Chapter 4: Scientific Computing with CuPy

CuPy Basics

CuPy is a NumPy/SciPy-compatible array library for GPU-accelerated computing with Python. CuPy acts as a drop-in replacement to run existing NumPy/SciPy code on NVIDIA CUDA or AMD ROCm platforms.

CuPy provides a multidimensional array, sparse matrices, and the associated routines for GPU devices, all having the same API as NumPy and SciPy.

The goal of the CuPy project is to provide Python users GPU acceleration capabilities, without the in-depth knowledge of underlying GPU technologies. The CuPy team focuses on providing:

- A complete NumPy and SciPy API coverage to become a full drop-in replacement, as well as advanced CUDA features to maximize the performance.
- Mature and quality library as a fundamental package for all projects needing acceleration, from a lab environment to a large-scale cluster.

The N-Dimensional Array / Cupy.ndarray data structure

The `cupy.ndarray` is the CuPy counterpart of NumPy `numpy.ndarray`. It provides an intuitive interface for a fixed-size multidimensional array which resides in a CUDA device.

This class implements a subset of methods of numpy.ndarray. The difference is that this class allocates the array content on the current GPU device.

Memory Management

CuPy uses a memory pool for memory allocation by default. The memory pool significantly improves the performance by mitigating the overhead of memory allocation and CPU/GPU synchronization.

There are two different memory pools in CuPy:

- Device (GPU) memory pool - Used for GPU memory allocation.
- Pinned (CPU) memory pool - Non-swappable memory used during CPU-to-GPU data transfer.

In most cases, CuPy users do not need to be aware of the specifics of memory allocation and deallocation, but it’s important to understand this optimization within CuPy in order to benchmark your application’s performance. You may not see memory completely deallocated due to caching in the memory pool.

CuPy provides both a high-level API to control this memory as well as a low-level API to CUDA memory management functions.

Current Device

CuPy has a concept of a current device, which is the default GPU device on which the allocation, manipulation, calculation, etc., of arrays take place (default id=0). All CuPy operations (except for multi-GPU features and device-to-device copy) are performed on the currently active device.

In general, CuPy functions expect that the array is on the same device as the current one. Passing an array stored on a non-current device may work depending on the hardware configuration but is generally discouraged as it may not be performant.

API

Cupy.ndarray

Cupy.ndarrays are the backbone of the CuPy ecosystem providing an intuitive counterpart to Numpy.ndarrays. Cupy.ndarrays, like Numpy.ndarrays, are a fixed-size multidimensional container of items of the same type and size

Cupy.ufuncs

In NumPy, a universal function (or ufunc for short) is defined as a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. In other words, ia ufunc is a “vectorized” wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs. These functions primarily operate on the NumPy array and constitute one of the most powerful ways to accelerate Python code. (see NumPy Universal Functions: https://numpy.org/doc/stable/reference/ufuncs.html)

Similarly, CuPy implements a similar ufunc also supporting broadcasting, type casting, and output type determination. Users can define Cupy.ufuncs that mimic NumPy ufuncs on Cupy.ndarray objects.

NumPy and SciPy Coverage

NumPy routines available: https://docs.cupy.dev/en/stable/reference/routines.html
SciPy routines available: https://docs.cupy.dev/en/stable/reference/scipy.html

While CuPy is designed to mimic NumPy, there are some limitations of using CuPy:

- Not all NumPy and SciPy functions are compatible with CuPy.
- CuPy may not always provide significant performance improvements.

- Performance is highly dependent on the operations performed and the hardware used.

There are also a few differences between CuPy and NumPy that might require adjustments in your code:

- Cast behavior from float to integer can be hardware dependent. This is a result of type conversion limitations within C++.
- Random function differences. The NumPy `random()` function does not support the `dtype` argument, but cuRAND, the random number generator under the hood in CuPy, does.
- CuPy handles out-of-bounds indices differently by default from NumPy when using integer array indexing. NumPy handles them by raising an error, but CuPy wraps around them.
- Matrix type (`numpy.matrix`) - SciPy returns numpy.matrix (a subclass of `numpy.ndarray`) when dense matrices are computed from sparse matrices (e.g., coo_matrix + ndarray). However, CuPy returns `cupy.ndarray` for such operations.
- CuPy arrays cannot be non-numeric like strings or objects.
- Universal Functions in CuPy only work with CuPy array or scalar. They do not accept other objects (e.g., lists or `numpy.ndarray`).
- Like Numpy, CuPy's RandomState objects accept seeds either as numbers or as full numpy arrays.
- NumPy's reduction functions (e.g. `numpy.sum()`) return scalar values (e.g. `numpy.float32`). However CuPy counterparts return zero-dimensional `cupy.ndarray`s.

There are more differences, but these are the most commonly encountered.

Coding Guide

Installation

Before setting up your CuPy programming environment, first ensure that you have fulfilled the following prerequisites:

- CUDA-compatible GPU. (see https://developer.nvidia.com/cuda-gpus for a list of NVIDIA GPUs)
- CUDA-compatible NVIDIA Drivers.
- CUDA Toolkit

The version of your CUDA Toolkit will determine the version of NVIDIA Drivers you will need to install. The CUDA Toolkit is compatible with many operating systems including Windows, Linux, and macOS, but you may need to update your OS version depending on which CUDA Toolkit release you intend to use.

See Current Installation Instructions here: https://docs.cupy.dev/en/stable/install.html

Best Practices

Before converting your program to CuPy, be sure to optimize its implementation on the CPU using NumPy and SciPy. Benchmarking your initial implementation will help you determine if you're accelerating your program when moving to the GPU.

To move your processing to CuPy from NumPy, you will need to

- Import CuPy.
- Move all calls in NumPy to CuPy.
    - CuPy covers most of the NumPy API so try this first.
- Move NumPy ndarrays to CuPy ndarrays
    - Use `cupy.array()` or `cupy.asarray()`
- Convert CuPy ndarrays back to NumPy ndarrays after GPU processing
    - Use `cupy.asnumpy()` or `cupy.ndarray.get()`

For example, this NumPy call:

```
import numpy as np
x_cpu = np.ones((1000,500,500))
```

Corresponds to this CuPy call:

```
import cupy as cp
x_gpu = cp.ones((1000,500,500))
x_cpu = cp.asnumpy(x_gpu)
```

If you are Benchmarking your code, you will need to call `cp.cuda.Stream.null.synchronize()` explicitly for timings to be fair. By default CuPy will run GPU code concurrently and the function will exit before the GPU has finished. Calling `synchronize()` makes us wait for the GPU to finish before returning.

Going Beyond NumPy and SciPy

Unfortunately, NumPy and SciPy don't necessarily provide all the functionality you will need to develop your software. In that case, you will need to know a few important patterns within CuPy:

CuPy Kernel Compilation

There are three kernel compilation classes available through CuPy. The instance of this class defines a CUDA kernel which can be invoked by the `__call__` method of this instance:

- ElementwiseKernel - Executes across each element of the array like a for-loop.
- ReductionKernel - Executes a map, reduce, and post-reduce function.
- RawKernel - Define a kernel with raw CUDA source code with control over grid size, block size, etc.

Each of these types can also be defined using the `@cupyx.jit.*` decorator counterpart: `@cupyx.jit.elementwisekernel`, `@cupyx.jit.reductionkernel`, and `@cupy.jit.rawkernel`.

CuPy Type-Generic Kernels

If type information in a kernel function is defined with one character, it is considered a *type placeholder*. The same character repeated throughout the function will be inferred as the same type. This allows the creation of re-usable generic kernels.

Moving Between GPU Devices

If you need to move data between GPU's (from device to another device), use the with statement to create a context. You may want to do this if you want to switch between the integrated graphics card and a dedicated graphics card in your system for either energy consumption or performance concerns.

```
import cupy as cp

device_id = 1

#Create context for device 1
with cp.cuda.Device(device_id):
   array_on_device1 = cp.array([1, 2, 3, 4, 5])

#Out of scope for context and execute on device 0
array_on_device0 = cp.array([1, 2, 3, 4, 5])
```

Performance Considerations

Moving Data from CPU to GPU

In order to take advantage of the GPU, we need to move data to the GPU over the PCI bus on your motherboard. This means we need to move data and code to the device and in order to execute that code. In this way, the PCI bus between the CPU and GPU can become a bottleneck.

There is a one-time performance cost to move data from the CPU to the GPU or vice versa.

Branching

Programs with many logic branches require the CPU. Switching between CPU and GPU will incur a cost that might impact performance. Programs with a lot of if-then statements might be better suited to the CPU depending on the overhead of switching between the two processors.

Make sure your function is vectorized in order to minimize branching.

Compiling Kernel Functions

When a kernel call is required, CuPy compiles a kernel code optimized for the dimensions and dtypes of the given arguments, sends them to the GPU device, and executes the kernel. CuPy then caches the kernel code sent to the GPU device within the process, which reduces the kernel compilation time on further calls.

There is a one-time performance cost to compile your kernel function.

Moving Data from Current Device

In general, CuPy functions expect that the array is on the same device as the current one. Similar to passing data from CPU to GPU or vice versa, passing an array stored on a non-current device may impact performance negatively depending on the hardware configuration.

There is a performance tradeoff when data is moved from one device to another.

Examples

The Anatomy of a Simple CuPy Program

NumPy Version on CPU:

```
import numpy as np

x_cpu = np.random.random((1000, 1000))
x_cpu *= 2
u, s, v = np.linalg.svd(x_cpu)
```

In this example,

- We create a two-dimensional numpy array populated with random numbers on the CPU
- Multiply the array by 2, which will dispatch the operation to all values in the numpy array.
- Perform a singular value decomposition operation on the array and store the unpacked output in u, s, and v.

You can easily convert this example to CuPy by switching the import statement. Now, the example will run on the GPU.

CuPy Version on GPU:

```
import cupy as cp

x_gpu = cp.random.random((1000, 1000))
x_gpu *= 2
u, s, v = cp.linalg.svd(x_gpu)
```

A Note: numpy can intelligently dispatch function calls like this one. In the above example we called `cp.linalg.svd`, but we could also call `np.linalg.svd` and pass it our GPU array and numpy would inspect it and call `cp.linalg.svd` on our behalf. This makes it even easier to introduce CuPy into your code with minimal changes.

A More Advanced CuPy Program

CuPy Example with User-defined Kernel Function:

```
import cupy
from cupyx import jit


@jit.rawkernel()
def elementwise_copy(x, y, size):
    tid = jit.blockIdx.x * jit.blockDim.x + jit.threadIdx.x
    ntid = jit.gridDim.x * jit.blockDim.x
    for i in range(tid, size, ntid):
        y[i] = x[i]


size = cupy.uint32(2 ** 22)
x = cupy.random.normal(size=(size,), dtype=cupy.float32)
y = cupy.empty((size,), dtype=cupy.float32)

elementwise_copy((128,), (1024,), (x, y, size))

elementwise_copy[128, 1024](x, y, size)

assert (x == y).all()
```

In this example:

- We start by importing CuPy and the CuPy jit (just-in-time compiler).
- The @jit.rawkernel() decorator above our function definition ensures that the user-defined kernel function is compiled the first time we call the script.
- In the body of the script, we initialize two CuPy arrays.
    - The first is a CuPy array filled with random 32-bit float values.
    - The second is an empty array of the same size.
- When we call the compiled elementwise_copy() function, we pass in our block and grid dimensions along with the size of the CuPy array. The function will loop through the first CuPy array and copy each element into the empty array.
- The last statement asserts that the copy was successful and all elements match.

Note: Our CUDA kernel function cannot return a value, so we have to move values into the output array for most kernel functions. When writing elementwise functions, also remember that the @cupyx.jit.elementwisekernel function decorator is also available for convenience.

CuPy Projects

For a list of projects using CuPy see: https://github.com/cupy/cupy/wiki/Projects-using-CuPy

References

CuPy User Guide for more information: https://docs.cupy.dev/en/stable/user_guide/index.html
CuPy API Reference: https://docs.cupy.dev/en/stable/reference/index.html
CuPy Github Repository (includes more examples): https://github.com/cupy/cupy
NumPy User Guide: https://numpy.org/doc/stable/user/
NumPy API Guide: https://numpy.org/doc/stable/reference/index.html