GB
- GPU: GeForce GTX TITAN X (Maxwell) x 4

The benchmark code is as follows:

```
>>> import time, numpy, cupy
>>>
>>> size = 10 ** 8
>>> def test(xp):
...     return xp.arange(size).reshape(1000, -1).T * 2
>>>
```

Table 1: Matrix operation performance

Size	NumPy [ms]	CuPy [ms]
10^4	0.03	0.58
10^5	0.20	0.97
10^6	2.00	1.84
10^7	55.55	12.48
10^8	517.17	84.73

```
>>> for xp in [numpy, cupy]:
...     test(xp)                              # Avoid first call overhead
...     # Synchronize CPU and GPU for benchmark
...     cupy.cuda.runtime.deviceSynchronize()
...     t1 = time.time()
...     test(xp)
...     cupy.cuda.runtime.deviceSynchronize()
...     t2 = time.time()
...     print(xp.__name__, t2 - t1)
('numpy', 0.5105748176574707)
('cupy', 0.08418107032775879)
```

Table 1 shows the computation time for both NumPy and CuPy for changing the size of an input matrix. For small matrices, CuPy is slower than NumPy since there is some overhead in CuPy from the CUDA kernel launch. For larger matrices, the overhead is small compared to the actual GPU computation, and CuPy is up to six times faster than CPU-based NumPy.

Machine learning practitioners often face the problem of how to apply their algorithms to large matrices in a reasonable amount of time. This is where the benefit of CuPy over NumPy is clear.

6 Continuing Development

The CuPy development team has made improvements since its initial release, not only adding more NumPy-compatible functions, but also introducing memory pooling and kernel fusion.

6.1 Memory Pooling

It is common practice in CUDA programming to avoid `cudaMalloc` as much as possible, since memory allocation and deallocation are slow and require synchronization with the CPU. CuPy supports memory pooling from v1.0 based on a best-fit algorithm to manage free bins of different sizes (512, 1024, 1536, ...). However, cache misses can still happen if there is a memory block that is larger than the maximum size. For that case, we implemented the best-fit with coalescing (BFC) algorithm [7], so that larger free bins can be split into the necessary part, with the remaining part going back to the memory pool. This mechanism is effective for natural language processing applications, where the size of input varies. In our experiment using seq2seq model on Chainer v3.0 & CuPy v2.0, the memory usage is reduced to 25% or less.

6.2 GPU Memory Profiler

Profilers are key for efficient debugging and optimization of code. Since GPU memory is limited, "cudaErrorMemoryAllocation: out of memory" errors can be common at runtime. Though NVIDIA provides nvprof and nvvp, it is still hard to investigate what is happening inside GPU. Therefore we developed GPU memory profilers for CuPy, (`MemoryHook` and `LineProfileHook`). MemoryHook enables us to measure the number of bytes used from the memory pool of CuPy (`UsedBytes`) and those from GPU device (`AcquiredBytes`) for each function that called CuPy functions. Based on this, Chainer v3.0 supports the function-wise memory profiler (`CupyMemoryProfileHook`). Table 2 shows printed report. `LineProfileHook` also reports the number of bytes used from the memory pool of CuPy and those from GPU device at each line in the stacktrace.

Table 2: GPU function memory profiler supported by CuPy

Function Name	UsedBytes	AcquiredBytes	Occurrence
LinearFunction	5.16GB	179.98MB	3900
ReLU	991.82MB	458.97MB	2600
SoftmaxCrossEntropy	7.71MB	5.08MB	1300
Accuracy	617.97KB	351.00KB	700

6.3 Kernel Fusion

GPUs can be inefficient for small jobs, because of the overhead to call each kernel. It is better to combine small jobs to make a larger kernel. Instead of making a custom optimized kernel, users can combine small operations in Python code. This is called kernel_fusion in CuPy.

To apply kernel fusion, users append `@cp.fuse()` just before the definition of Python function that they want to combine. For functions with `@cp.fuse()`, CuPy stores a series of operations without actually executing them and compiles and invokes a single kernel that executes those operations. The generated kernel correctly handles broadcasting.

```
>>> @cp.fuse()
... def fused_func(x, y, z):
...     return (x * y) + z
...
>>> fused_func(cp.arange(10), cp.arange(10), cup.arange(10))
array([ 0,  2,  6, 12, 20, 30, 42, 56, 72, 90])
```

7 Conclusion

CuPy runs NumPy code at GPU calculation speeds. Though developed as the array back-end for the deep learning framework Chainer, it can also be used for general purpose, scientific computing on GPU. CuPy is open sourced and still growing. Though we believe that major functionalities are already supported and compatible with NumPy, contributions are always welcome. In the future, we hope more users of NumPy in the machine learning community will use CuPy for algorithm development and experiments to accelerate their research.

Acknowledgments

The authors would like to thank all of the developers of CuPy, especially Naotoshi Seo and Masayuki Takagi for their significant contributions to improve CuPy.

References

[1] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[2] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

[3] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[4] Jacob Schreiber. Pomegranate: fast and flexible probabilistic modeling in python. *CoRR*, abs/1711.00137, 2017.

[5] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 137–150, 2004.

[7] Yusuf Hasan and J. Morris Chang. A study of best-fit memory allocators. *Computer Languages, Systems & Structures*, 31(1):35–48, 2005.