CuPy

NumPy compatible GPU library for fast computation in Python

Preferred Networks
Crissman Loomis crissman@preferred.jp
Shunta Saito shunta@preferred.jp
What is CuPy?
CuPy is...

a library to provide NumPy-compatible features with GPU

import numpy as np
X_cpu = np.zeros((10,))
W_cpu = np.zeros((10, 5))
y_cpu = np.dot(x_cpu, W_cpu)

import cupy as cp
x_gpu = cp.zeros((10,))
W_gpu = cp.zeros((10, 5))
y_gpu = cp.dot(x_gpu, W_gpu)

y_cpu = cp.asnumpy(y_gpu)
y_gpu = cp.asarray(y_cpu)
```python
import numpy as np
X_cpu = np.zeros((10,))
W_cpu = np.zeros((10, 5))
y_cpu = np.dot(x_cpu, W_cpu)

import cupy as cp
x_gpu = cp.zeros((10,))
W_gpu = cp.zeros((10, 5))
y_gpu = cp.dot(x_gpu, W_gpu)

for xp in [np, cp]:
    x = xp.zeros((10,))
    W = xp.zeros((10, 5))
    y = xp.dot(x, W)
```

Support both CPU and GPU with the same code!
Why develop CuPy? (1)

- Chainer functions had separate implementations in NumPy and PyCUDA to support both CPU and GPU

Even writing simple functions like “Add” or “Concat” took several lines...
Why develop CuPy? (2)

• Needed a NumPy-compatible GPU array library
  – NumPy is complicated
    • dtypes
    • Broadcast
    • Indexing

https://www.slideshare.net/ryokuta/numpy-57587130
Why develop CuPy? (3)

- There was no convenient library
  - gnumpy
    - Consists of a single file which has 1000 lines of code
    - Not currently maintained
  - CUDA-based NumPy
    - No pip package is provided

⇒ Needed to develop it ourselves
CuPy was born as a GPU backend of Chainer.

**CuPy: Add and use a new GPU array backend with NumPy-compatible interface #266**

*beam2d merged 305 commits into master from cupy on 20 Aug 2015*

*beam2d commented on 27 Jul 2015*

This is a large PR aiming at replacing the CUDA array backend from PyCUDA/scikit-cuda to a new one named CuPy. This PR includes the implementation of CuPy and updates on Chainer.

Background: PyCUDA is a great wrapper of CUDA that enables us to write our own kernels and call them from Python. However, its GPUArray has few functionalities and almost every time we have to write our own kernels to write down Function implementations. We want to make it easier to write user-defined Functions runnable on GPU. It requires us to use more powerful GPU-array implementations.
## History of CuPy

<table>
<thead>
<tr>
<th>Date</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015/6/5</td>
<td>Chainer v1.0</td>
</tr>
<tr>
<td>2015/7/?</td>
<td>CuPy development started</td>
</tr>
<tr>
<td>2015/9/2</td>
<td>Chainer v1.3</td>
</tr>
<tr>
<td>2017/2/21</td>
<td>From PyCUDA to CuPy</td>
</tr>
<tr>
<td>2018/4/17</td>
<td>CuPy v4.0</td>
</tr>
<tr>
<td></td>
<td>Started quarterly releases</td>
</tr>
<tr>
<td></td>
<td>PyCUDA Age</td>
</tr>
</tbody>
</table>

Inside CuPy

CuPy

User-defined CUDA kernel
DNN Utility
Linear algebra
Sparse matrix
cuSOLVER
Random numbers
Sort
Multi-GPU data transfer

cuDNN
cuBLAS
cuSPARSE
cuRAND
Thrust
NCCL

CUDA
NVIDIA GPU
NumPy compatible features

- Data types (dtypes)
  - bool_, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, float64, complex64, and complex128
- All basic indexing
  - indexing by ints, slices, newaxes, and Ellipsis
- Most of advanced indexing
  - except indexing patterns with boolean masks
- Most of the array creation routines
  - empty, ones_like, diag, etc...
- Most of the array manipulation routines
  - reshape, rollaxis, concatenate, etc...
- All operators with broadcasting
- All universal functions for element-wise operations
  - except those for complex numbers
- Linear algebra functions accelerated by cuBLAS
  - including product: dot, matmul, etc...
  - including decomposition: cholesky, svd, etc...
- Reduction along axes
  - sum, max, argmax, etc...
- Sort operations implemented by Thrust
  - sort, argsort, and lexsort
- Sparse matrix accelerated by cuSPARSE
New features after CuPy v2

- Narrowed the gap with NumPy
- Speedup: Cythonized, Improved MemoryPool
- CUDA Stream support
- Added supported functions
  - From NumPy
  - Sparse Matrix, FFT, scipy ndimage support
Comparison with other libraries

<table>
<thead>
<tr>
<th>Feature</th>
<th>CuPy</th>
<th>PyCUDA*</th>
<th>Theano</th>
<th>MinPy**</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA CUDA support</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CPU/GPU agnostic coding</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Autograd support</td>
<td>***</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>NumPy compatible Interface</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>User-defined CUDA kernel</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Halted</td>
<td>2017/11 Halted</td>
<td>2018/2 Halted</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* https://github.com/inducer/pycuda  
** https://github.com/dmlc/minpy  
*** Autograd is supported by Chainer, a DL framework on top of CuPy
Projects exploiting CuPy

**Chainer**
Deep learning framework
https://chainer.org/

**pomegranate**
Probabilistic and graphical modeling
https://github.com/jmschrei/pomegranate

**spaCy**
Natural language processing
https://spacy.io/
OpenCL version of CuPy: ClPy

ClPy: OpenCL backend for CuPy

ClPy is an implementation of CuPy's OpenCL backend. In other words, ClPy enables softwares written in CuPy to work also on OpenCL devices, not only on CUDA (NVIDIA) devices.

Current status

Current ClPy is beta version, forked from CuPy v2.1.0. ClPy is still under development and works on only limited APIs.

- Most of ndarray are supported, but not perfectly
- Most of universal functions are supported, but not perfectly
- All custom kernels are supported.
- All BLAS APIs used by ClPy itself are supported. Other types are currently not.
- Sparse matrix, dnn, rand libraries are not supported
- half and complex are not supported
- Works on only a single device
Where CuPy is headed

- Support GPU in Python code with **minimal changes**
  - High compatibility with other libraries made for CPUs
  - Not only NumPy, but also SciPy etc.
- Enable GPU acceleration with **minimal effort**
  - Easy installation
  - No need for tuning
How to use CuPy
Installation

https://github.com/cupy/cupy#installation

1. Install CUDA SDK
   – If necessary, install cuDNN and NCCL too
2. (Use environment variable CUDA_PATH for custom installation)
   – setup.py of CuPy finds CUDA libraries automatically
3. $ pip install cupy
Pre-built binaries!

$ pip install cupy-cuda80 (Binary Package for CUDA 8.0)
$ pip install cupy-cuda90 (Binary Package for CUDA 9.0)
$ pip install cupy-cuda91 (Binary Package for CUDA 9.1)
$ pip install cupy-cuda92 (Binary Package for CUDA 9.2)
$ pip install cupy-cuda100 (Binary Package for CUDA 10.0)

cuDNN and NCCL included!
How much faster is CuPy than NumPy? Add funcs

```python
a = xp.ones((size, 32), 'f')
b = xp.ones((size, 32), 'f')

def f():
    a + b

# Transpose
a = xp.ones((32, size), 'f').T
b = xp.ones((size, 32), 'f')

def f():
    a + b
```

https://github.pfidev.jp/okuta/cupy-bench
Xeon Gold 6154 CPU @ 3.00GHz
Tesla V100-PCIE-16GB
How much faster is CuPy than NumPy? Dot products

For a rough estimation, if the array size is larger than L1 cache of your CPU, CuPy gets faster than NumPy.

Advanced Features

Preferred Networks
Researcher, Shunta Saito
Agenda

- Kernel Fusion
- Unified Memory
- Custom Kernels
- Compatibility with other libraries
  - SciPy-compatible features
  - Direct use of NumPy functions via __array_interface__
  - Numba
  - PyTorch via DLPack
  - cuDF / cuML
Fusion: fuse kernels for further speedup!

```python
a = numpy.float32(2.0)
x = xp.ones((1024, size), 'f')
y = xp.ones((1024, size), 'f')

def saxpy(a, x, y):
    return a * x + y
saxpy(a, x, y)  # target

@cupy.fuse()
def saxpy(a, x, y):
    return a * x + y
saxpy(a, x, y)  # target
```
Advantages of @cupy.fuse()

• Speedup function calls
• Reduce memory consumption
• Relax the bandwidth bottleneck

Limitations of @cupy.fuse()

• Only element-wise and reduction operations are supported
• Other operations like cupy.matmul() and cupy.reshape() are not yet supported
You want to save GPU memory?

```python
import cupy as cp
size = 32768
a = cp.ones((size, size))  # 8GB
b = cp.ones((size, size))  # 8GB
cp.dot(a, b)  # 8GB
```

```
Traceback (most recent call last):
  ...
cupy.cu...py.dot
OutOfMemoryError: out of memory to allocate 8589934592 bytes (total 17179869184 bytes)
```
Try Unified Memory! (Supported only on V100)

- Just edit 2 lines to enable **unified memory**

```python
import cupy as cp

pool = cp.cupy.MemoryPool(cp.cupy.malloc_managed)
cp.cupy.set_allocator(pool.malloc)

size = 32768
a = cp.cupy.ones((size, size))  # 8GB
b = cp.cupy.ones((size, size))  # 8GB
cp.cupy.dot(a, b)              # 8GB
```
Custom Kernels

• CuPy provides classes to compile your own CUDA kernel:
  – ElementwiseKernel
  – ReductionKernel
  – RawKernel (from v5)
    • For CUDA experts who love to write everything by themselves
    • Compiled with NVRTC
Basic usage of ElementwiseKernel

```python
squared_diff = cp.ElementwiseKernel(
    'float32 x, float32 y',  # input params
    'float32 z',        # output params
    'z = (x - y) * (x - y)',  # element-wise operation
    'squared_diff'       # the name of this kernel
)

x = cp.arange(10, dtype=np.float32).reshape(2, 5)
y = cp.arange(5, dtype=np.float32)

squared_diff(x, y)
```
Type-generic kernels

```python
squared_diff_generic = cp.ElementwiseKernel(
    'T x, T y',  # input params
    'T z',       # output params
    'z = (x - y) * (x - y)',  # element-wise operation
    'squared_diff'  # the name of this kernel
)

x = cp.arange(10, dtype=np.float32).reshape(2, 5)
y = cp.arange(5, dtype=np.float32)
squared_diff_generic(x, y)
```
Type-generic kernels

```python
squared_diff_generic = cp.ElementwiseKernel(
    'T x, T y',
    'T z',
    '''
    T diff = x - y;
    z = diff * diff;
    ''',
    'squared_diff_generic')

x = cp.arange(10, dtype=np.float32).reshape(2, 5)
y = cp.arange(5, dtype=np.float32)

squared_diff_generic(x, y)
```
Manual indexing with raw specifier

```python
add_reverse = cp.ElementwiseKernel(
    'T x, raw T y',  # input params
    'T z',  # output params
    'z = x + y[_ind.size() - i - 1]',  # element-wise operation
    'add_reverse'  # the name of this kernel
)

x = cp.arange(5, dtype=np.float32)
y = cp.arange(5, dtype=np.float32)

add_reverse(x, y)

=> This is same as: x + y[::-1]
```
Reduction Kernel

```python
l2norm_kernel = cp.ReductionKernel(
    'T x',          # input array
    'T y',          # output array
    'x * x',        # map
    'a + b',        # reduce
    'y = sqrt(a)',  # post-reduction map
    '0',            # identity value
    'l2norm'        # kernel name
)
x = cp.arange(1000, dtype=np.float32).reshape(20, 50)
l2norm_kernel(x, axis=1)

=> This is same as : cp.sqrt((x * x).sum(axis=1)) but much faster!
```
import cupy as cp

square_kernel = cp.RawKernel(r'''
extern "C" __global__ void my_square(long long* x) {
    int tid = threadIdx.x;
    x[tid] *= x[tid];
}
''', name='my_square')

x = cp.arange(5)
square_kernel(grid=(1,), block=(5,), args=(x,))
print(x)  # [ 0  1  4  9 16]
SciPy-compatible features: ndimage

**Interpolation**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cupyx.scipy.ndimage.affine_transform</code></td>
<td>Apply an affine transformation.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.map_coordinates</code></td>
<td>Map the input array to new coordinates by interpol:</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.rotate</code></td>
<td>Rotate an array.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.shift</code></td>
<td>Shift an array.</td>
</tr>
<tr>
<td><code>cupyx.scipy.ndimage.zoom</code></td>
<td>Zoom an array.</td>
</tr>
</tbody>
</table>

**OpenCV mode**

`cupyx.scipy.ndimage` supports additional mode, opencv. If it is given, the function performs like cv2.warpAffine or cv2.resize.
SciPy-compatible features: scipy.sparse

Sparse matrix classes

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.sparse.coo_matrix</td>
<td>COOrdinate format sparse matrix.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.csc_matrix</td>
<td>Compressed Sparse Column matrix.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.csr_matrix</td>
<td>Compressed Sparse Row matrix.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.dia_matrix</td>
<td>Sparse matrix with DIAgonal storage.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.spmatrix</td>
<td>Base class of all sparse matrixes.</td>
</tr>
</tbody>
</table>

Functions

Building sparse matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.sparse.eye</td>
<td>Creates a sparse matrix with ones on diagonal.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.identity</td>
<td>Creates an identity matrix in sparse format.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.spdiags</td>
<td>Creates a sparse matrix from diagonals.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.rand</td>
<td>Generates a random sparse matrix.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.random</td>
<td>Generates a random sparse matrix.</td>
</tr>
</tbody>
</table>

Identifying sparse matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.sparse.issparse</td>
<td>Checks if a given matrix is a sparse matrix.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix</td>
<td>Checks if a given matrix is a sparse matrix.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix_csc</td>
<td>Checks if a given matrix is of CSC format.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix_csr</td>
<td>Checks if a given matrix is of CSR format.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix_coo</td>
<td>Checks if a given matrix is of COO format.</td>
</tr>
<tr>
<td>cupyx.scipy.sparse.isspmatrix_dia</td>
<td>Checks if a given matrix is of DIA format.</td>
</tr>
</tbody>
</table>

Linear Algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cupyx.scipy.sparse.linalg.lsq</td>
<td>Solves linear system with QR decomposition.</td>
</tr>
</tbody>
</table>
Use CuPy with Numba!

```python
import cupy as cp
from numba import cuda

def square(x):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)
    for i in range(start, len(x), stride):
        x[i] **= 2

a = cp.arange(5)
square[1, 32](a)

print(a)  # => [ 0  1  4  9 16]
```
NumPy’s __array_interface__ support

- From CuPy v6.0.0 beta 2, you can pass a CuPy ndarray directory to NumPy functions!

```python
import numpy
import cupy

x = cupy.random.rand(10)  # CuPy array!
numpy.sum(x)  # Pass to a NumPy function!

# => array(4.5969301)
```
DLpack support

You can convert PyTorch tensors to CuPy ndarrays without any memory copy thanks to DLPack, and vice versa.

**PyTorch Tensor -> CuPy array**

```python
import torch
import cupy
from torch.utils.dlpack import to_dlpack

tax = torch.randn(3).cuda()  # Create a PyTorch tensor
t1 = to_dlpack(tax)  # Convert it into a dlpack tensor

cx = cupy.fromDlpack(t1)  # Convert it into a CuPy array
```
**DLpack support**

You can convert PyTorch tensors to CuPy ndarrays *without any memory copy thanks to DLPack*, and vice versa.

**CuPy array -> PyTorch Tensor**

```python
import torch
import cupy
from torch.utils.dlpack import from_dlpack

# Create a CuPy array
ca = cupy.random.randn(3).astype(cupy.float32)
t2 = ca.toDlpack()  # Convert it into a dlpack tensor

cb = from_dlpack(t2)  # Convert it into a PyTorch tensor!
```
import cupy
import cudf
import cuml

# Input data preparation
samples = np.random.randn(5000000, 2)
X = np.r_[samples + 1, samples - 1]

# Create CuPy ndarray
X_cp = cupy.asarray(X, order='F')

# Convert to cuDF DataFrame
X_df = cudf.DataFrame(
    [(str(i), cudf.from_dlpack(xi.toDlpack()))
     for i, xi in enumerate(X_cp.T)])

from cuml import KMeans
kmeans = KMeans(n_clusters=2, n_gpu=1)
kmeans.fit(X_df)
Future of CuPy
Future development plans

- [v5] @cupy.fusion()
- [v5] Raw CUDA Kernel (it replaces PyCUDA)
- [v5] Adding more compatibility: Numba, DLPack
- [v5] Windows support
- [v6] Adding more functions
- [v6] Improve memory allocation
- [v6] Speed-up kernel call
- [v6] Support more various GPUs
- ([?] CUDA Graphs support?)
Steady efforts increased speed

- How we can go closer to NumPy when allocating an array on GPU?

```
xp.empty((1024 * 1024,), dtype='b')
```
Any feedback is welcome!

- What do you use CuPy for?
- How do you use CuPy?
- What features of CuPy do you want?
- What part of CuPy do you want us to improve?

Github Issue: https://github.com/cupy/cupy/issues
#general-cupy channel in the official Slack team: https://bit.ly/join-chainer-slack
Dear CuPy users...

- Please let the NVIDIA developers and GPU technologists know the fact that you are using CuPy.
  - NVIDIA will become to support CuPy development further
- If you developed a software using CuPy, please let us know!
  - We are making a list of softwares using CuPy:
    https://github.com/cupy/cupy/wiki/Projects-using-CuPy
CuPy: NumPy-like API accelerated with CUDA (cuBLAS, cuDNN, cuRAND, cuSOLVER, cuSPARSE, cuFFT, Thrust, NCCL)

Install:
$ pip install cupy-cuda100
(replace 100 with your CUDA ver. e.g., 92 for CUDA 9.2)

Web: https://cupy.chainer.org/
Github: https://github.com/cupy/cupy/
Example: https://github.com/cupy/cupy/tree/master/examples
Forum: https://groups.google.com/forum/#!forum/cupy

Please join us and accelerate CuPy development!